Introduction

It is important to understand, to internalize, and to be able to use various definitions, principles, and concepts of

1. Java and

2. Data structures and algorithmic complexity

With regard to Java, we can ask a lot of different questions. It is best that you not memorize the answers but be able to figure them out based on definitions, principles, and concepts. Only then have you really mastered Java and programming in it.

We are continually amazed at the number of people who, at the end of the semester, cannot tell us how the new-expression is evaluated. "What do you mean, evaluate it?" they ask in a puzzled manner. If you really know Java, you can explain in a simple, crisp manner to anyone how the assignment statement, the method call, and the try-statement are executed and how the new-expression and the conditional expression are evaluated. We expect you to know these things.

With regard to data structures. You should master recursion and various forms of linked lists, trees (e.g. binary trees, BSTs, heaps), and graphs. You should be able to develop algorithms from their specifications —e.g. searching/sorting depth-first search, breadth-first search. You should master algorithmic complexity and be able to examine a method/program and determine what its worst-case algorithmic complexity is.

Below, we give you *some* of the definitions, principles, and concepts in Java that you should master. A later document will deal with data structures.

Practice developing certain algorithms from their pre-and-postconditions.
Start with a blank piece of paper and develop. When done, compare to
slides listed below. If difference, see what the difference is.
Next night, do the same thing.
This practice not only helps you learn the algorithm, it instills in you a way of thinking about the development of loops.

(a) binary search. WATCH the ppt slides in lecture 10, slides 6..7
(b) insertion sort. WATCH the ppt slides in lecture 10, slides 9..15
(c) selection sort. WATCH the ppt slides in lecture 10, slides 16..18
(d) partition algorithm of .. WATCH the ppt slides in lecture 10, slides 19..20

Suggestions for studying

1. Writing definitions and concepts on a piece of paper.

Suppose you want to learn some definition, like how the new-expression is evaluated, or how a method call is executed, or what the inside-out rule is, or what is meant by f(n) is O(g(n)).

One evening, write it down, copying from our lecture slides or elsewhere if you don't know it. It is not enough just to read. It's *doing* that is important. Copy it twice.

Next evening, try writing it down on a blank sheet of paper and compare what you wrote with our definition. If you had trouble writing it down, or what you wrote wasn't exactly right, copy down the correct definition again, focusing on what you got wrong.

Next evening, do the same thing.

As you do this, think about what it means. If it something like the new-expression, evaluate some new expression yourself for some simple class you know about, following the 3 steps. Think of ramifications of the definition —here are examples:

For executing a method call, what does the definition tell you about when space for a local variable is allocated? Consider a new-expression; what value is in a field of the new object if the constructor does not assign it a value?

Consider the inside-out rule: be able to describe a realistic example of two declarations of a name of a variable and the inside-out rule does not allow access to one of them using just the name.

Spending 5 minutes in each of 2-3 evenings is a relatively painless way to internalize something. Especially if you start now, rather than wait until 2 days before the final and then cram. And, learning the definitions, principles, concepts given below will help make you a better programmer.

2. Hand-execute program segments, drawing objects as they are created, drawing frames for method calls, etc.

Yes, you as well as the computer can execute statements.

Sometimes, you can best find an error in your program by hand-executing a small sequence of code. You will also find requests to execute sequences of code on exams. So get used to doing it.

But you *have* to do it right. For example, to execute the statement x= new C(5); you *have* to draw the new object (putting in it only the fields and methods that are needed to find an error or answer a question), execute the constructor call, write down the value of the new-expression, and store the value in x. If you don't, the chances of getting it right are slim.

3. Use diagrams/pictures. Visualization helps.

Example: the meaning of "f(n) is O(g(n))" is best learned by visualizing a picture of it (see our slides).

Example: to remember what classes an object can be cast to, draw an object, as we draw them, say something with partitions for Object, Animal, Cat, Siamese, and look at it as you try to remember.

Example: We draw many preconditions, postconditions, and loop invariants for algorithms that manipulate arrays.

Java Definitions, principles, concepts

Below are *some* of the Java items you should be completely fluent with.

1. Definition of type. Lecture 1, slide 20.

2. How we draw objects of a class. Lecture 2, slide 11..12; Lecture 44, slide 7.

3. Java wrapper class. Piazza supplementary material note @248.

4. Overloading method names (declaring two method with the same name but different parameters), overriding methods (Lecture 4, slide 10)

5. Constructor: Slides for lecture 5.
   - *Main purpose*: Initialize fields of a newly constructed object to make the class invariant true.
   - *Principle*: Initialize fields of superclass before fields of subclass
   - The first statement of any constructor body you write must be a call on another constructor --**this**(...) or **super**(...). If not, Java inserts **super**().
   - If a class C does not have a constructor declared in it, Java declares this one: **public** C(){}

6. What you should know about executing/evaluating:
   - The new-expression: What is the syntax of a new-expression? What are the three steps in evaluating a new-expression? (See slide 21 of lecture 3)
   - In an expression **this**.x or **this**.m(...), what does "**this**" evaluate to? (Lecture 5, Slide 12)
   - What are the four steps in executing a method call? (Lecture 7, Slides 11..22, esp. 22)
   - What's the syntax of the try-statement? How is a try-statement executed? (Recitation 3, slide 20)

7. **this**. and **super**. (Lecture 5.)
   - In an expression **this**.x or **this**.m(...), what does "**this**" evaluate to? Slide 12
   - In an expression **super**.x or **super**.m(...), what does "**super**" evaluate to? Slide 13
   - When is this needed "**this**." Only when a variable is shadowed. If that is not the case avoid useless clutter and don't put in "**this**.". Slide 12.8.

8. Four kinds of variable: field (instance variable), static variable (field), parameter, local variable. Scope of a local variable Lecture 5, slide 5.

9. What is a class invariant? Code style guidelines, section 3.3. slide 10 of lecture 3.

10. Inside-out rule. Lecture 5, slide 9-11.

11. Exception handling (Recitation 3)
    - Throwable: the superclass of all throwable objects. Slide 15.
    - Subclasses Exception and Error
    - What a subclass declaration should look like. Slide 24.
    - Execution of a try-statement —you should know the steps. Slide 20.
    - The throw statement: Slide 21.

12. Abstract classes and methods (Slides for recitation 4)
    - Why make a class abstract?
    - Why make a method abstract?

13. Interface. What is the syntax for an interface declaration? How can an interface be used to define a type (at least its syntax, with comments giving meaning). People call these kinds of types ADTs —for Abstract Data Type.

14. Generics. Why can't ArrayList<String> be a subclass (subtype) of ArrayList<Object>?