CS 231 Data Structures and Algorithms, Fall 2016

Dr. Bruce A. Maxwell
Department of Computer Science
Colby College

Course Description

Focuses on the common structures used to store data and the standard algorithms for manipulating them. Standard data structures include lists, stacks, queues, trees, heaps, hash tables, and graphs. Standard algorithms include searching, sorting, and traversals. Along with implementation details, students will learn to analyze the time and space efficiency of algorithms and how to select appropriate data structures and algorithms for a specific application. In homework, labs, and programming projects, students will implement their own data structures and make use of existing libraries to solve a variety of computational problems.

Prerequisites: CS 151, or Permission of Instructor

Desired Course Outcomes

A. Students understand the advantages and disadvantages of fundamental data structures and can implement them using object-oriented design principles.

B. Students understand, can implement, and can calculate the time and space efficiency of classic search, sort, and traversal algorithms, including the use of big-Oh notation.

C. Students understand the tradeoffs between different implementation of data structures and algorithms and can make appropriate design decisions based on application data requirements.

D. Students can use fundamental data structures and algorithms appropriately to solve a variety of computational problems.

E. Students can communicate the result of their work and describe an algorithm.

1 Introduction to Java

Java, as an object-oriented language, provides structures for programming that let us encapsulate both code and data/information to make it easier to design programs and re-use code for multiple purposes.

1.1 Java v. Python

Python is an interpreted language. The code that you write is read by the Python interpreter and executed as it reads the code. Java is a compiled language. The code is read by the java compiler (javac) and converted into Java bytecode, which is stored in a .class file. To execute the code, you run the java virtual machine (java) and tell it the name of the class whose main method you want to execute.

Java bytecode is not code that is directly executable by a CPU (though some native Java computers have been built). Instead, the bytecode runs on a Java Virtual Machine [JVM], which is a program that runs on your computer and translates the Java bytecode into actual machine instructions. Therefore, Java is not necessarily any faster than Python, it still has an interpreter between the code and the hardware CPU, but the intermediate compile step and the standardization of the Java bytecode can make it a bit faster.

Python also does not have types attached to variable names. If you create a variable X and assign something to X, Python has to keep track dynamically of the type of variable being held by X. Java, however, requires you to assign a type to a variable X when you define X. It is a strongly-typed language. The compiler will throw an error if your code tries to assign something to X that is not of the proper type.

In Python, everything is a class reference, even numbers and characters. In Java, there are primitive types (numbers and characters) and there are references (all classes). The two types behave a little differently, which means you need to be careful how you use them. That usage is not actually that different than Python because of the fact that the basic number and character types in Python are immutable (e.g. the value of an integer cannot changed by a reference to that integer).

Finally, both Java and Python do not require you to worry about cleaning up memory used by your program. Both languages have automatic garbage collection. That means that the Python interpreter and the JVM both watch how your program is using memory. They also keep track of which parts of memory are no longer in use. When memory gets scarce, the interpreter or JVM halts the program, clean up memory, and then proceed.

1.2 Classes

Classes provide the structure on which we build software systems. A class can encapsulate a complete program or provide a small part of a larger program's functionality. Classes enable us to divide a program into logical parts and write the parts separately. Division of a problem into manageable parts and hierarchical design are two major principles of software engineering and make large programs possible to design, debug, and modify.

The start of a class is the public/private designation followed by the name of the class. By convention Java class names start with an upper case letter and use PascalCase format: a capital letter defining each word in the name. A public designation means that other classes can create instances of and use this class. Using no designation makes a class visible only with a package, a method Java uses to bundle classes into a library.

Classes defined within other classes can also use the designations private and protected, which limit access to those classes to the enclosing class and subclasses.

Once named, a class has several major parts.

- Fields: member or class variables that hold information
- Constructors: methods that are called to initialize newly created instances of a class
- Methods: code that provides required functionality
- Internal Classes: classes used only within another class (not children)

A field can be public, private, or protected. A public field is accessible within any scope that can create an object of that class. Using public fields is generally not a good idea in object-oriented programming, as it means the implementation of a class is exposed to other programmers. That, in turn, means that if you change an implementation, it may break external code. A static field, which means there is a single instance of the variable attached to the class, not individual objects, are often made public because they are used as constants, and external programmers may need to know the value of the constant when writing code.

Making fields private means that only methods within a class have access to the field. This encapsulates the code and requires external programmers to use methods to work with an object, leaving you free to modify the implementation of a class without breaking external code.

The protected designation means that the class and any subclasses, e.g. child classes, have access to the fields. This can make subclass implementations faster and more efficient, although it ties the subclasses and parent classes together; if the parent class changes implementation, all child classes must update their code as well.

The following example shows a declaration of a public class Die with a single private int field, numFaces and two constructor methods.

```
public class Die {
  private int numFaces;

  public Die() {
    numFaces = 6;
  }

  public Die(int faces) {
    numFaces = faces;
  }
}
```

As with Python, classes have constructor functions that are called when a new instance of a class is created. In Python, the constructor for a class is the __init__ method. In Java, it is a method with the same name as the class. Unlike Python, a class can have multiple constructor functions that differ in their argument lists. The prior example contains two constructor functions for the Die class.

Like fields, methods can also be public, private, or protected, with the same limitations. Sometimes, for example, you want to have functions that are used only within a class and are dependent on the specifics of the implementation. Most methods will be public.

Static v. Non-static methods

Classes can have static and non-static methods.

Non-static methods are called using an instance of a particular class. Within a non-static method, the object from which the method was called, its fields and its methods are available via the variable `this`.

Static methods can be called without an actual instance of the object. Any `main` method must be declared static so that the program can be executed (no instance of any object exists until the program begins).

Static methods are useful for classes that are intended to encapsulate functionality but not necessarily any data (the Math class, for example). I/O classes are often designed to have static methods, and their purpose is to connect an object to a stream. Thus, `System.out.println()` is a static method of the System class. The Hello World program in Java is more complex than most, since it has to have a class, a static main method, and make use of the static println function in the System.out class.

```
public class HelloWorld {
  public static void main(String args[]) {
    System.out.println("Hello World");
  }
}
```

1.3 Java Syntax and Semantics

1.3.1 Variable Declarations

Before you can use a variable, its name and type must be declared within the current scope. The fields of a class should be declared at the top of the class, before the constructor or first method. Class fields should hold state information about a that needs to be stored between class method calls.

Local variables should, in general, be declared at the beginning of a method. Variables can be declared anywhere within a function. However, spreading variable definitions throughout a function makes it difficult to comprehend and debug. Placing all of the variable declarations in once place makes it easier to debug and to understand the total array of variables being used inside a method.

All variables must have a type. A local variable declaration is of the form: `<type> <name>;`. For example, the following declares on variable of type int (primitive type), one variable of type Integer (class form of ints), and one variable of type Die.

```
int x;
Integer y;
Die d;
```

The eight primitive types defined in Java are labels for actual memory locations that hold data: boolean, char, byte, short, int, float, long, double. While Drake claims that byte, float, long, and short are not commonly used, it depends upon the application, precision requirements, and memory availability.

All other types in Java are labels for memory locations that hold pointers to where the data may actually be stored. These are also called references.

Why does this matter?

1. Function arguments are call by value

2. Comparing primitives compares the values of the primitives
3. Comparing references does not compare the value of the references, just the pointers
4. Copying a primitive (assignment statement) copies the value of the primitive
5. Copying a reference copies the pointer to the reference

It also matters because any time you declare a variable to be a non-primitive type you have to remember that the memory for the object itself does not yet exist, only a pointer to `NULL`. You must use the `new` statement to actually allocate the space for the object. If you try to access the object prior to allocating memory for it, bad things happen.

1.3.2 Assignments

Assignments copy the value generated by evaluating the right side of an expression to the variable on the left side. If the value on the right side evaluates to a reference, then only the reference is copied. As Java is a strongly typed language, most type mis-matches in assignments will generate compiler errors. For example, the first three assignments will all generate compilation errors. The first assignment is a lossy conversion from a double (default type for floating point constants). The second assignment is a lossy conversion from a float to a double, and the third tires to assign a String reference to a double.

```
int q;
float r;
double s;

q = 3.14;
r = 3.14;
s = new String("Hi");
```

The following assignments with the same variables are valid and throw no warnings or errors.

```
q = (int)3.14; // cast fixes the error
r = 3.14f;  // designation as a float sets the right type
s = new Double(3.14); // Java silently casts to the equivalent primitive type
s = new Float(3.14); // non-lossy conversion, so Java does a silent cast.
s = new Integer(3); // non-lossy conversion, so Java does a silent cast.
```

1.3.3 Statements

Statements in Java end with a semi-colon. Statements can be variable declarations, assignments, return statements, or function calls. Blocks–functions, for loop bodies, if statement bodies, and class bodies defined by curly brackets–do not need to be closed with a semi-colon. In the last assignment, the designator f indicates that 3.14 is a float, not a double (default for constants). Without the designator, the statement requires a cast from double to float or the compiler throws an error.

```
int b;
float c;

b = 42;
c = 3.14f;
```

1.3.4 Conditionals

If statements in Java follow C syntax. The condition is specified inside parentheses. If the body of the if statement requires more than statement, then the body must be enclosed in curly brackets. An if-statement does not need to have an else if or else case, but both are possible.

```
float c = 15.0;
if( c < 15 ) {
  System.out.println("c is less than 15");
}
else if(c > 15) {
  System.out.println("c is greater than 15");
}
else {
  System.out.println("c is 15");
}
```

1.3.5 For-loops

Standard for loops also follow C syntax. The for loop syntax has three parts to it: the initialization step, the test step, and the post-loop step. It is possible to declare variables in the initialization step. The following example sums the first 10 non-negative integers. The for loop itself creates a loop variable and initializes it to 0, continues looping while the loop variable is less than 10, and increments the loop variable at the end of each loop.

```
int sum = 0;
for(int i=0;i<10;i++) {
  sum += i;
}
System.out.println("sum is " + sum );
```

1.3.6 While-loops

A while loop executes so long as its condition is true. Like conditionals and for loops, the condition is specified inside parentheses.

```
sum = 2;
while(sum < 1000) {
  sum *= sum;
  System.out.println("sum is " + sum);
}
```

1.3.7 Equality

Whenever you see a statement like:

if($A == B$)

It is essential to know what exactly is being compared. If A and B are primitive types, the actual values are being compared, and the result of the statement is likely to be exactly what you want. However, if either

A or B are references, then undesirable things can happen because the statement is comparing pointers to objects, not the objects themselves. See figure 1 for an example.

Because of the potential problems, it is wise to create an `equals()` method for each class you create that may be used in an equivalency comparison. For the reference classes belonging to the eight primitive types, the `equals()` is already defined, as shown in figure 1

1.3.8 Copying

The same issues arise when copying from one object to another. A simple assignment statement only sets the value of the memory reference, not the contents of the object. In order to set the value of one object to the value of another object, the typical practice is to make both a constructor and a `set` function that take as an argument the source object and copy the data in the source object to the target object. The code in figure 2 demonstrates the problem.

1.4 Initialization and Memory

When a local variable is declared in Java, the JVM does not initialize its value. The compiler will throw an error if it detects a variable being read prior to it being assigned a value.

When an object is created, however, any of its member fields are initialized by the JVM. Primitive type fields are initialized to a zero equivalent. Reference type fields are initialized to `null`. The value `null` is a special value for references that indicates the variable does not point to a valid object.

Java requires the programmer to allocate the memory required to create a class object. The `new` keyword is used to create a new instance of a class. Each time new is called, it allocates memory to hold the object and calls the appropriate constructor function, given the arguments. If no matching constructor function exists, the compiler will throw an error.

```
Thing t;

t = new Thing();
```

1.5 Java Libraries

Java has a large number of built-in and external packages and libraries. To import a package or library, use the `import` statement. The following example is a common import statement, with the asterisk acting as a wild-card. Importing everything in java.util is not necessarily the most efficient practice (it can take longer to compile), and it is better to be selective about which packages are necessary for a program to run.

```
import java.util.*;
```

```
public class strangeThings {

    // example of how equals does odd things with Objects
    public static void main(String[] args) {
        int a, b;
        Integer A, B;

        a = 6;
        b = 5;

        // with primitives, == works fine
        System.out.println("a: "+a);
        System.out.println("b: "+b);
        System.out.println("a == b: "+(a == b)); // prints false

        b = 6;
        System.out.println("a: "+a);
        System.out.println("b: "+b);
        System.out.println("a == b: "+(a == b)); // prints true

        // with references, things don't work so well
        A = new Integer(6);
        B = A;
        System.out.println("A: "+A);
        System.out.println("B: "+B);
        // ok here, because A and B point to the same place
        System.out.println("A == B: "+(A == B)); // prints true

        B = new Integer(6);
        System.out.println("A: "+A);
        System.out.println("B: "+B);
        // not here, because A and B point to different places
        System.out.println("A == B: "+(A == B)); // prints false
        System.out.println("A.equals(B): "+A.equals(B)); // prints true
    }
}
```

Figure 1: strangeThings class showing problems with ==.

```
public class strangeCopy {
    private int q;

    public strangeCopy(int x) {
      q = x;
    }

    public void set(int x) {
      q = x;
    }

    public int get() {
      return q;
    }

    public static void main(String[] args) {
      int a, b;
      strangeCopy A, B;

      a = 6;
      b = 5;

      A = new strangeCopy(6);
      B = new strangeCopy(5);

      System.out.println("a: "+a+"  b: "+b); // prints a: 6  b: 6
      System.out.println("a = b");
      a = b;
      System.out.println("a: "+a+"  b: "+b); // prints a: 5  b: 5
      System.out.println("a = 4");
      a = 4;
      System.out.println("a: "+a+"  b: "+b); // prints a: 4  b: 5

      System.out.println();
      System.out.println("A: "+A.get()+"  B: "+B.get()); // prints A: 6  B: 6
      System.out.println("A = B");
      A = B;
      System.out.println("A: "+A.get()+"  B: "+B.get());  // prints A: 5  B: 5
      System.out.println("A.set(4)");
      A.set(4);
      System.out.println("A: "+A.get()+"  B: "+B.get()); // prints A: 4   B: 4

      return;
    }
```

Figure 2: strangeCopy class showing problems with assignment.