**CMSC 131 Lecture Notes**

This file contains lecture notes for CMSC 131. This material need not be presented in the order given here. However, it is divided into two parts, with the assumption that each part could constitute a smaller, standalone course. The two parts, put together, comprise the full 131.

**Table of Contents**

[Part I](#7yhm3nuw5j7a)

[1) Course Introduction 3](#_30j0zll)

[2) Introduction to Eclipse 3](#_1fob9te)

[3) Brief History/Overview of Programming Languages 3](#_3znysh7)

[4) Random Access Memory (RAM) 4](#_2et92p0)

[5) Introductory Examples 5](#_tyjcwt)

[6) Primitive Types 6](#_3dy6vkm)

[7) Strings 6](#_1t3h5sf)

[8) Creating a Project in Eclipse 7](#_4d34og8)

[9) Scanner 7](#_2s8eyo1)

[10) Arithmetic operators 7](#_17dp8vu)

[11) Comparison operators (< , >, <=, >=) 7](#_3rdcrjn)

[12) Equality operators (==, !=) 7](#_26in1rg)

[13) Testing equality of objects. 7](#_lnxbz9)

[14) Conditional Statements 7](#_35nkun2)

[15) Logical operators (&&, ||, !) 8](#_1ksv4uv)

[16) Short-circuiting 8](#_44sinio)

[17) Ternary operator (?:) 8](#_2jxsxqh)

[18) Switch statements 8](#_z337ya)

[19) Choosing identifiers 8](#_3j2qqm3)

[20) Programming Errors 9](#_1y810tw)

[21) Scope of Local variables 9](#_4i7ojhp)

[22) Loops 9](#_2xcytpi)

[23) Break 10](#_1ci93xb)

[24) Continue 10](#_3whwml4)

[25) Increment/decrement operators 10](#_2bn6wsx)

[26) Augmented assignment operators (+=, -=, \*=, /=, %=) 10](#_qsh70q)

[27) Rules of precedence 10](#_3as4poj)

[28) Casting with Primitives 11](#_1pxezwc)

[29) Writing (static) methods 11](#_49x2ik5)

[30) Method overloading 11](#_2p2csry)

[31) Static Variables 11](#_147n2zr)

[32) Introduction to Objects 11](#_3o7alnk)

[33) Garbage collection 13](#_23ckvvd)

[34) Assignment operator with references 13](#_ihv636)

[35) == vs. equals (Revisited) 13](#_32hioqz)

[36) “this” 13](#_1hmsyys)

[37) Commenting 13](#_41mghml)

[38) Memory diagrams for method calls 14](#_2grqrue)

[39) Public vs. Private 14](#_vx1227)

[40) Java packages 14](#_3fwokq0)

[41) Libraries 15](#_1v1yuxt)

[Part II](#mv1qjvmiessw)

[42) Checking for correctness 16](#_4f1mdlm)

[43) Eclipse Debugger Demonstration 16](#_2u6wntf)

[44) Pseudocode 17](#_19c6y18)

[45) Software development process and systematic method design 17](#_3tbugp1)

[46) Roundoff error 18](#_28h4qwu)

[47) Exception Handling 18](#_nmf14n)

[48) Abstract Data Types (Brief Overview) 19](#_37m2jsg)

[49) Java Collection Framework 19](#_1mrcu09)

[50) Arrays 20](#_46r0co2)

[51) For-each loops 21](#_2lwamvv)

[52) Mutability 21](#_111kx3o)

[53) Deep vs. Shallow Copy 22](#_3l18frh)

[54) Privacy Leaks 22](#_206ipza)

[55) Basic Recursion 23](#_4k668n3)

[56) Java’s Wrapper classes 23](#_2zbgiuw)

[57) Java Interfaces 23](#_1egqt2p)

[58) Brief Intro to Inheritance 24](#_3ygebqi)

[59) Equals method 25](#_2dlolyb)

[60) More Advanced Recursion 26](#_sqyw64)

[61) Javadoc 27](#_3cqmetx)

**Part I**

# Course Introduction

* 1. Instructor/TA info
  2. General expectations
  3. Schedule
  4. Purpose of course
  5. Class webpage
  6. Course tools
  7. Tips for Success
     1. Don’t miss class
     2. Ask questions
     3. Come to office hours
     4. Start projects early
     5. Review items missed on quizzes/exams
     6. Do all handouts from labs and practice questions provided
     7. Study constantly (not just the night before the exam)

# Introduction to Eclipse

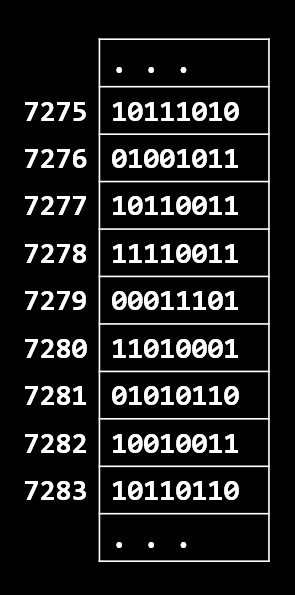
* + 1. Idea of Integrated Development Environment (IDE)
    2. Installation of Java (including the JRE and the JDK)
    3. Installation of Eclipse (including CVS plugin and UMCP plugin)
    4. Eclipse perspectives
    5. Connecting to student’s CVS repository
    6. Checking out and submitting projects (demonstration)

# Brief History/Overview of Programming Languages

* 1. What is “machine language”?
     1. Actual instructions executed by CPU.
     2. Platform specific
  2. Assembly language
     1. Pneumonic representation of machine code
     2. Very primitive
     3. Difficult to code
  3. History of some Higher Level languages
     1. Fortran (very old; used for math/science/engineering)
     2. Cobol (very old; used for business applications)
     3. Lisp (very old; interesting because it’s still used heavily in A.I.)
     4. Pascal (was used for many years to teach programming)
     5. C (very, very popular. Still used a lot. Easy to learn. Efficient.)
     6. C++ (An expansion of C to an “object oriented” language. Has some pitfalls.)
     7. Java (The language we will learn. Designed for Object Oriented programming.)
     8. Python (Increasingly popular. In some ways similar to Java. Allows several different programming approaches.)
     9. Ruby (Principle of “least astonishment”)
     10. There are MANY others. Once you learn a few, it’s pretty easy to pick up more.
  4. Why Java for us?
     1. Popular in industry
     2. Works well with Object Oriented approach.
     3. Portable.
        1. Source code is compiled into Java bytecode (which looks sort of like assembly language, but is not platform-specific, so the same byte code works for everyone)
        2. The JVM interprets the bytecode into native machine language as the program runs.
     4. Extensive and modern libraries.
     5. Good for beginners.
        1. Static typing
        2. Garbage collection
        3. Not too many “sharp edges”

# Random Access Memory (RAM)

* 1. Number base conversions
     1. Algorithm for converting from one arbitrary base to another
     2. Examples going from binary to decimal
     3. Examples going from decimal to binary
  2. Define bit (binary digit)
  3. Define byte (a sequence of 8 bits)
  4. RAM is organized as a sequence of “cells”
     1. Each cell holds a byte
     2. Cells have sequential “addresses”
     3. Show a picture



* 1. How many combinations of bits can be stored in a cell? (256)
  2. More generally, how many combinations of bits can be stored in a sequence of k bits? 2^k
  3. Define kilobyte, megabyte, gigabyte, terabyte.
  4. How is information stored in RAM as 0’s and 1’s?
     1. Non-negative integers are translated into base 2
     2. Negative integers are also translated into base 2, but by using a twist called “two’s compliment”, which you can google if you’re curious.
     3. Floating point numbers are stored by gluing together two pieces: The sequence of digits (in binary or course) and then some further bits that describe where the decimal point belongs (just like scientific notation).
     4. Text is stored by converting each character (symbol) into a numerical code, and then storing this code as binary. For example, the symbol ‘A’ is stored as 65 or 01000001. In older systems, each character was stored as a single byte, but these days most languages (like Java) use 2 bytes to get a much larger set of symbols. So ‘A’ would be stored in two cells as 00000000 010000001.

# Introductory Examples

* 1. Demonstrate a very simple example like this:

/\* This is a very basic Java program to get

things started. \*/

public class SimpleProgram {

public static void main(String args[]) {

System.*out*.print("To be ");

System.*out*.println("or not to be.");

System.*out*.println("That is the question.");

}

}

* + 1. Comments (basic purpose and syntax)
    2. Class declaration
    3. Method declaration (can’t describe all of the details yet – public, static, void, string[] args… all of these will be talked about later in the course)
    4. Proper placement of braces
    5. Indentation
    6. Statements
    7. System.out.print and System.out.prinln
  1. Do an example that illustrates local variables, like this:

/\* Have you ever noticed that your dryer eats your socks? \*/

public class VariablesExample {

public static void main(String args[]) {

int numberOfSocks;

numberOfSocks = 27; // An odd number of socks??? Crazy.

System.out.print("The number of socks is: ");

System.out.println(numberOfSocks);

System.out.println("OK, putting them in the dryer...");

int socksLostInDryer = numberOfSocks / 3; // many are lost!

numberOfSocks = numberOfSocks - socksLostInDryer;

System.out.print("The number of socks lost was: ");

System.out.println(socksLostInDryer);

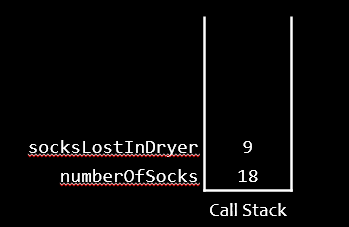
System.out.print("The number of socks is now: ");

System.out.println(numberOfSocks);

}

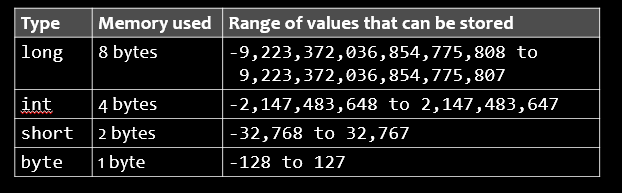
}

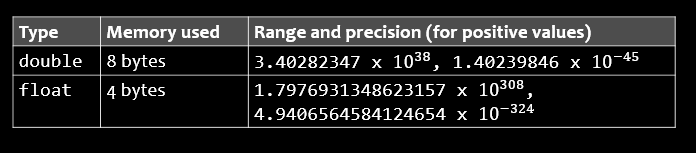
* + 1. Two local variables are declared
    2. Assignment operator
    3. Talk about Java ignoring white space
    4. Talk about declaring more than one variable in a statement
    5. Talk about Eclipse showing errors in red, warnings in yellow
    6. Draw a memory diagram like this:



# Primitive Types

* 1. Basic “atoms” of data
  2. Whole numbers



* 1. Floating point numbers
  2. Characters (char)
  3. Boolean
  4. “Literals” 17, 15L, 3.2, 5.77F, ‘A’, true, false

# Strings

* 1. Syntax for declaration
  2. Literals: “Hi”, “Byte”, etc.
  3. Strings are NOT primitives, they are objects (more about that later)
  4. Concatenation with +
     1. Implicit conversion of primitives to String when one operand is a String, e.g:

52 + “cards”

* 1. length()
  2. Distinction between char and String (‘x’ vs. “x”)

# Creating a Project in Eclipse

* 1. New Project
  2. New Class
     1. Checkbox for main method

# Scanner

* 1. Demonstrate import (without full-blown discussion of packages)
  2. Declare Scanner variable and instantiate one (without fully explaining syntax):

Scanner myScanner = new Scanner(System.in);

* 1. Demonstrate API with some examples.
     1. nextInt
     2. nextFloat
     3. nextDouble
     4. next
     5. nextLine
     6. close

# Arithmetic operators

* 1. Quickly mention +, -, \*, /
  2. Explain modulus operator (%). Do many examples (with non-negative numbers only)
  3. Mention precedence of these operators
  4. Demonstrate integer division (truncates)

# Comparison operators (< , >, <=, >=)

* 1. At first just say the “return” a boolean value (true/false)

# Equality operators (==, !=)

# Testing equality of objects.

* 1. Demonstrate that == does not compare as expected (don’t talk about memory diagram yet).
  2. Show syntax for equals (without much explanation, just as a way of comparing two objects, such as Strings)

a.equals(b) // returns true or false

# Conditional Statements

* 1. If statements (syntax, discuss, examples)
  2. If-else statements (syntax, discuss, examples)
  3. Explain that braces are actually optional in cases where block contains just one statement (but mention that it’s better to always include the braces).
  4. Nesting if and if/else statements (do a few examples)
  5. “else if” style
     1. Explain that this is not a language feature, but a different style
     2. Demonstrate by taking nested if/else statements and transforming into “else-if” style by removing braces and white-space
     3. Do a couple examples
     4. Be sure to characterize the set of cases where “else if” style can be applied

# Logical operators (&&, ||, !)

* 1. Do some examples.

# Short-circuiting

* 1. Do examples with both && and ||

# Ternary operator (?:)

* 1. Syntax and semantics
  2. Describe cases where it applies
  3. Do a couple examples

# Switch statements

* 1. Syntax
  2. Variable must be one of these: int, short, byte, char, String
  3. Cases can only be literals
  4. Discuss “break” and what happens with/without it
  5. Discuss “default” case
  6. Do some examples

# Choosing identifiers

* 1. Syntax rules
     1. Available symbols (A-Z, a-z, 0-9, \_, $)
     2. Cannot begin with digit
     3. Must avoid keywords
  2. Style
     1. Lower camel-case for variables, methods
     2. Upper camel-case for classes
     3. Use meaningful names!
        1. Some abbreviation is OK, but not too much
        2. Avoid names like “temp”, “x2”, “a”
     4. Named Constants
        1. What are they and why do we care?
           1. Facilitates easier code maintenance (if value changes)
           2. More readable
        2. Use all-caps with underscores, e.g: SPEED\_OF\_LIGHT
        3. Introduce “final”
        4. Do some example(s)

# Programming Errors

* 1. Syntax
  2. Semantic
  3. Logical
  4. Debugging
     1. Later we’ll learn to use the Eclipse “debugger”
     2. For now, put lots of “trace outputs” into the code
  5. Advice: Write code incrementally in small parcels and test thoroughly at each stage.

# Scope of Local variables

* 1. Talk about blocks and block scope
  2. Demonstrate how Eclipse can get confused about variables not being initialized. For example:

int x = myScanner.nextInt();

int y;

if (x == 0) {

y = 3;

}

If (x != 0) {

y = 7;

}

System.out.println(y); // Will display an error in Eclipse!

# Loops

* 1. Talk about repetition. Mention iteration and recursion (but don’t teach recursion yet!)
  2. While loops (syntax, semantics, some examples)
  3. Do-While loops (syntax, semantics, some examples)
  4. For-Loops (syntax, semantics, many examples)
  5. Demonstrate stopping an infinite loop in Eclipse
  6. Nested loops. Do a lot of examples here. Suggestions:
     1. Prompt user for length, width then draw a multiplication table of that size
     2. Draw a triangle of asterisks
     3. Draw a checkerboard pattern (ASCII art)
     4. Guessing game (Computer chooses random value from 1-100, user has to guess what it is. Computer says one of “higher, lower, correct” and then repeats, if necessary. At the end, ask if user wants to play again.)
     5. One example with three techniques: Draw this ASCII-art of size n (n is selected by user). Below is case where n = 5.

XOOOO

XXOOO

XXXOO

XXXXO

* + - 1. Implement using a “formula” to compute number of O’s on current row
      2. Implement using a variable to keep track of number of O’s on current row
      3. Use a comparison for each position to decide whether to draw an X or an O.

# Break

* 1. Syntax
  2. semantics
  3. examples
  4. Be sure to mention that break is not ideal style.

# Continue

* 1. Syntax
  2. semantics
  3. examples
  4. Be sure to mention that continue is not ideal style.

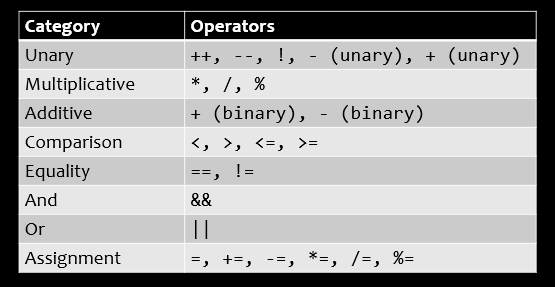
# Increment/decrement operators

* 1. Define “side effect”
  2. Define “value”
  3. Syntax for ++
  4. Explain difference between pre-increment and post-increment: They have the same side effect but carry different values
  5. Talk about – (same thing, just decrementing)
  6. Do some examples

# Augmented assignment operators (+=, -=, \*=, /=, %=)

# Rules of precedence

* 1. Show table:



* 1. Explain with examples
  2. Encourage use of (sometimes superfluous) parentheses for clarification
  3. Almost all operators are left-to-right associative
  4. Assignment operators are right-to-left associative

# Casting with Primitives

* 1. Suggest the following hierarchy of “wideness”:

double

float

long

int

short

byte

* 1. Show cases where Java will implicitly convert (assigning a value that is “narrower”):

double y = 7;

* 1. Show cases where Java will not convert (assignment a value that is too “wide”):

int x = 33.877;

* 1. Show explicit conversion:

int x = (int)33.877;

* 1. Explain truncation and how data is typically lost when using explicit casting.

# Writing (static) methods

* 1. Explain how writing methods can eliminate redundancy
  2. Talk about syntax (use static for now)
  3. Explain syntax and semantics of method calls
  4. Talk about parameters and arguments
  5. Talk about return types and return values
  6. Do many examples.
     1. Be sure to include one with a method calling a method that calls a method.
     2. Be sure to include one with multiple parameters
     3. Be sure to include one that calls a method from within a loop

# Method overloading

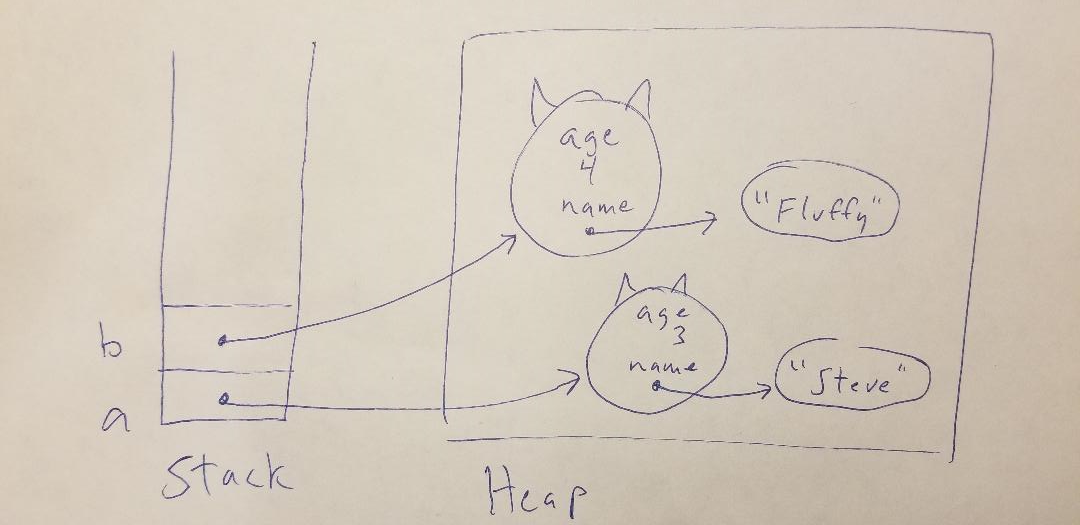
* 1. Examples of correct method overloading
  2. Examples of incorrect attempts to overload (with matching parameter lists)

# Static Variables

* 1. Syntax for declaring
  2. Explain very clearly that there is always ONE copy of these in memory

# Introduction to Objects

* 1. Explain use of Objects in programs (in abstract)
     1. Talk about how objects have “state” and “behaviors”
  2. Role of classes as “blueprints” for a kind of object
  3. Instance variables for state
     1. Explain as clearly as you can that each “instance” of the class will have its own copy of these variables
  4. Instance methods for behaviors
     1. Explain as clearly as you can that in order to invoke these methods there must be a “current object”, which is an instance of the class.
  5. Talk about how we can now use the name of our class as the type of a variable, and how we use “new” to create instances (objects) of the class.
  6. Demonstration:
     1. Write a simple class (perhaps “Cat”) that contains some instance variables and instance methods
        1. Be sure that you demonstrate using “this” in the instance methods to access that state of the current object
     2. Write another class with a main method
        1. Create Cat variables
        2. Assign instances of the Cat class (explain “new”)
        3. Show how to use the dot operator to access members (both data and methods)
        4. Talk very clearly about how the object invoking the method will be the current object (“this”) while the method is running. They really struggle with this concept!
        5. Talk about how reference variables do not actually store an object; they store they memory address of the Object (where it is in RAM). Draw a memory diagram like this:



* + 1. Illustrate how “this” can be used to access both state and behaviors of the current object. Be sure to do an example where “this” is passed as an argument to a method.
    2. Talk about constructors.
       1. Add a typical constructor to the Cat class.
       2. Show, by editing your main method, how the constructor is invoked. (As you write the other constructors, described below, add statements to the main method to demonstrate invoking them all.)
       3. Mention how the “default” constructor supplied by Java is no longer available once you’ve written a constructor of your own.
       4. Write a few other constructors
       5. Write a no-arg constructor
       6. Explain the purpose of a copy constructor and write one.
       7. Show the common style of using parameters with the same names as the instance variables (and using “this” to do the usual assignments).
       8. Show how we can use “this” to have a particular constructor invoke another constructor from the same class.
          1. Mention that the invocation must be the first statement
          2. Only one such invocation can be supplied
    3. Implement a toString method for the Cat class. Demonstrate how Java will implicitly call toString sometimes.
  1. Mention that Strings are unique in that we don’t always use “new” to instantiate one.

# Garbage collection

* 1. Talk about Java’s garbage collection
  2. Explain with memory diagrams

# Assignment operator with references

* 1. Using memory diagrams, explain how using assignment with references creates “aliasing” of variables.
  2. Can we make a copy instead?
     1. Mention clone (covered in 132)
     2. Mention and show syntax for call to “copy constructor”. We will learn this later (in 131).

# == vs. equals (Revisited)

* 1. Demonstrate with memory diagrams how the == operator (when used with references) is really checking for aliasing.
  2. Review the equals method (the syntax now makes sense) and explain that its purpose is to compare the state of the two objects.

# “this”

* 1. Elaborate on using “this” both to access state/behaviors of the current object.
  2. Do an example where “this” is passed as an argument to a method.

# Commenting

* 1. Describe the purpose and review the syntax for providing comments.
  2. There should be a comment at the top of every class describing the general purpose of the class.
  3. There should be comments above all methods describing the full contract for the method.
     1. Mention that frequently the Javadoc utility is invoked and so many times users of the class will rely on these comments without actually being able to inspect the code directly. (We will demonstrate the Javadoc utility later in the course.)
     2. These comments should be very complete.
        1. Pre-conditions
        2. Post-conditions
  4. There should be comments near most (but not all) variable declarations, describing the role of the variable.
  5. There should be comments scattered throughout the code in places where it would help the reader to understand the flow of the code. Give examples.

# Memory diagrams for method calls

* 1. Talk about the Abstract Data Type “Stack”
     1. Push
     2. Pop
  2. Explain that the call-stack is really a collection of “frames”, not variables. (Each method call results in a frame being pushed onto the stack.)
  3. Show a memory diagram and describe a hypothetical example involving a chain of method calls, demonstrating frames being pushed/popped.
  4. Show a coding example including a method with two parameters: a primitive and a reference variable. Show with a memory diagram that when the method is called, the local copy of the primitive variable cannot be modified, but that (due to aliasing), the state of the object on the heap can be modified by the method call. (This is called a “destructive” method.)
  5. Adfasdf

# Public vs. Private

* 1. Define API (Application Programming Interface) for a class, and explain how “users” of the class rely on just a subset of the features of the class.
  2. Talk about Java visibility specifiers public and private.
  3. Explain that API includes public members, including not just method prototypes, but also the “contracts” of the methods.
  4. Talk about data and procedural encapsulation.
  5. Demonstrate data encapsulation with an example.
     1. Include private instance variables.
     2. Include and discuss the purpose of public getters and setters.
     3. Talk about how having private fields allows us to enforce a “policy” on the state of the object, e.g.: The weight can never exceed 300 pounds. (Public constructors and mutators can now be modified to enforce such a policy.)
     4. Make some kind of substantial change to the class (like changing a field from an int to a String), and talk about how advantageous it is if we can modify the class without actually changing the API that is exposed to users of the class.
     5. Conclude that maintaining/modifying a particular class can often be performed without having to re-code other classes if the modules are “loosely coupled”. Having a small, simple API facilitates this.

# Java packages

* 1. Hierarchy of packages for organization of code
  2. Syntax for specifying fully qualified names (with dots)
  3. Declaring a package at the top of the class (done automatically in Eclipse).
  4. Accessing members of other classes
     1. When class is in the same package
     2. When class is in a remote package
        1. Using a “fully qualified name”
        2. Using import
     3. Import with wildcard. (Discourage this, due to potential naming collisions).
     4. Mention that java.lang is imported automatically

# Libraries

* 1. Describe the notion “library”
  2. Show online documentation for Java libraries
  3. Look at a few particular classes and discuss how to interpret the documentation. (Math and String classes are suggested as potential candidates.)

**Part II**

If Part II is taught as a stand-alone course, begin with the usual course introduction.

# Checking for correctness

* 1. Talk about “Formal Verification”
     1. Formal specification
     2. Proof of correctness (sometimes automated)
  2. Testing
     1. Describe unit vs. integrated testing. We will be focusing on unit testing.
     2. Testing is hardly ever “perfect”. (There will be program flows and/or input cases that haven’t been tried.)
     3. As many different types of cases should be tested as possible.
     4. Corner cases.
     5. JUnit
        1. Describe the API for a broken class that you have written in advance.
        2. Show how to write some JUnit test cases for your class.
           1. assertTrue
           2. assertFalse
           3. assertEquals
        3. Demonstrate running JUnit and talk about the color-coded output.
        4. Explain that when an assertion fails, the test case is aborted.
        5. Failure could be due to an assertion failing or an exception being thrown.
        6. Failing one test does not prevent the others from running.
        7. JUnit tests are run in an arbitrary order, so tests must be independent of each other
     6. Talk about how test code should always be kept with production code indefinitely (so subsequent maintenance/modifications can be re-tested, as needed.)

# Eclipse Debugger Demonstration

* 1. Provide a class (written in advance) with several buggy methods to use as a basis for the demonstration.
  2. Write some JUnit tests, one-by-one, and run them as you go. For tests that fail, demonstrate the debugging process.
     + 1. Place a breakpoint at the top of the test.
       2. Step over the code until you see something go amiss in the variables window.
       3. Stop the debugger and then restart it, but this time step into the method call that seems to be causing the problem.
       4. Once inside the method call, step over statements to try to identify what has gone wrong.
       5. Demonstrate fixing the problem with a good “guess”, and then running the JUnit test again to see if your fix seems to have corrected the issue.
  3. Show how breakpoints can be added or removed during a debug session.
  4. Demonstrate the “continue” button (that runs the program again until the next breakpoint)
  5. Talk about how toString gets used to show the states of objects conveniently in the variables window (so we should be in the habit of supplying a toString method with our classes).
  6. Show how arrays appear in the variables window.
  7. Demonstrate setting a conditional breakpoint in a loop.

# Pseudocode

* 1. Discuss difficulties in coming up with an algorithm while simultaneously worrying about language details
  2. Define pseudocode as a language-independent way of fleshing out the details of an algorithm: “English phrases” mixed with programming jargon, organized like code.
  3. Give a few examples where an algorithm is sought, and pseudocode is used to help figure out the details without using a formal language.

# Software development process and systematic method design

* 1. General software development process (iterative approach)
     1. Identify the components that make up the system.
     2. Figure out and analyze use cases that the system should be able to handle
     3. Determine the program modules (classes) and their APIs (interfaces). How will the components interact? Which components rely on each other?
     4. Build a basic prototype that features the essential/core components. (Just enough to be able to run and demonstrate the primary functionality)
     5. Iteratively refine this prototype so that it comes closer and closer to the desired product.
  2. Systematic Method (function) design
     1. Define data types/structures
        1. Figure out what kind of data must be represented
        2. Identify which primitives/classes/structures can be used for this kind of data
     2. Define Signature and contract
        1. Types of parameters for input
        2. Return type (or side effects) for output
        3. Document the “purpose” of the method. What does it do?
        4. Create a “stub” for the method (to be filled in)
     3. Come up with cases (examples)
        1. Determine the spectrum of input cases that may occur.
        2. Write down what the output should be for various inputs, E.g. foo(7) → orange
        3. Think about edge/corner cases
        4. Optionally, implement actual test cases (JUnit) now.
     4. Create an outline/template for the method
        1. Come up with an “inventory” of what is available at the start of the method. In Java, this may include parameters, (public) static data members, and the current object in an instance method.
        2. Write down (perhaps in pseudocode) the basic algorithm/steps the method should follow.
     5. Implementation
        1. Rely heavily on previous two steps
     6. Testing
        1. Based on cases from step iii
  3. Do a good example illustrating the steps of systematic method design.

# Roundoff error

* 1. Demonstrate with an easy example such as:

int x = 1.1, y = 2.2;

int z = x + y;

assertTrue(z == 3.3);

* 1. Talk about how binary representation using a finite number of bits (such as 32) frequently results in truncation.
  2. Show how to use a “machine-epsilon” value to test whether or not the result of a computation matches the theoretical expectation.
  3. Suggest that whenever possible it is desirable to use integers for arithmetic operations, since there is no roundoff error (but don’t forget about division resulting in truncation). Mention that there is still a possibility of “overflow error”!

# Exception Handling

* 1. Talk about runtime errors. In the context of a programming working on a method “in isolation”, what can be done in cases where the method is unable to fulfill its “contract” due to some unforeseen circumstance, such as a hard drive failure or another “exceptional” event.
  2. What can be done?
     1. Display an error and terminate the program. (Obviously a naïve and terrible idea.)
     2. Return an “error code”. Not uncommon. Relies on the user of the method not accidentally processing the return value as an ordinary value. (Can lead to mysterious errors.)
     3. “Throw” an exception.
        1. Show syntax.
        2. Describe the notion of an exception as a “signal” that the method has failed.
        3. Briefly go over some of the variety of different Exception classes that are available.
        4. Describe how the JVM is now looking for a “handler” for this exception
           1. Talk about how the JVM goes down each frame in the call-stack one-by-one looking for a “handler”. (We’ll describe how a handler is implemented soon.) If one is found, it runs. If one is not found, the current frame is popped off the stack and the JVM then looks to the next one down (the method that called this one), etc.
           2. If no handler is found, then the program terminates (in Eclipse we see the red and blue “stack trace” in the console).
  3. How do we write a “handler”? (try/catch)
     1. Explain that the method call in which the exception is being thrown must be enclosed in a try block.
     2. Show the syntax for supplying a single catch block.
        1. Explain the role of the variable the refers to the exception object.
     3. Explain the semantics, going over the sequence of events for cases:
        1. The typical case where no exceptions are thrown (so catch block is ignored).
        2. The case where the type of the exception thrown within the try block matchs the type being caught.
           1. The handler runs.
           2. The program now begins executing statements after the catch block.
        3. The case where the type of the exception thrown during the try block does not match the one being caught. (The current frame is popped off the stack and Java looks for an appropriate handler in the method that called this one.)
        4. The case where an exception is thrown before the try block begins (in which case our catch block is irrelevant.)
  4. Talk about places where exceptions are thrown
     1. In your own code.
     2. In code you are calling (perhaps in a library or written by a co-worker).
     3. Thrown by the JVM internally.
  5. Go over some examples of code that was presented in class (or was part of a project) previously where it would have made sense to throw exceptions.
  6. Finally block
     1. Show syntax
     2. Describe purpose. (Ensuring that some “mission critical” code is completed in case the current process terminates abruptly. Often used for “cleaning up” loose ends, or to prevent things being left in a state that is not “well-defined”.
     3. Talk about semantics (if the try block has begun then the finally block will run no matter what happens.)
     4. Go over cases:
        1. If try block has started and no exceptions are thrown then the finally block runs.
        2. If try block has started and an exception is thrown and caught locally, then the finally block runs.
        3. If try block has started and an exception is thrown but not caught locally then (on the way out) the finally block still runs. (!)
        4. If try block has started and a return statement is executed, then the finally block still runs. (!)
  7. Do at least one good example where students must trace the flow of various cases in which exceptions are sometimes being thrown, sometimes being caught other times not being caught.

# Abstract Data Types (Brief Overview)

* 1. Briefly talk about the following:
     1. Linear data types
        1. List
        2. Stack
        3. Queue
     2. Trees
     3. Graphs
     4. Sets

# Java Collection Framework

* + 1. Explain that raw arrays are one way to create a simple collection, but that there are more sophisticated tools available for many data types. We will just touch on it here, much more will be taught in CMSC 132.
    2. Descibe the ArrayList class.
    3. Talk about the need for generic notation. (Allowing collections that can store anything at all can lead to run-time errors in cases where the wrong kind of thing is accidentally inserted into the collection – these bugs may be hard to pin down. Specifying a specific type in the beginning results in compiler errors instead, which are easy to find.)
    4. Show syntax (using generic notation) for creating an ArrayList variable and instantiating an empty ArrayList:

ArrayList<Cat> a = new ArrayList<>();

* + 1. Show the API documentation for ArrayList and talk about the common methods.
    2. Do an example of a small program that uses an ArrayList.

# Arrays

* 1. Motivate need for storing collections of data.
  2. Very quickly mention the availability of the Java Collection Framework (we’ll learn it later).
  3. Describe an array as a sequential list of values, stored contiguously.
  4. Arrays of primitives.
     1. Show syntax for declaring an array variable.
     2. Show syntax for instantiating an array of primitives.
     3. Show the corresponding memory diagram
        1. Arrays are objects in Java, so they go on the heap.
        2. Draw a row of boxes on the heap.
        3. Mention that Java will automatically initialize the elements of the array with default values (0, 0.0, false, (char)0, or null).
     4. Talk about how elements are indexed (0-based)
     5. Show how to use square brackets to access elements individually.
     6. Mention that in Java arrays are strictly typed.
     7. Talk about the length field.
     8. Show the standard for-loop used to process elements of an array sequentially.
     9. Do several examples involving the processing of arrays of primitives.
     10. Show how simple assignment of array variables results in aliasing (not copying).
     11. Show how an array can be copied by writing an appropriate loop.
     12. Show syntax for instantiating an array with initial values (using curly braces).
     13. Explain that once instantiated, the size of an array cannot be modified.
     14. Show how we can “fake” making an array larger (or smaller).
         1. Assume that there is an array of a certain length that already contains data, and we would like to make this array a bit bigger (adding an element to the end).
         2. Create a new array that is just a bit larger than the original.
         3. Write a loop that copies the data from the original array to the new one.
         4. Add a value to the remaining box.
         5. Re-assign the variable that was a reference to the original array so that it is now a reference to the new array.
  5. Arrays of references
     1. Show an example of the syntax for creating a variable that will refer to an array of references.
     2. Show syntax for instantiating the array, and draw a memory diagram showing the array, and indicating that the array is automatically filled with null references.
     3. Show the syntax for instantiating the objects that are to be contained in the collection, and show on the memory diagram (arrows leading from each box to the object).
     4. Do an example or two utilizing arrays of references.
  6. Two dimensional (rectangular) arrays.
     1. Show syntax for declaring an appropriate variable to create a rectangular array of primitive values.
     2. Show syntax for creating the rectangular array (supplying values for both sets of square brackets).
     3. Draw the memory diagram.
        1. Local variable on the stack
        2. Array leading to a vertical array of references.
        3. Each element of the vertical array contains an arrow pointing to a horizontal array (“row”).
        4. Indicate that each box is automatically populated with the default value (0, or whatever).
     4. Show syntax for accessing each box (using two indices and two sets of square brackets).
     5. Show the common idiom (nested loops) used to process each element of the collection.
     6. Do an example that utilizes a 2-Dimensional array of primitives.
     7. Talk about 2-dimensional arrays of references, drawing a memory a diagram at each stage:
        1. Go through syntax for declaring the variable, and for creating the array. Be sure to mention that the array is automatically initialized with null references.
        2. Show how to use the usual nested loops to assign values each box. (Draw arrows leading from each box to a new object on the heap.)
        3. Do an example illustrating the use of a two-dimensional array of references.

# For-each loops

* 1. Show syntax and explain for-each loops
  2. Do examples:
     1. Array of primitives
     2. Array of references
     3. ArrayList
  3. Limitations
     1. Can’t easily traverse just a portion of the collection.
     2. There are no “indices” that are sometimes useful during the traversal.
     3. Adding/Removing an element during the traversal results in a ConcurrentModificationException.

# Mutability

* 1. Define terms “mutable” and “immutable”.
  2. Go over some classes that students have seen during the course (in lecture, during a lab exercise, or project) and talk about which ones are mutable and which are immutable.
  3. Explain that in Java, this concept is relevant in cases where variables are aliased.
     1. If variables are aliased to an immutable object, it is unlikely to cause problems.
     2. If variables are aliased to a mutable object, care must be taken because changes to the state of the object performed via one variable will be seen via the other. This can be confusing or even problematic.
     3. Since aliasing occurs frequently in Java (via assignment of reference variables, passing references as arguments to a method, etc.), we strongly prefer immutable classes whenever possible.
  4. Explain that there is no particular Java construct to enforce (or detect) a class being immutable.
  5. It is important when writing a class to document clearly for the user whether the class is mutable or immutable.
  6. Introduce the StringBuffer class, and talk about String vs. StringBuffer as an example of immutable vs. mutable. Strings are much easier to use in most cases (since aliasing doesn’t cause any trouble), but they are slow in cases where modifications of the string are needed.

# Deep vs. Shallow Copy

* 1. Come with a class that has instance variables that are references.
  2. Draw memory diagrams for deep and shallow copy.
     1. Deep copy should be a completely different copy of the whole thing.
     2. Explain that a shallow copy is just a bit-for-bit copy from the original object to the new one, so references get copied (not the objects they refer to), resulting in aliasing of corresponding instance variables.
  3. Show a coding example that results in a shallow copy (perhaps a copy constructor with simple assignments).
  4. Show a coding example that results in a deep copy (perhaps a copy constructor where the fields are copied individually by invoking their own copy constructors).
  5. Shallow copy is obviously more efficient.
  6. Under what circumstances would a deeper copy be preferred?
     1. In cases where instance variables are references to mutable objects, deep copy is probably desirable so as to avoid aliasing of those mutable members.

# Privacy Leaks

* 1. Do an example of a class with a private field.
  2. Show the memory diagram for a simple getter (that just returns a reference to the field). This results in aliasing of the instance variable and the variable that is assigned the return value from the getter.
     1. In cases where this common object is immutable, there is no problem here.
     2. In cases where the object mutable, we have created a “privacy leak”.
        1. Explain that the user who has invoked the getter now has a direct reference to this private member, and so they can change the state at will, violating the privacy of the member.
        2. Modify the getter to fix the privacy leak by invoking the copy constructor on the member and returning a copy. This is a “defensive copy”.
  3. Do an example that has a private field that is an array of references.
     1. A getter that just returns a reference to the array is automatically a privacy leak, since Java arrays are always mutable.
     2. So we should consider a getter that returns a copy of the array. But should it be a deep or shallow copy?
        1. If the elements of the array are immutable, a shallow copy is sufficient, and is more efficient than a deeper copy.
        2. If the elements of the array are mutable then a deep copy is better, otherwise the caller of the getter can change the state of the elements in the array.

# Basic Recursion

* 1. Review the “iterative” approach that we implement with loops
  2. Describe the “recursive” approach.
     1. If the dataset we’re working on is sufficiently small/simple, we provide an immediate answer.
     2. Otherwise, we take the dataset and break it into smaller pieces.
     3. We ask the method we are writing to provide solutions for these smaller parts, and we use this information to provide an answer for our original dataset.
  3. Do several examples of basic recursion. Some ideas are:
     1. factorial
     2. fibonnacci
     3. exponentiation (as repeated multiplication)

# Java’s Wrapper classes

* 1. Define “wrapper”
     1. Class encapsulates a single private member.
     2. May be used as an “adapter” (replacing the API of the inner member with the preferred API of the wrapper class.)
  2. Mention examples of wrapper classes that students have already seen during the course (in projects, lab exercises, or lecture examples).
  3. Java has wrapper classes corresponding to each primitive type. Double, Float, Long, Integer, Short, Byte, Boolean, Character.
  4. Show them the API for one of these classes (perhaps Integer).
  5. Do an example of “auto-boxing”.
  6. Do an example of “auto-unboxing”.

# Java Interfaces

* 1. Do an example in which there is a method with a parameter or two, using interface(s) as types (but don’t mention what an interface is yet).
     1. Be sure to invoke several methods from the interface(s) within the method you are demonstrating.
     2. Suggest that it may be desirable to have multiple different implementations for these method calls so that the behavior of the method you are demonstrating could vary.
     3. Reveal that the types of the parameters are actually “Java interface(s)”.
     4. Show the code for the interface(s).
     5. Show examples of multiple classes that implement the interface(s).
     6. Talk about polymorphic variables.
        1. What kinds of objects can the variable refer to? (Instances of any class implementing the interface).
        2. What methods can be invoked via this variable? (Only those that are explicitly included in the interface).
     7. Be sure to mention that it is impossible to instantiate the interface itself.
     8. Go back to the original method that you are demonstrating and explicitly state that the parameters are examples of polymorphic variables. The behavior of this method will vary, depending on what kind of arguments are passed.
     9. Be sure to emphasize that the very same statement (invoking a method from the interface) will execute different versions of the method, depending on what class of object the variable is referencing is at the time.
     10. Write an additional method that instantiates different classes (implementing the interface(s)) and use them to call the original method, demonstrating that the behavior of the method can be quite different depending on the class of object that is supplied.
  2. Do additional examples. This concept is difficult for student to grasp.

# Brief Intro to Inheritance

* 1. The goals are to give students an idea about what inheritance is, and to get them to the point where we can write a proper equals method. The details of inheritance are taught in CMSC 132.
  2. Write a short, typical Java class. (E.g.: Clock).
  3. Suggest that we would like to write a class that is a more specific version of the original one. (E.g. AlarmClock).
  4. At first it “copying and pasting” the code from the Clock class into the AlarmClock class may seem like a good way to get started. But explain that we will do something more interesting and potentially more useful.
  5. Add the expression “extends Clock” to the AlarmClock class.
  6. Explain that the Alarm clock class now “inherits” all of the features of a Clock (even though we have not entered any code at all into the AlarmClock class.)
  7. Be sure to mention the terms “extend”, “extension”, “inherit”, “inheritance”.
  8. Prove that is works by writing a main method that instantiates an AlarmClock and invokes some of the inherited methods.
  9. Draw an inheritance diagram illustrating the relationship between Clock and AlarmClock. Add additional Clock classes to the diagram (perhaps WeatherClock, RadioAlarmCock, NatureClock, etc.)
  10. Talk about the notion of “base/super class” and “derived/sub class”.
  11. Mention the informal “IS-A” relationship.
  12. Talk about transitivity. (Since AlarmClock IS-A Clock and RadioAlarmClock IS-A AlarmClock, we can conclude that RadioAlarmClock IS-A Clock.)
  13. Introduce the notion of polymorphism via superclass type.
      1. Declare a variable of type AlarmClock and ask:
         1. What kinds of objects can this variable refer to? (AlarmClock or any of its subclasses).
         2. What kinds of methods can be invoked via this variable (AlarmClock methods, or anything inherited from the Clock class).
  14. Compare/contrast Java interfaces with extentions.
      1. Both techniques are useful because they allow polymorphism.
      2. Implementing and interface requires you to include various methods, but you don’t inherit anything
      3. Extending a class allows you to inherit its features.
  15. Define method overriding. (Replacing an inherited method with a local one).
  16. Be sure to go over the difference between overloading and overriding.
      1. Suppose the Clock class has this method:

public static void tick()

* + 1. Overload the method in the AlarmClock class like this:

public static void tick(int n)

* + 1. Next override the tick method by supplying one in the AlarmClock class with no parameters.
  1. Explain that EVERY class we write is an extension of something. If you don’t explicitly type “extends” then your class extends a class called Object.
  2. Go back to your inheritance diagram and state that the Object class should really be at the root.
  3. Ask: What kinds of objects can be assigned to variable of type “Object”? (Any).
  4. Show the API documentation for class Object and make it clear that these methods are inherited by EVERY class in Java. (In passing, mention that we have been overriding the toString method already).

# Equals method

* 1. Propose the following (incorrect) implementation for a class called Clock:

public boolean equals(Clock c) {

return hours = c.hours && mins == c.mins;

}

* 1. What is wrong? (It’s an overload, not an override of the equals method from the Object class. Therefore the Clock class has two different equals methods available which can lead to problems.)
  2. Try this:

public boolean equals(Object c) {

return hours = c.hours && mins == c.mins;

}

* 1. Explain that this won’t work because hours and mins are not members of the Object class.
  2. We need a variable of type “Clock”. Try this:

public boolean equals(Object x) {

Clock c = x;

return hours = c.hours && mins == c.mins;

}

* 1. Explain that the compiler is not happy about assigning c as an alias for x, because c needs to be assigned a clock, but we don’t know what x is!
  2. Introduce “instanceof” and try the following:

public boolean equals(Object x) {

if ( ! (x instanceof Clock)) {

return false;

}

Clock c = x;

return hours = c.hours && mins == c.mins;

}

* 1. The assignment is still not working. Even though we know that x refers to a Clock at that point, the compiler doesn’t. We need to do an explicit type cast. Introduce this notion briefly, without going into too much detail (it will be covered more completely in CMSC 132). The final (correct) version:

public boolean equals(Object x) {

if ( ! (x instanceof Clock)) {

return false;

}

Clock c = (Clock)x;

return hours = c.hours && mins == c.mins;

}

* 1. Mention that there are other variations. An alternate technique using getClass instead of instanceof. The following optimization is often included:

if (x == this) {

return true;

}

# More Advanced Recursion

* 1. Do several examples requiring a “helper” method (because we need to add another parameter so that we can pass simple references instead of making copies of the dataset). Example:
     1. Original (inefficient) version:

public static boolean isPresent(String s, char c) {

if (s.length == 0) {

return false;

}

if (s.charAt(0) == c) {

return true;

}

return isPresent(s.substring(1, s.length()), c);

}

* + 1. What’s wrong with the above? (Calling substring repeatedly wastes an enormous amount of time and creates an enormous amount of garbage on the heap).
    2. Better version, using a helper method:

public static boolean isPresent(String s, char c) {

return *isPresentHelper*(s, c, 0);

}

private static boolean isPresentHelper(String s, char c, int pos) {

if (pos >= s.length()) {

return false;

}

if (s.charAt(pos) == c) {

return true;

}

return *isPresentHelper*(s, c, pos + 1);

}

* 1. Do some rexamples that are of a “different flavor”. Ideas:
     1. Mergesort
     2. Write code that counts the number of files in a directory (and including all descendent directories)
     3. Write code that solves the Hanoi Towers puzzle for arbitrary numbers of disks. (This is a fun example to code up “live”, with students offering help/suggestions as you go.

# Javadoc

* 1. Explain the purpose of the Javadoc utility.
  2. Do a demonstration by showing a fairly complex example (with several classes/interfaces) that has been commented thoroughly using Javadoc comments.
  3. Be sure to show examples of @return @param, @throws @author, etc.
  4. Invoke the Javadoc utility to create the Javadoc documentation and show it to the students, pointing out things that it has done automatically and places where it created content based on our comments and tags.