-proper abstraction allows us to treat complex ideas as “black boxes” so that we can reason about their behavior without knowing their exact contents

-often objects are abstract, but they usually have some data or information that they “know” and some actions that they can perform when asked (methods), usually involving the data that they hold

-abstract data types (ADTs) are collections of operations that abstract what goes on

-data structures are concrete ways of representing data

-“main” defines the program to run…wrapped up in a class

-a class essentially gives a “blueprint” or a template for an entity in a program, which is called an object

-have to declare variables and say what type each will hold

-only declaring the type of the variable leaves it empty—“new” is required to create an object

-private means that objects from other classes don’t have access

-public keywords says that objects are accessible to anybody

-methods “belong to” the object, and we invoke them on the object

**Class Definition:** Java is organized around classes, which provide a blueprint for how to organize objects in terms of data and methods. The name of the class should match the name of the file.

**Method Definition:** A method performs an operation for an object. It is defined inside the class, and at its heart include a name, list of parameters, and return type, followed by a body, which is a code block to be executed.

**public static void main (String args[]):** A special method name and type signature telling Java where to begin execution of the program—it is the main method.

**Syntax Characters:** Note that Java uses curly braces, to mark blocks of code, bodies of classes, etc. Wherever, we would’ve indented in Python. Each individual is ended with a semicolon.

**Output:** System.out handles the basic output. One method on System.out is println, and it prints a string and then a newline.

**Method Invocation:** to ask “object” to perform “method” with “parameters”, we write object.method(parameters)

**Import:** Lots of additional functionality is provided by extra packages; the “import” statement says to use them and the classes they define

**Variable Declaration:** Every variable has a type. That’s true in any language—it tells the machine how a bunch of 0s and 1s should be interpreted. But in many compiled languages like Java that type must be specified in the source code

**Variable Assignment:** The equals sign is used to assign a value to a variable. That can be done at the same time as the variable is declared, and then again later.

**Object Creation:** An instance of a class, an object, is created by the “new” operator, with the name of the class and any parameters needed to initialize the object.

**Comments:** Multi-line comments start with /\* and end with **/**; single-line comments are marked with //. Comments beginning with /\* are used to automatically generate HTML-based program documentation-javadocs.

**Instance Variables/Fields:** These store data specific to an object; they are declared outside any method.

**Visibility (public/private/protected):** This part of a declaration says who has access: public (any other method has access), private (only the object’s methods), or protected (subclasses can too)

**Constructor:** Creating an object creates its instance variables. Java will initialize these. Java provides a special type of method called a constructor, that has the same name as the class. It is called via “new”, and is responsible for giving all of the instance variables appropriate values. It can take parameters, too.

**Capitalization:** By convention, we name our classes so as to start with capital letters and our variables and methods to start with lowercase letters. We capitalize each word in a multi-word name.

**Conditional:** Java has the usual kinds of conditionals, with the test in parentheses, the “then” in braces, and optionally an “else” (which can be an “else if”)

**Return:** This exits a method immediately, and if the method returns a value, specifies what to pass back. Methods that don’t return values just have “return” all by itself to return early, or just naturally return at the end of the code block.

**Return Type:** If a method is to return a value, the type of that value is specified. If it doesn’t return a value, “void” is indicated as the return type

**Increment**: The increment operator adds 1 to a variable. var++ is the same as var=var+1

**Null:** This indicates a non-existent object

**Drawing GUI**

-initWindow(width,height)—call this to create the GYI window to be the given size

-repaint—call this to refresh the display

-draw(Graphics)—provide a new version of this in order to display what we want

-handleMousePress(x,y)—provide a new version of this to respond to the press at the location

-handleKey Press(key)—provide a new version of this to respond to the character pressed

-“static” tag indicates that a function is not a method of an object, but rather just a standalone function of the class…invoke it without an object and dot before its name

**For Loop:** The basic structure is “for(initialization; conditional; step){statements}.” The initialization is done once before the loop starts.

**Equality Testing:** = = is really only appropriate for numbers and characters

**Primitive Types v. Objects:** Primitive types, including “int” “double” “float” boolean” “char” are stored directly in a variable and are passed by value. Objects are stored as references to the actual data passed that way.

**Type Conversion/Casting:** In some cases we can convert one type to another…(int)1.6

**Variable Scope:** Local variables declared inside a method are only there during that method invocation. In fact, they’re only accessible within their containing curly braces. **Static:** A “static” method or variable “lives” outside of any particular object. A static method does not need any object.

-ArrayList is just a list of objects with methods…

* add(E elmt) - appends element elmt to the end of the list.
* add(int index, E elmt) - inserts element elmt at position index.
* get(int index) - returns the element at position index.
* remove(int index) - removes the element at position index.
* set(int index, E elmt) - sets position index to elmt
* size() - returns the number of elements in the ArrayList.

**Generics:** When a container object doesn’t really care what type of thing it holds, that type can be given as a parameter in angle brackets…ArrayList<BufferedImage>. Then only objects of that type are in the container, and when we get one, we know its type.

**Final:** This tag on a declaration indicates that its value won’t ever be changed after being initially set. This is a good way to establish overall settings of a program in a clear, visible way, without “magic numbers”

-generally considered bad form for a class to let other classes have access to its instance variables, as that means that they are dependent on a particular implementation that might be subject to change

-inheritance is the object-oriented way to create new classes from existing ones. Existing class is called the superclass, new class is called the subclass. Each object of the subclass “inherits” the instance variables and methods of the superclass

-Use inheritance to model “is-a” relationships and only “is-a” relationships

**For Each:** A special type of loop that iterates over members of a collection

**Extends:** This indicates that a class is a subclass of another class, extending its instance variables and methods, and perhaps overriding some of them.

**Super:** A way to directly access a method of the superclass from a method of the subclass. The superclass constructor is invoked automatically from the subclass constructor, but if it needs parameters, then this is a way to provide them. By default, other superclass methods are not automatically invoked from corresponding subclass methods; the whole point of the subclass method is to refine the functionality appropriately. If that refinement entails just doing something in addition, then specifically call the “super”.

-With an ADT, we can perform certain operations supported by an interface, but we can’t directly get at the implementation by which the data are represented and the operations supported

-Inheritance captures similarities among classes, in that we can use a subclass in any context where we can use its superclass

-An interface just specifies a bunch of method headers, but without bodies implementing them…the interface gives the collection of operations supported by the ADT, without specifying the data representation

-an “implements” tag in the subclass definition is a promise to implement all the methods of the interface

-with protected access, the variable or the method should be accessible to only the class and any subclass that extends the class

-package access makes the variable or method visible to everything in the same package. A package is a way of grouping methods together that are intended to work together as a group.

-an exception indicates something happened at run-time that wasn’t part of the standard flow, in that there’s not a clear way to continue from this point

-Exceptions are either caught and handled in the method that throws them or passed onto the method that called it. IF the main program doesn’t catch the exception it kills the program and prints an error message

-The basic idea is to wrap a try/catch around code that might throw an exception. The catch specifies the type of exception it can handle and the code to perform when that exception is thrown. We can also follow the try with a finally that has code to be executed whether the code in the try block succeeds or fails

**Interface:** An interface specifies a set of methods, but no instance variables, and does not provide implementations for those methods. When a class implements an interface, it must provide all the method implementations

**Inner Class:** A class can be defined inside another class.

**Throws:** This annotation on a function indicates that it can pass up an exception to its caller

**Try/Catch/Finally:** Exception-handling machinery is introduced in more detail in the body of the notes. This is the syntax for wrapping up some code that might throw an exception

**System.err:** same idea as System.out except it’s meant for error messages

**toString:** A standard method provided by any object to return a string representation for printing

-represent hierarchical relationships using a data structure called a tree, which is built from nodes. A tree has a left and a right, which could be children or could be null…most tree code is recursive…a bunch of inserts and deletes can lead to an unbalanced tree

**Primitive Wrapping/Boxing:** To use a primitive piece of data where we need an object, we use a wrapper that boxes the data in an object

**-Binary search runs in log(n), sequential search runs in n, many run in n­2, some in 2n when all possible combinations are dealt with**

**Order of Growth**

-log(n)

-n

-n2

-2n

-Theta-N is when a runtime is bounded above and below, O() is only worst case, when bounded above

-constant factors don’t matter, neither do low-order terms

**Stack:** a stack is a LIFO

-push-add to top of stack

-pop-return the top item on the stack and remove

it

-peek-return the top item but don’t remove it

-isEmpty-return true if the stack is empty

-O(1) runtime, but add can take longer if we need more space

**Queue:** queues are FIFOs

-enqueue-add at rear of the queue

-dequeue-remove and return the first item in the

queue

-front-return the first item, but don’t remove it

-isEmpty-return true if the stack is empty

-all operations are O(1)

-Deques are double ended queues that are implemented with doubly-lined lists

**Priority Queue:** a queue which instead of FIFO is Best Out

-low priority numbers are removed first

-isEmpty, a predicate that tells whether the

priority queue is empty

-insert, which inserts an element into the priority

queue

-minimum, which returns the element with the

smallest key

-maximum, which returns the element with the

largest key

-use sorted or unsorted ArrayLists, mostly O(1), some are O(n)

**Heaps**

-the parent is at index (i-1)/2

-the left child is 2\*i+1

-the right child is 2\*i+2

**Sets**

-an unsorted collection of things

-add(E e)

-contains(E e)

-isEmpty()

-remove(E e)

-size()

**Maps**

-associates keys with values

-containsKey(K key)

-containsValue(V Value)

-get(K key)

-isEmpty()

-keySet()

-put(K key, V value)

-remove(K key)

-size()