# CSC 461 Quiz #1

**Study Guide**

Mary Moore

---

---

Basic Java Code
Exceptions
Class v.s. Instance
Access Levels
Methods to Critique Languages
Basic Categories of Programming Languages
Basic Components that Influence Acceptance and Learnability
SOLID
Visitor Concept
Basic OOP Diagraming
Syntax Concepts

---

---

## Basic Java Code

- Java overview
    - generation: high level
    - paradigm: imperative/OOP
    - location: everywhere
    - primary applications:
        - internet/web programming
        - graphics
        - GUI programming
        - concurrency (threads)
        - globalization/internalization
        - many useful libraries
        - embedded systems & mobile devices
    - philosophy:
        - simpler than C++
        - portability
            - "build once, run anywhere"
            - not actually true, but better than C++
        - GUI focused
        - C-like syntax
        - modularity
            - "pure" OOP language
            - everything is part of a class
        - reliability
            - tries to forbid constructs w/high rate of misuse
- forbidden construct "fixes"
    - parameter passing is only by value
    - no default param values
    - no operator overloading
    - no structs or unions
    - no typedefs
    - no explicit deletion of heap-dynamic obj (auto garbage collection)
    - no multiple inheritance
    - much smaller primitive set
- strengths
    - built in GUI support
    - enforces structures of ur files
    - A LOT of built in libraries
- weaknesses
    - console I/O is a pain (b/c originally meant for GUI)
    - slower than compiled languages
    - can reverse compilation (so there's security issues)

- garbage collection
    - the language handles cleaning dynamic memory allocations 🙂
    - tends to hit at the worst time 🙁
    - pros:
        - lessens memory leaks
        - simplifies the code
        - automatic cleanup when closing
    - cons:
        - accidental deletion (early)
        - loss of control
        - slower
- 10 universal structures
    - commenting
    - output
    - input
    - selection
    - repetition
    - assignment
    - function
    - exception
    - I/O handling
    - classes
- other common structures
    - arrays
    - strings & formatting
    - enums
    - primitives for the language

**Questions:**

1. **Write a for loop in Java that outputs even numbers from 0 to 10, inclusive.**
   for (int i = 0; i <=10; i++)
   if (i % 2 == 0)
   System.out.println(i);

2. **How do you write a constant in Java?**
   final static datatype name = x;

---

## Exceptions

- an exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions
    - exceptions can be caught and handled by the program
    - when an exception occurs within a method, it creates an object
    - this object is called the exception object

        - contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred
- major reasons why an exception occurs
    - invalid user input
    - device failure
    - loss of network connection
    - physical limitations (out-of-disk memory)
    - code errors
    - opening an unavailable file
- error: indicates a serious problem that a reasonable application should not try to catch
- exception: indicates conditions that a reasonable application might try to catch
- all exceptions and errors are subclasses of the Throwable class in Java
- exception keywords in Java:
    - try
    - catch
    - throw → manually throw an exception
    - throws
    - finally → code to be executed after the try catch ends
- order exceptions from most to least specific

---

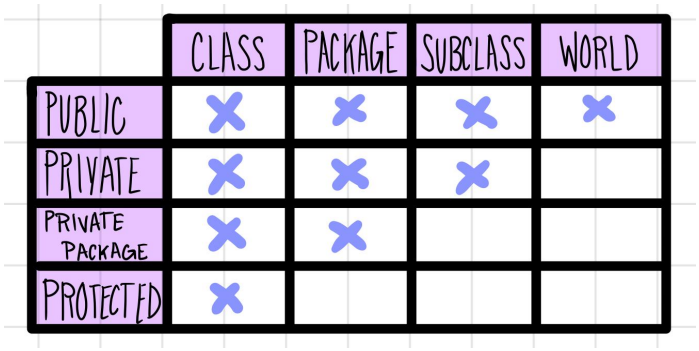
## Class v.s. Instance

- class method:
    - static, no instantiation needed
- instance method:
    - need data
- when methods reference non-static member variables, we must define them as instance methods
- we sometimes define a method that doesn't reference member variables or only references *static* variables
    - when we do this, we can make the method a *static* method
    - this means that we don't need an instance of the class to invoke the method
- it's important to remember that while *static* methods may seem like a good choice, they can be difficult to unit test since there's no object to mock
    - static methods can introduce concurrency issues if the method operates on a *static* member variable…

---

## Access Levels

- package availability is the default access level (just Java I think…)




Access Modifiers In Java

| Access Modifier | Within the Class | Other Classes [Within the Package] | In Subclasses [Within the package and other packages] | Any Class [In Other Packages] |
|---|---|---|---|---|
| public | Y | Y | Y | Y |
| protected | Y | Y | Y | N |
| default | Y | Y | Same Package – Y Other Packages - N | N |
| private | Y | N | N | N |

Y – Accessible
N – Not Accessible

---

# Methods to Critique Languages

- what is a programming language?
  - can make/perform an algorithm
  - sequence of steps
  - selection (if)
  - repetition (loop)
  - can make & use some data format
- "partial languages"
  - HTML, XML, CSS, markup languages ⇒ don't meat language criteria

## Basic Categories of Programming Languages

- programming languages "category" interpretation
  - there are many ways to group languages
  - 3 big ares:
    - generation
    - paradigm
    - application/location
- generation
  - programming languages "grow up"
  - machine ⇒ assembly ⇒ high level (C/C++/C#) ⇒ "higher" level (SQL, close to spoken language) ⇒ "higher yet" level (prolog, specialty)
- paradigms
  - procedural
    - no functions (parse), run till completion
    - functions used, but could be one giant piece of code
    - ex. assembly & some forms of C
  - imperative
    - "normal"
    - states how to get things
    - ex. C++, Java, Python
  - functional
    - bunch of inputs, one output

        - state “what” you want
        - ex. Scala, R, LISP, Haskell
    - OOP
        - pair data & algorithms
    - logic
        - if, →, then
        - ex. prolog
- location
    - desktop
        - larger, general purpose languages
        - ex. C, C++, Java
    - web/server
        - ex. Java, PHP, JS
    - “quick/dirty” or scripting
        - often ‘glue’ stuff together
        - often advanced string options
        - ex. Python, JS, Ruby

## Basic Components that Influence Acceptance and Learnability

- portability
    - write once, run everywhere
    - Java & C are both portable, but for very different reasons
        - Java’s vm is written in C and/or C++
    - compile v.s. interpret is the BIGGEST factor for portability
    - interpretive languages “died” in the middle years b/c they were too “slow” ⇒ now, compiled is too slow for us (the programmers)
    - compiler:
        - translates ALL to lower level languages
        - src program → compiler → assembly language → assembler → machine code
    - interpreter:
        - translates line by line
        - read, eval, print loop ⇒ REPL
        - slow, but much easier to test out code snippets
        - src program → interpreter → output
          input →
          OR
          src program → translator → intermediate program → vm→ output
          input →
    - Java portability
        - interpreter ‘compiles’ Java to byte code
        - originally, it was purely interpreted
        - the byte code is then forwarded on to a virtual machine

        - virtual machine: a program that finishes the compilation to machine code… in a portable language → usually C
    - C is portable b/c it's SMALL
        - only 20 keywords
- simplicity
    - simple & ez to learn
    - too simple & things can be hard to write
- orthogonality
    - the syntax means the same thing (consistent) for ALL fundamental constants
- expressivity
    - how many ways to do the same thing?
    - orthogonality & expressivity *normally* affect each other
- reliability
    - less reliability = more flexibility, but more stuff for programmer to check
    - how does the languages handle something going wrong?
        - exception handling
        - some make it hard & just fail → C
        - some force you to catch them and deal with it → Java
        - different options are better for different circumstances

**Questions:**

1. **One of the ways we group languages is by generation. What is a generation? Name a particular generation and a language for that generation.**
   high level: C

---

# SOLID

- S → single responsibility principle
    - classes should have one responsibility
    - there should only by one reason to change the class… if there are two, you need another class 🙂
- O → open-close principle
    - "open for extension, but closed for modification"
    - extend a module if new methods/data are needed
    - use inheritance rather than potentially break tested code
- L → Liskov substitution principle
    - we can substitute a subclass for its parent & use it the same way ⇒ a.k.a. polymorphism
    - can overwrite a function & change result, but not meaning
    - breaking pre/post conditions → BAD
        - sign that "is-a" relationships are not correct
- I → interface segregation principle
    - no callee should be forced to depend on methods it does not use
        - a symptom of this is heavy special casing…

- D → dependency inversion principle
    - BOTH high & low level modules should depend on abstractions
    - use abstractions, not details
    - the implementation can change !!!
    - one of the most broken
- violation of these rules means code is “welded” together
    - lacks effective code reuse & flexibility
- code smells:
    - OOP abuse → SOID
        - improper use of OOP
    - bloating → SIO
        - file, function, or class has gotten enormous!
    - welding → SOLD
        - can’t change one class w/o changing another
    - repetition → OL
        - unneeded or repeated code
    - functional decomposition
        - making every function its own class
    - proper use of instance v.s. class variables
        - class:
            - affect ALL instances of the class
            - should be able to use w/o any state
            - relatively rare
            - often helpers & “factories”
        - instance
            - used 95%+ of the time
            - needs “state” information
            - effect varies between instances of the class

**Questions:**

1. **What is the O in SOLID & describe its purpose.**
   O is the open-closed principle. It says to extend rather than modify existing code.

---

## Visitor Concept

- why design patterns?
    - result in shorter & more readable code
    - use shortcuts for problems that have already been solved
- design of a program
    - iterative & collaborative process
    - you will not be building program from scratch or alone most of the time
    - upfront design
        - can assign modules to different ppl for efficiency & assign by specialty

        - can reuse libraries build by other ppl
        - more classes = less coupling ⇒ easier to reuse code & SDK updates are less likely to break everything
        - often, we think b/c we can see it, we should be able to edit it…
        - consistent design for maintenance requires an API structure
        - helps to find flaws early before code is written
- warning on patterns!!
    - patterns CHANGE with the problem
    - don't code patterns w/o thought to the problem
- manager
    - not an official "pattern"
    - just a fundamental part of class decomposition
    - decouple storage method from usage
    - intermediate class
        - builds in flexibility when we have an obj that owns many of another
        - don't ALWAYS need a manager
    - signs you need a manager:
        - over 8 items
        - 2+ classes need to access the collection
        - there is a many to many relationship
        - an operation only considers *some* elements in a collection
        - often, managers are added by default if you see a plural
    - signs you do NOT need a manager
        - adds another layer of access
        - if the collection's functions are all that is needed
        - you have a small set that have additional 'meanings'
- visitor pattern
    - makes it possible for us to work w/collections of different types of objects and do type-specific operations in a type-safe way
    - make an interface that ALL class types can accept called **Visitor**
        - each visitor interface needs a function for EACH type it cares about
    - add the function to accept the visitor to the base class (make it abstract so all derived classes will have access)
    - add the function to accept the visitor → override the function in the derived classes s.t. it calls the function it matches
    - make the concrete visitor class → that implements the visitor interface
        - have a check according to the type
    - call the visitor!
- when to use a visitor:
    - there are many distinct and unrelated operations
        - no operation → struct
        - many unrelated operations leads to class bloat that "pollutes" the classes
    - there are multiple types
        - a.k.a. many child classes

        - otherwise a regular manager probably works 🙂
    - the operations are type specific
        - if it's not type specific then inheritance works well
- visitors are best when the classes defining the object structure rarely change, but you often need to define new operations over that structure
    - simulations & video games

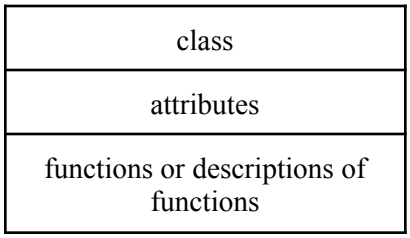
**Questions:**

1. **What is the visitor pattern's purpose?**
   Its purpose is to provide a clean interface when you have an operation on diverse classes. Without this, maintaining single responsibility would be difficult as would the open-closed principle.

---

## Basic OOP Diagraming

- motto of OOP: "those that know, do!"
    - if a class knows abt something, any functions on that data should be in that class too
- steps:
    - find nouns & adjectives
        - become classes & attributes
    - figure out what needs what
        - become connections & links
    - figure out verbs or tasks
        - become functions
        - adverbs become attributes needed by the tasks
    - diagram it!
    - don't add "what ifs"
- application diagrams will change as you code & realize you missed something
- just a graphical representation of the program, shouldn't include any language specific stuff




- class name matches actual class name
- attributes
    - name : data type
    - arrays: name : datatype []
- functions
    - major functions for the class
    - or just a description of the class's general jobs/tasks
    - fn (name, name…) : return type
    - if you KNOW you need it, add the function

- inheritance
    - shown with an arrow
    - interface implementation shown with a dashed arrow
    - interfaces tagged with << interface >>
- connection
    - shown with a line (no arrow) between two classes
    - the role is on the far end of the connection so it reads well in english
    - has-a
        - 1 ⇒ just 1
        - 0:1 ⇒ 0 or 1
        - x:y ⇒ x to y instances
        - * ⇒ many
    - its okay to have more than one connection between the same to classes
    - you MUST have a multiplicity or a very clear role, but having both is better
    - visitors have a use-a relationship with the main program, mark this as a note on the diagram
- add additional notes if the diagram is not sufficient
- public/private/protected is not mentioned
- abstract is not mentioned → requires a note
- don't put a connection line AND a member variable → redundant 😕
- don't add getters/setters

---

## Syntax Concepts

- portability
    - write once, run everywhere
    - Java & C are both portable, but for very different reasons
    - compile v.s. interpret is the BIGGEST factor for portability
    - interpretive languages "died" in the middle years b/c they were too "slow" ⇒ now, compiled is too slow for us (the programmers)
    - compiler:
        - translates ALL to lower level languages
        - src program → compiler → assembly language → assembler → machine code
    - interpreter:
        - translates line by line
        - read, eval, print loop ⇒ REPL
        - slow, but much easier to test out code snippets
        - src program → interpreter → output
          input →
          OR
          src program → translator → intermediate program → vm→ output
          input →
    - C is portable b/c it's SMALL

        - only 20 keywords
- simplicity
    - simple & ez to learn
    - too simple & things can be hard to write
- orthogonality
    - the syntax means the same thing (consistent) for ALL fundamental constants
- expressivity
    - how many ways to do the same thing?
    - orthogonality & expressivity *normally* affect each other
- reliability
    - less reliability = more flexibility, but more stuff for programmer to check
    - how does the languages handle something going wrong?
        - exception handling

**Questions:**

1. **The grammar or context-free part of a language is called what?**
   Syntax

2. **The meaning of a language is called what?**
   Semantics

---

## Object Oriented Programming

- what is an OOP language?
    - minimum for objects:
        - to bundle data & "legal"...
    - fundamental concepts
        - encapsulation
        - inheritance
        - dynamic method binding (polymorphism) ⇒ w/o NOT considered OOP (it's possible to have objects but no OOP tho…)
- encapsulation
    - access levels
        - public, private, protected
- inheritance
    - reuse code by sharing attributes/methods with parent
- dynamic method binding
    - ability to use derived class in a context that expects the base class instead (can substitute derived for base)
    - abstract classes
    - virtual methods
- OOP v.s. modules
    - modules permits some encapsulation, but are harder to debug → common in operating systems

- why OOP?
    - predominant paradigm for over 30 years
    - OOP's preliminary goals:
        - easier debugging
        - maintainability
        - flexibility
        - decoupling
        - code reuse
    - does NOT have speed
        - OOP as an operating system driver is a BAD idea

**Questions:**

1. **Select all of the following that should use a manager. Your other options would be normal classes, struct-like structures, and inheritance.**

    ☐ **In graphics, one item often owns another. Specifically, say you have a post that owns a propeller. Check if the propeller should use a manager.**

    ☐ **You have a line with 2 distinct points that need to be called directly.**

    ☐ **You have a collection of letters (the count is limited only by hardware) that will fill 0 or more boxes every 10 minutes.**

    ☐ **You have an array of yards of at most 200 yards, that you need to make a schedule to mow them all.**

    At minimum, a manager needs a collection of multiple types of sufficient size. This is so the implementation of the storage can change. The most restrictive is that a manager needs a collection of sufficient size, of one type, and there is an operation on the collection.

    There are only 2 points in the line, and no operation. This is not a collection of sufficient size. Moreover, we could not call the points directly with a manager. A manager has no logical reason to be here.

    You have a collection of letters, AND you have an operation that needs to look at sets of them. This meets the most restrictive requirements for a manager. It is an extremely good time to have a manager.

    The post to propeller is a 1 to 1 relationship. No collection, no manager.

    The yards may have a set max, but it is a collection of sufficient size. Also, there is an operation that needs to be done. This meets the most restrictive requirements for a manager. It is an extremely good time to have a manager.

2. **OOP abuse is code smell. Explain why.**
   It shows a lack of understanding of the problem and can easily cause repeat code in the future.

3. **Name a disadvantage of using OOP.**
   Speed, possibility of improper use

---