JAVA LANGUAGE FEATURES

Structure

Classes → subpackages, → packages [import package.subpackage.class]
→ java.lang is automatically included in any Java programs
→ A Java program must contain at least one class, which contains the main method
→ public static void main (String[] args) { code }
→ The compiler turns your code into bytecode, which the machine can execute

Source files: all the files in your Java program

Javadoc Comments → Outputted into a Javadoc format for easy reference

```
/** Description:....
 * Precondition: ...
 * @param : one of the parameters (one per line)
 * @returns : what the method returns
 * @throws : possible exceptions thrown (not required for AP)
 */
```

casting

Primitive (and inbuilt) types

→ Identifier: name for variable, parameter, constant, etc (case sensitive)
→ Convention: variable and methods are in camelCase
→ Reserved words (lowercase) cannot be used as identifiers

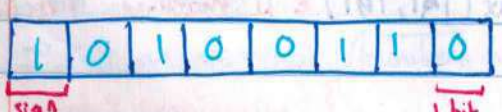
int: an integer, positive or negative
boolean: a "true" or "false" value (1-bit: 0 or 1)
double: a decimal number

char is not required for AP-CSA

→ Casting: converting between types target varName = (target) valueOtherType
→ int → double: happens automatically (no "casting" necessary)
→ double → int: chops off the decimal place
→ cast (double) before an expression to avoid integer division
→ If rounding a negative number, using (int)(doubleNum - 0.5) is always correct

Storing integers

1 byte →



→ Stored in sequences of bits using the binary number system
→ If the number is stored in b bits, the range of the number is $[-(2)^{b-1}, (2)^{b-1})$
→ byte: 8 bits, integer: 32 bits, short: 16 bits, long: 64 bits, boolean: 1 bit

How to get max value: Integer.MAX_VALUE or Integer.MIN_VALUE

Storing floating-point numbers

1 bit | 52 bits | 11 bits → 64 bits for a double (32 for a float)

sign · mantissa · 2 ^ exponent

sign: 1 bit
mantissa: determines digits of number
2: base

-> These numbers can't be represented exactly in binary, which leads to a round-off error
-> Roundoff-errors are compounded by arithmetic operations

Other number systems

-> Hexadecimal: base 16 (digits 0-9, A-F) [prefix 0x in Java]
-> Octal: base 8 (digits 0-7)

Final variable: constant defined by the user

-> By convention, capitalized
-> final double TAX_RATE = 0.08;

Operators (math)

() -> *, /, % -> +, -

-> + (addition), - (subtraction), * (multiplication)
-> division /: integer division returns number without decimal -> 2/5 = 0
-> (double) 2/5 = 0.4, (double) (2/5) = 0.0 because (2/5) evaluated first
-> % (modulo operator) returns remainder 10%3 = 1

Operators (relational) -> Return booleans

-> == (equal to), != not equal to
-> > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to)
-> Because of round-off errors / lack of precision, floating point numbers shouldn't be compared directly with relational operators for equality
-> better way: $|a - b| \leq \epsilon * \max(|a|, |b|)$ ϵ is machine precision 10^{-16} for double

Logical operators (Boolean operators)

-> ! (not), && (and), || (or)
-> Compares two booleans and returns a boolean
-> Short circuit evaluating: the expression is evaluated from L->R, and stops when the value is known
ex. if (num != 0 && num2/num == 1) won't throw an exception

highest
[!, ++, --] -> [*, /, %] -> [+, -] -> [greater, less..] -> [!=, ==] -> [&&] -> [||] -> [+=, --]

Assignment operators

= ⇐ simple assignment

compound assignment operators:
- += ⇐ increment existing value by
- -= ⇐ decrement existing value by
- *= multiply value by
- /= divide value by
- %= modulus value by

assignment operators can be chained: next = prev = sum = 0;

Increment / Decrement operators

[]++ → increments value by 1 (ex. x++) ++[] increments by 1

[]-- → decrements value by 1. (ex. x--) --[] decrements by 1

after the assignment (ex. println) before the assignment (ex. println)

Inputs and Outputs

→ on the AP exam, input is defined as [get user input here]

→ Scanner class can handle inputs and outputs easily (for reference)

→ Output: System.out.println(...) or System.out.print(...)

→ Escape sequence: prints special characters \n → new line, \" → ", \\ → \

Control Structures: how you can make a program run in non-sequential order

→ Decision making: if-else statements, switch statements (based on booleans)

→ Iteration: loops

> break; can be used to break out of any code block (ex. loop, if-statement, etc)

If statements

→ will evaluate a boolean, the run a code block if it is true

```
if (condition) {code}
if (condition) {code} else {code}
if (condition1) {code} else if (condition2) {code} else {code} ...
```

→ Can be nested to create secondary conditions

Switch Statements

→ tests an expression (number, boolean, etc) against a number of cases

```
switch (expression) {
  case a: // code ... break;
  case b: // code ... break;
  default: // code block
```

⚠ NOT REQUIRED FOR AP CSA EXAM

... all the cases

for-loop for (initialize, termination condition, update statement) {code}
- -> termination condition is evaluated before code is run
- -> update statement is performed after code is run
- -> if initialization variable = 0 and update ++ (i < n), the loop runs n times
- -> terminates if condition is false
- -> should be used to run code a predetermined number of times

for-each loop for (someType element : collection) {code}
- -> used for accessing each element in a data structure (list, array, etc)
- -> the name of the accessor variable (element) refers directly to the array element (acts like array[i] in a for loop)
  - -> type in for loop has to match array type

while-loop while (condition) {code}
- -> evaluates as long as the condition evaluates to true
- -> often has an iterator variable that is part of the condition

Exceptions are objects (?)

Errors and exceptions
- -> exception: error that occurs when the program is running (run-time)
- -> unchecked exception: handled automatically by Java
- -> checked exception: handled directly by the programmer
  - -> try {code} catch (Exception exception) finally {code to execute}
    - -> tests for exceptions, controls flow accordingly
  - -> throw new ExceptionName ("optional message");
    - -> throws exception from inside the code
- -> Useful: ArithmeticException (division by 0), NullPointerException, ArrayIndexOutOfBoundsException, IndexOutOfBoundsException, IllegalArgumentExcepti

break; breaks out of the loop, onto the next control structure
continue; skips one iteration of the loop, but continues looping

CLASSES AND OBJECTS

Objects
- -> representations of real world objects
- -> has state (data attributes stored) and behavior
  - -> ex. Book object : title (state) and turnPage() (behavior)
- -> stored in the memory, accessed by an object reference
- -> Objects are instances of classes

Classes
dot syntax?
- -> "Blueprint" (type) for creating objects
- -> state: data field / instance variable, behavior: methods
- -> encapsulation: combining an objects data and functionality into a class

Public, private, static
- -> public class: can be accessed by any client
- -> private class: accessible only to classes in the same package
- -> Clients can't see inside the classes, even public ones
- -> public method / variable: accessible by anyone using the class
- -> private method, variable: accessible only within the class
- -> static variable: class variable that is shared by all instances
  - -> (allocation only happens once)
- -> static final variable: static variable where the value can't be changed

Methods
name
public (access) void (return type) methodName (type1 param1, type2 param2) (parameters) { code; return type; } (return statement)

invoked using objectName.methodName(params)

Method Types
- -> constructor: creates an object of the class (prepares variables)
  - -> ~~ClassName~~ public ClassName(params...) { code }
  - no return type, same name as class
- -> can be multiple constructors
- -> accessor / getter: returns some info about the object without changing it
- -> mutator / setter: changes something about the class
- -> static: refers to the entire class, doesn't need an instance of (object)
  - className.staticMethodName(param);
  - shared between all the classes

as opposed to instance methods

Driver Class
-> Contains main method, usually used to test classes
-> Doesn't have an instance; all methods are static (no objects exist yet)

Method Overloading
-> Methods in the same class with the same name (but different parameters)
-> distinguished by method signature: its name and list of parameter types
  -> return type doesn't matter (not in signature)

Scope
-> Area of the code where a variable can be accessed
-> only accessible inside the pair of braces {} where it's defined
-> if defined outside of any, it is a global / instance variable (accessible everywhere)
-> Local variables take precedence over instance variables with the same name

The this Keyword
-> in a method, refers to the object that it's being called on
  ex. this.variable = value
-> called an implicit parameter

References
-> primitive types (int, boolean, etc..) have "memory slots": they are stored directly in memory. The value of one variable won't affect the other
-> objects are stored in memory; the variables reference them
-> If multiple references are made to one object, changing the object changes it for all the references
-> New objects are created using the new keyword
-> Null reference: reference points to no object
  -> references can be set to null: objectName = null;

Parameters
-> dummy / formal parameter: placeholder name for the real parameter
-> actual parameter: the value actually passed for a given object

a copy of each variable / reference is made and stored in each parameter value

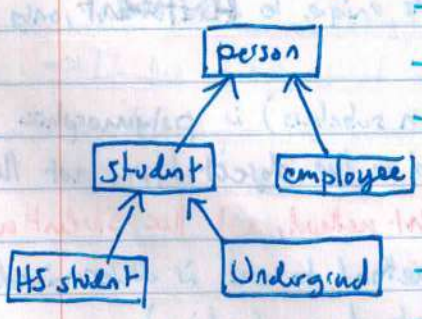
Primitive parameters
-> Value is passed directly into the method
-> scope of values is within that method: they get erased once the method is exited
-> if the parameters are (more) global variables, they will revert back to their original values after the method call.
-> said differently: passing variables as parameters won't change them

Object parameters
-> The object reference is passed directly into the object, and copied and stored.
-> The object stays the same, and if it is changed by the method, it will stay changed after the method call

INHERITANCE AND POLYMORPHISM

Inheritance
-> When a new class is created from an existing class, and absorbs (inherits) the public and protected (but not privated) variables and methods, and adds additional ones
-> protected (access modifier): accessible everywhere except different packages (subclasses of)
not stated (left blank)
-> default (access modifier): accessible in the same package (less permissive)
-> the subclass inherits its characteristics from the superclass
-> Useful for re-using tested code

Inheritance Hierarchy



-> A class can be both a subclass and a superclass
-> Classes can exist in a hierarchy (based on their relationships)
-> ⟶ points from subclass to superclass, designates inheritance, signifies an "is a" relationship
ex. HS is a student (but not student is an HS student)
-> Subclasses can have methods not in the superclass
-> Subclasses can redefine methods that already exist in the superclass (method overriding or partial overriding (if only partially overrides)

a subclass cannot redefine a public method as private
cannot override static methods of a superclass

extends

public SubClass(){
super();
}

-> extends is used to create / implement inheritance (subclass extends superclass)
-> public class SuperClass {code, constructor, etc}
-> public class SubClass extends Superclass {additional code, constructor}
-> the subclass cannot access private variables / methods (must use getter / setter)

super and super();

-> methods from the superclass can be called using super.methodName();
  -> usually used in method overriding (when specifically when something is added)
-> constructors are never inherited!
-> if a subclass has no constructor and...
  -> the superclass has a default (0-parameter) constructor: this is followed
  -> the superclass has no default constructor: compile-time error
  -> all the variables present in the superclass are initialized however, the ones that only exist in the subclass are assigned 0 (primitive) or null (object)
-> a superClass's constructor can be implemented with super(params);
  -> often done inside the subclass constructor, with stuff after
  -> may not include all the variables, so more will come after
  -> must be done on the first line

Declaring subclasses

-> a superclass object can be declared as an object from any of its subclasses
  -> ex. Student nolan = new HSstudent();
  -> won't work the other way (because Student isn't necessarily a HSstudent)
  -> because HSstudent inherits from Student, calling a student method on nolan will work (as will a specific method to HSstudent)
  -> nolan has a Student object reference, but is an HSstudent object and type
    -> nolan cannot use methods unique to HSstudent, only inherited ones

Polymorphism

for method overriding

-> A method defined more than one way (via a subclass) is polymorphic
-> The "correct" method will be called based on the object type, not the reference type (nolan will use the HSstudent method, not the Student method)
-> Dynamic Binding / Late Binding: the method to call is determined during runtime, not in compile time (unlike method overloading)
-> Calling a super method that includes an overridden method will use the method version that applies to the subclass (the overwritten one)

Object class

-> Every class that isn't defined as a subclass is an implicit subclass to Java's Object class

-> Can inherit the default constructor from the Object class

Downcasting

-> Methods unique to subclasses cannot be applied to superclass objects, or subclass objects with superclass references (/ types)

-> However, a superclass object can be downcasted to a subclass object

```
int x = ((GradeXIIStudent) nolan).someMethod();
```

if GradeXIIStudent extends HSStudent and someMethod is unique to GradeXIIStudent

-> parentheses must be used because the dot operator is evaluated before the cast (without them, it would parse `nolan.someMethod();`

class CastException
trying to cast to something other than a subclass

Polymorphic Methods Review

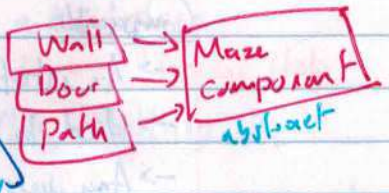
-> Which method to use is chosen at runtime (but params must be correct at compile)

-> For Superclass A = new subclass(); A is always a subClass object, but has a type of Superclass at compile time and subclass at runtime

-> The method chosen (if there is overlap/overriding) is from the subclass

Abstract Classes

should never be instantiated



-> a superclass that represents an abstract concept

-> used simply for categorizing classes in a sensible way

-> can have abstract methods: methods that must be overriden in all the subclasses (so why bother implementing), but are still common to each subclass

-> these methods are just empty headers

no {}, just `method();`

-> can still have normal (concrete) methods that can be applied to the subclasses (and instance variables)

-> If the subclass of an abstract class doesn't implement all the abstract methods, it also must be an abstract class

-> Denoted with the abstract keyword (concrete is default)

Object class

-> Every class that isn't defined as a subclass is an implicit subclass to java's Object class

-> Can inherit the default constructor from the Object class

Downcasting

-> Methods unique to subclasses cannot be applied to superclass objects, or subclass objects with superclass references (/types)

-> However, a superclass object can be downcasted to a subclass object

```
int x = ((GradeXIIStudent) nolan).someMethod();
```

if GradeXIIStudent extends HSStudent and someMethod is unique to GradeXIIStudent

-> parentheses must be used because the dot operator is evaluated before the cast (without them, it would parse `nolan.someMethod();`

class CastException
trying to cast
something oth
than a
subclass

Polymorphic Methods Review

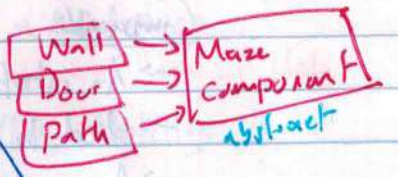
-> Which method to use is chosen at runtime (but params must be correct at compile)

-> For Superclass A = new subclass(); A is always a subClass object, but has a type of Superclass at compile time and subclass at runtime

-> The method chosen (if there is overlap/overriding) is from the subclass

Abstract Classes

should never be instantiated



-> a superclass that represents an abstract concept

-> used simply for categorizing classes in a sensible way

-> can have abstract methods: methods that must be overriden in all the subclasses (so why bother implementing), but are still common to each subclass

no {}, just method();

-> these methods are just empty headers

-> can still have normal (concrete) methods that can be applied to the subclasses (and instance variables)

-> If the subclass of an abstract class doesn't implement all the abstract methods, it also must be an abstract class

-> Denoted with the abstract keyword (concrete is default)

Interfaces

-> Collection of related methods that many disparate classes may want to implement
-> These can be either abstract or concrete (but are by default public)
-> Goal: have compatibility between these classes (without inheritance)
-> A class that implements an interface must implement all the abstract methods in the interface (or be declared as an abstract class)

Declaring and Implementing Interfaces

-> Interface is declared like a class: public interface FlyingObject
-> Like an abstract class, only method headers are included for abstract methods
-> If a class is implementing an interface, the implements keyword is used
  -> public class Bird implements FlyingObject
-> A class can implement many interfaces
-> If a superclass implements an interface, the subclass does too by inheritance
-> Interfaces cannot have instances, but can be object references / types
  ⊗ FlyingObject robin = new FlyingObject();
  ✓ FlyingObject robin = new Bird();

Comparable

-> An interface included in the Java.lang package
-> Used to compare the value of two objects (-1 -> <, 0 -> ==, 1 -> >)
-> Any class that implements this may use the compareTo method
  -> compares implicit object with parameter object
-> If the objects being compared are not type-compatible, a class Cast Exception is thrown

Interfaces vs. Abstract Classes

-> Abstract classes while can have instance methods (interfaces can't)
-> Interfaces can be used across many contexts, abstract classes are only really conducive to the problem (inheritance structure) at hand
-> Interfaces don't require inheritance to use (abstract classes do)

* Java doesn't allow inheritance from multiple classes

SOME STANDARD CLASSES

Object

- -> Universal superclass (all objects inherit from it)
- -> Methods in Object apply to all objects
  - -> toString() returns string representation of an object (often overriden)
    - -> defaults to the address in memory
  - -> equals() determines if objects are the same
    - -> defaults to testing whether two references point to the same address in memory
    - -> often overriden to test if two objects share the same contents, not address
- -> hashCode() returns an integer representation of the object

optional

String

- -> sequences of characters (arrays of chars)
- -> string literal: "example"] instance of the string class — quotes are not part of the object
- -> Immutable: once a String object has been created, it cannot be changed
  - -> You can only create a new String (but it can have the same name)
- -> However, Strings can be initialized like primitives: String test = "oh no";
  - -> Equivalent to String test = new String("oh no")
- -> Concatenation operator + combines two strings into a third string
  - -> String result = "aaa lol" + "Help"; -> result == "aaa lol Help"
  - -> If one of the objects being concatenated isn't a String, toString is called on it to convert it
  - -> If one is a primitive, the string equivalent of it is used
  - -> An error occurs if neither thing is a string
- -> Strings are compared using equals() if (string1.equals(string2)) — overriden to compare values
- -> Can also be compared using compareTo: string1.compareTo(string2)
  - -> If this value < 0, string1 comes before string2
  - -> Alphabetization order: numbers -> capital letters -> lowercase letters
  - -> If the characters are equal, the shorter string is first
- -> length() returns the length of the string
- -> substring(index) and substring(index1, index2) return substrings — the first char is at index 0
  - -> index 1 is the first character you want, index 2 is the first one you don't want
- -> indexOf(string) returns the index of the first occurrence of string in the string it's called on

Interfaces

-> Collection of related methods that many disparate classes may want to implement
-> These can be either abstract or concrete (but are by default public)
-> Goal: have compatibility between these classes (without inheritance)
-> A class that implements an interface must implement all the abstract methods in the interface (or be declared as an abstract class)

Declaring and Implementing Interfaces

-> Interface is declared like a class: public interface FlyingObject
-> Like an abstract class, only method headers are included for abstract methods
-> If a class is implementing an interface, the implements keyword is used
  -> public class Bird implements FlyingObject
-> A class can implement many interfaces
-> If a superclass implements an interface, the subclass does too by inheritance
-> Interfaces cannot have instances, but can be object references / types
  ⊗ FlyingObject robin = new FlyingObject();
  ✓ FlyingObject robin = new Bird();

Comparable

-> An interface included in the java.lang package
-> Used to compare the value of two objects (-1 -> <, 0 -> ==, 1 -> >
-> Any class that implements this may use the compareTo method
  -> compares implicit object with parameter object
-> If the objects being compared are not type-compatible, a class Cast Exception is thrown

Interfaces vs. Abstract Classes

-> Abstract classes can have instance methods (interfaces can't)
-> Interfaces can be used across many contexts, abstract classes are only really conducive to the problem (inheritance structure) at hand
-> Interfaces don't require inheritance to use (abstract classes do)

* Java doesn't allow inheritance from multiple classes

Wrapper Classes

-> An object version (class) of a primitive type
-> Used if a method, data structure, etc requires an object
-> Java has one for all of its primitives
-> Ex. Integer, Double, Character, etc..
-> Methods (Integer): Integer(int value) (constructor), compareTo(Integer other) (returns int), intValue() (returns int), equals(), toString()

same for double/Double

Math

-> static class -> no "Math" objects, functions are called using Math.method()
-> abs(int/double/etc) -> returns absolute value of number
-> pow(base, exp) -> returns the value of base^exp
-> sqrt(number) -> returns the square root of a number
-> random() -> returns random double between 0.0 and 1.0

PROGRAM DESIGN AND ANALYSIS

Waterfall Model

-> Software development process
1. Analysis of Specification: Understand the requirements of the program
2. Program Design: Detailed Plan on how to create the problem
3. Implementation: Code is written
4. Testing and Debugging: All types of input values should be tested
   -> Test data: inputs that test the limits of the programs
   -> Errors: Compile time, Run time, intent / logic error
   -> Assume your users are less smart than you (robustness)
5. Program Maintenance: Updates as circumstances change

OOP Model

Dominant since mid-90s

sets of related classes
List of b.O. | Edits b.O. | Displays B.O

1. Identify classes to be written (basic Object, Collection, Controller, Display)
2. Identify methods (behaviors) for each class
3. Determine relationships between classes (inheritance, composition ("has-a" relationship) <- nested class)
4. Write interface methods for each class
5. Implement the methods

Implementing Classes

-> For a given class, its methods need other classes (collaborators) to be implemented
  -> A class without collaborators is independent
-> Bottom up design: independant classes (usually basic items) are implemented first (possibly by diffent programmers), built on by other classes
  -> Detail can be added later
-> Top-down design: highest-level controlling class is implemented first, lower ones follow (split off from the "main" class)

Implementing Methods

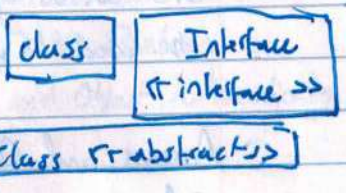
top-down development

-> Helper method: a method that replaces repeated code (help readability)
-> Example of procedural abstraction
-> Stepwise refinement: breaking a complicated method into simpler method(s)
-> Information hiding: using public and private to control what info is visible
-> Stub method: stand-in for a method before it is implemented
-> Algorith: step-by-step procedure to solve a problem

UML Diagrams

-> Keeps track of relationships between classes

——> Association
- - - - - -> Dependency
——▷ Inheritance
- - - - -▷ Implementation
◆—— Composition



Program Analysis

-> Just because a program works under some conditions doesn't mean it always will
  -> A program isn't always correct
-> Assertion: a specific statement about a program at a given point
-> Precondition: something that must be true before the code runs
-> Postcondition: something that must be true after the code runs

Efficiency

-> Efficient algorithms shouldn't use excessive CPU time or memory
-> Algorithms are less efficient if they have unnecessary statements / computations
-> All algorithms have a best, worst, and average case efficiency

ARRAYS AND ARRAYLISTS

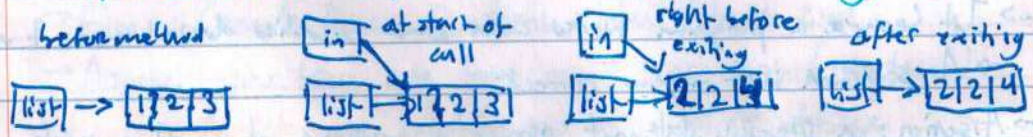
Arrays
- → Object that stores a list of the same type (primitive or object)
- → Index starts at 0, ends at #items − 1
- → Is an object; must be initialized with new
  - → int[] data = new int[n] or int data[] = new int[25]
  - → double[] myArr; → myArr = new double[n]
- → The size of an array is fixed once it has been created
- → When arrays are declared, the values default to 0 or null

Accessing and Editing Arrays
- → An element can be changed with myArr[n] = value;
- → Can be declared with initializer list int[] nums = {1, 1, 2, 3, 5, 8};
- → An array object has a final instance variable length (number of elements)
  - → myArr.length (not a method)

primitives
- → If you want to go through each element in an array without removing or editing elements, use a for-each loop
- → In other cases, use a for loop

- → For an array of objects, a for-each loop and a mutator method can be used to modify objects

Arrays and Objects
- → Arrays are objects ∴ they are passed as references, not by value
  - → If the method it is passed into changes it, the array stays changed



- → Arrays of both primitives and objects can be used as instance variables

Analyzing Array Algorithms
- → Higher efficiency = less operations performed in total
- → An algorithm may have a best and worst case efficiency

ArrayList

ArrayList: A dynamic array ("better" array to store values)
-> Shrinks and grows as needed in the memory (array is constant)
-> Size accessed via a method: myArrList.size();
-> Elements can be inserted and deleted in a single statement, instead of having to create a new object every time
-> Can be nicely printed using System.out.println(myArrList);

Collection<E> is an interface ↑

The Collections Library / API
-> Package by Java that provides a number of useful data structures
-> Designed to be efficient, have methods for insertion and deletion, provide ways to iterate over the whole collection
-> Has a number of interfaces (Arraylist implements List interface)
-> Collection classes are generic: they have type parameters (indicated <type>)
  -> ArrayList<E> instance contains objects of type E (declared with it)
-> The type information is examined at compile time, then erased
  -> This property of generic classes is called erasure

Auto-Boxing and -Unboxing
-> Generics need to contain objects, but programmers might want to use primitives
-> Solution: put numbers in wrapper classes (ex. Integer, Double, Boolean)
-> Auto-Boxing / Unboxing: Converting between the primitive and Object automatically
  -> An accessor method is not needed

The List<E> Interface
-> An interface for data structures that Allows you to
  -> Access an element via an Integer index
  -> Insert an element in the list
  -> Iterate over all the elements in the List using Iterator
-> ArrayList Implements the List<E> interface
-> boolean add(E obj): adds element to end of list, always returns true
-> int size()
-> E get(int index)
-> void add(int index, E element)
-> E remove(int index): returns the element that was removed
-> Iterator<E> iterator(): returns an iterator over every element in the list

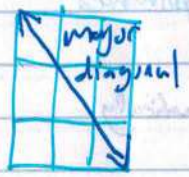
The ArrayList<E> Class

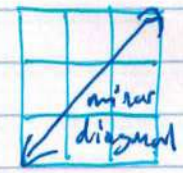
-> Implementation of the List<E> interface
-> Can be resized at runtime (unlike arrays, which are fixed)
-> Operations to remove, insert etc. are optimized, but an array is always more efficient if it can be used
-> Data type: dynamic array
-> Constructor: ArrayList() creates an empty list
  -> ex. ArrayList<Integer> myArrList = new ArrayList<Integer>();

Collections and Iterators

-> Purpose is to traverse a collection one element at a time
-> Iterator: is a generic interface Iterator<E> implemented by ArrayList, etc.
  -> boolean hasNext(); returns true if there is another element to be examined
  -> E next(); returns the next element
  -> void remove(); removes the last element returned by next();
    -> can only be used once per next() call
-> Iterator element type must match the collection element type

Two-Dimensional Arrays





-> Data structure that is matrix-like (grid with rows and columns)
-> In reality: an array of arrays of (type)
  -> declared type [rows][columns] (normal array declaration)
  -> An initializer list can be used with nested {}
  -> Value can be accessed with arr[row][col], (arr: array)
  -> Often traversed with a nested for or for each loop (to access each element)
  -> Can be traversed row-by-row with a single loop

RECURSION

Recursive Methods

-> Recursive methods: methods that reference themselves
-> Multiple methods (calls) may be "pending" (you need to be able to trace them)
-> A recursive method has two pieces
  -> base case: condition that causes the method
  -> recursive case: a non-base case that moves the method towards completion
    -> contains the recursive method call
-> The base case is often the simplest form of a problem
-> Tail recursion: recursive method that has no pending statements after the recursive call
-> Recursive algorithm: repeats a simpler case of itself -> simpler code
-> Iterative algorithm: repeats a process until a condition -> often less memory used
-> Any algorithm can be written recursively or iteratively

General Rules

-> Avoid algorithms that create very large arrays: could cause memory overflow
-> Use recursion when it simplifies code significantly
-> Avoid using when simple iterative solutions exist (ex. factorial, fibonacci, etc.)
-> Recursion is generally useful for
  -> Branching processes, like searching trees
  -> Divide-and-conquer algorithms like merge sort and binary search

Recursive Helper methods

-> Common technique: have a public driver method that calls a private recursive helper method
  -> Useful for hiding the recursive implementation from the client
  -> Make the program more efficient by reducing the number of times a condition must be checked
    -> If value > 0 -> run recursion (instead of checking at each call)

Recursion with 2D grids -> used to traverse a 2-D grid

check that starting point is in range (base case)
If (some requirement)
  Perform action
  RecursiveCall(row+1, col), RecursiveCall(row-1, col), ... (4 calls)

SORTING AND SEARCHING

$O(n^2)$ Selection Sort

→ Search and swap algorithm that passes through a list many times
① Finds smallest element in unsorted part of list (the whole array)
② Swap it with list[0]
③ Find smallest element in unsorted list
④ Swap it with list[1]
... Find the smallest element in the unsorted array, add it to the sorted one
→ Guaranteed to be sorted after n-1 passes if the array has n elements
→ First k elements are sorted after the k^{th} pass

$O(n^2)$ Insertion Sort

→ Has a sorted and unsorted list, and moves elements from one to the other
① Divides list into sorted array (list[0]) and unsorted array (list[1] - list[n-1])
② "Selects" list[1] and places it in the correct position in the sorted array
→ Must iterate through the sorted list
③ "Selects" list[2] and places it in the correct position in the sorted array
... "Selects" the next element and places it correctly in the sorted array
→ Guaranteed to be sorted after n-1 passes if the array has n elements
→ After the k^{th} pass, the first k elements are sorted relative to each other
→ Worst case: sorted in reverse order, best case: sorted in proper order

$O(n \log n)$ Merge Sort

→ Recursive, divide-and-conquer sorting approach
Ⓡ If there is more than one element in the array
① Break the array into two halves
Ⓡ1 Merge sort the left half Ⓡ2 Merge sort the right half
② Merge the two sorted sub-arrays into a sorted array
→ Disadvantage: must use a temporary array as large as the one being sorted
→ Order of the elements doesn't matter; runtime is similar for best, average, worst

* Sorting algorithms are often structured as classes, where the array is a private instance variable, and there is a sort(arr) method

Quicksort

-> Recursive, divide-and-conquer sorting approach

Best O(n log n)
Worst O(n²)

-> Fastest known sorting algorithm for large n

(R) If there are at least two elements in the array
(1) Partition the array (choose a pivot element and organize the list so that everything to the left is larger, and everything to the right is smaller
(R1) Quicksort the left array (R2) Quicksort the right array

-> Partitioning Method (1)

(1a) Pick a random (usually first) item to be partitioned
(1b) Scan from the left side of the array until a greater entry is found
(1c) Scan from the right side of the array until a lesser entry is found
(1d) Swap these (they are out of place)
... Keep scanning and swapping until the scans meet

-> Worst case: one of the arrays is always empty (becomes shitty insertion sort)
Often, this is when the smallest or biggest element is the pivot

Sequential Search

-> Linear Search algorithm
-> Compare the key to each element in the list in order

average: n/2 comps
-> Best case: key matches the first element in the array
-> Worst case: the key matches the last element, or isn't in the list

O(log n)

Binary Search

-> Divide-and-conquer approach that works on sorted arrays
(1) Set two "markers" as the first and last elements of the array
do this until mid = key or high = low
(2) If the key is in between these markers, check if it is in between the midpoint and the first marker (or if it is the midpoint)
(3) If it is, these become the new markers. If not, the midpoint and last element become the new markers
... Keep evaluating and moving markers

# of comparisons: 2^k (k = # of elements to test) → k+1

-> Best case; the key is the midpoint of the array
-> Worst case; the key is not at the end of any subset, or not in the array