Inheritance (Chapter 8)

**Access Modifiers – how they impact attributes / methods, Identify which attributes can be accessed by a derived class**

* Members of derived classes have access to public members of class but not private members

Protected – provides access to derived classes and other classes in the same package but nowhere else.

No specifier – assumes protected. Known as package private

**Articulate how overridden methods work and how the super keyword works, Override vs Overload**

Overrides – member methods in derived class have same name, parameters as base class

Overloading – methods have same name, different parameters.

* A derived member method takes precedence

Super keyword – super.printItem() – this method call allows access of a base class

**“Is-A” vs “Has-A”**

Superclass (more general) in comparison to subclass (specific)

Ex. Butterfly is an insect -> Butterfly = subclass, insect = superclass

Thus, subclass is a superclass.

**“Instance of” keyword, final keyword**

* Checks an object for class membership

Ex. If (account instanceof object) { System.out.print(“account in an object”); }

Final keyword – makes it so that a method can’t be overridden

Ex. private final String(defaultMonsterName) {

How constructors are invoked along inheritance hierarchy (\*)

**Polymorphism**

Compile-time polymorphism -> compiler determines which method to call based on methods arguments

Runtime Polymorphism\* - compiler can’t make that determination, does so instead while program is running

* Ex. Derived classes
  + Say you have genericItem and produceItem where produce extends generic
  + Will dynamically call correct print function based on objects type
  + ABSTRACT CLASSES use this

**Anonymous classes (\*)**

Anonymous inner class – a class inside another class that has no name

* Must implement an interface or extend another class

**Abstract classes vs Interfaces (\*)**

Abstract class – guides design of subclasses but can’t itself be instantiated as an object (SUPER)

* Concrete classes IMPLEMENT Abstract classes (SUB)
* Ex.) A shape class that specifies that ANY subclass must define a method named computeArea()
* Ex. 2) biology ->
  + Upper levels of hierarchy specify features in common across all members below that level of hierarchy, but, the upper levels can’t actually be instantiated. Only the lowest level, species, can. This makes sense!
* Provides variables/fields to subclasses, interfaces don’t.

Interface – can specify a set of abstract methods than an implementing class must define

* Ex. public interface Drawable {
  + Notice: no abstract keyword necessary
  + Can use “implement” keyword -> Square implements Drawable, Drawable ASCII {
  + Classes can implement multiple interfaces using a comma separated list
* Declare public static final fields, doesn’t restrict future inheritance

**Object Class**

* Ex. toString() -> returns string representation of object
* Ex. equals(otherObject) -> Boolean, returns true if object equals otherObject

Searching and Sorting Algorithms (Chapter 9)

**Selection sort, bubble sort, and insertion sort**

*Selection* – treats input as two parts, sorted and unsorted

* O(N^2) runtime

For (i = 0; I < numbers.length – 1; ++i) {

IndexSmallest = i;

For (j = i + 1; j < numbers.length; ++j) {

If (numbers[j] < numbers[indexSmallest]) {

indexSmallest = j;

}

}

}

Temp = numbers[i]

Numbers[i] = numbers[indexSmallest];

Numbers[indexSmallest] = temp;

}

*Insertion* – again, two parts. Repeatedly inserts next value from unsorted part into the correct sorted part location.

* O(N^2) runtime
* Explanation: if N elements, outer loops N-1 times. Inner loop executes N/2 times on average. (N-1) \* (N/2) = O(N^2)
* For sorted or nearly sorted lists, insertion has a O(N) runtime

For (i = 1; i < numbers.length; ++i) {

j = i;

while ( j > 0 && numbers[ j ] < numbers[ j – 1 ]) {

temp = numbers[ j ];

numbers[ j ] = numbers[ j – 1];

numbers[j – 1] = temp;

--j;

}

}

*Bubble Sort* – compares adjacent numbers and flips them if necessary

**Sequential search vs binary search**

*Sequential* – i.e. linear, starts at beginning of list and searches for element until found

* O(N) complexity

Code (its easy!)

For ( I = 0, I < numbers.length; ++i) {

If (numbers[i] == key) {

Return I;

}

Return -1;

}

Binary – checks middle of list first and goes from there

* O(logN) complexity
* Max steps to find number: log(N) + 1

Code (more difficult but still easy)

Low = 0;

High = numbers.length-1;

While (high >= low) {

Mid = (high + low) / 2;

If (numbers[mid] < key) {

Low = mid + 1;

}

Else if (numbers[mid] > key) {

High = mid – 1;

}

Else {

Return mid;

}

}

**Big-O notation**

* Mathematical way of describing how a function’s running time behaves in relation to input size
* Higher order kept, other terms discarded
* Constants are dust

Recursion (Chapter 10)

Base case, recursive step

* Base case is what describes how to actually do something
* Recursive case calls itself

Tail recursion