CMSC403 Final Exam Study Guide

Unit 1:

Define the von Neumann architecture and how it relates to the Fetch-Decode-Execute cycle

Contains below

- A processing unit that contains an arithmetic logic unit and processor registers
- A control unit that contains an instruction register and program counter
- Memory that stores data and instructions
- External mass storage
- Input and output mechanisms

Fetch-Decode-Execute cycle

This is the cycle that the (CPU) follows from boot-up until the computer has shut down in order to process instructions. It is composed of three main stages: the fetch stage, the decode stage, and the execute stage.

- Fetches a program instruction from the main memory
- Decodes the instruction, i.e. works out what needs to be done
- Executes, i.e. carries out, the instruction

Define the von Neumann bottleneck and understand how it is related to the Processor-Memory performance gap

The von Neumann bottleneck is the idea that computer system throughput is limited due to the relative ability of processors compared to top rates of data transfer. According to this description of computer architecture, a processor is idle for a certain amount of time while memory is accessed. The CPU executes instructions much faster than memory can give them to the CPU.

Know the computational paradigms discussed in class

- **Imperative** – Sequential instruction execution, variables representing memory locations.
- **Functional** – Treats programs as functions, treats functions as data.
- **Logic** – Based on formal symbolic logic.
- **Scripting** – Coordination or concatenation of components from some surrounding context.
- **Object Oriented** – Allows programmers to write reusable code that mimics the behavior of objects in the real world.

Be able to compare and contrast a compiler, an interpreter, and a hybrid implementation system

compilation method has programs translated to machine language before execution, while the interpreted method has programs translated into machine language at runtime. Compilation method usually leads to greater performance than Interpreted method. Interpreted languages are slower to execute. It is easier to implement an interpreter. Additionally, the compilation process has several phases, where the interpretation process is much simpler.

The hybrid implementation combines both compiler and interpreter implementations into one.

If it's a hybrid system, so having those hallmarks of compilation, if it runs through does a transformation but also an analysis and can report errors at a time other than when the code is being executed

Be able to define a Just in Time Compiler

In Just-In-Time, java source code is compiled into .class bytecode at compile time, then in run-time the JIT compiler compiles the bytecode of that method into native machine code.

JIT is part of JVM. It compiles bytecodes to machine code at run time.

Be able to name and define the language design criteria focus areas

Efficiency

- Usually thought of as efficiency of the target code.

Example: Static data typing, enforced at compiled time means that the runtime does not need to check the data types before executing operations. This is an efficiency of a programming language. So, you don't have to do any data checking at runtime.

Aside from "zoom" efficiency, another way we can look efficiency Is **PROGRAMMER EFFICIENCY.** How quickly and easily can I read and write in the programming language?

- **Writability:** The quality of a language that enables a programmer to use it to express computation clearly, correctly, concisely, and quickly.
- **Expressiveness:** How easy is it to express complex processes and structures? "{}" block structure in general
- **Readability:** How easy is it to comprehend the computations in a program?
-

Regularity – Refers to how well features of a language are integrated.

- **Generality:** Achieved by avoiding special cases in the availability or use of constructs and by combining closely related constructs into a single more general one. A language with this property avoids special cases whenever possible.
- **Orthogonal Design:** Where constructs can be combined in any meaningful way with no unexpected restrictions or behavior. Constructs should behave the same when used in different contexts. In a language that is truly orthogonal, constructs do not behave

differently in different context. **Restrictions that are context dependent are non-orthogonal. Restrictions that apply regardless of context exhibit a lack of generality.** An example of a lack of orthogonality can include function return types. Pascal allows only scalar or pointer types as return values. C and C++ allows values of all data types except return types. Ada and Python allow all data types. Another example of lack of orthogonality is that only arrays can contain primitive types; other collections (e.g. ArrayList) must contain objects. Java gets around this by having wrapper classes where you can basically store integers in an ArrayList.

- **Uniformity:** Design in which similar things look similar and have similar meanings, WHILE different things look different and have different meanings. A lack of uniformity means that the same things look different, and that different things look the same.

```
1  function gcd(x,y:longint):longint;
2  begin
3   while x<>y do
4    begin
5     if (x>y) then x := x-y
6     else y := y-x;
7    end;
8   gcd := x;
9  end;
```

Security

- Related to reliability
- A language designed with security in mind discourages programming errors and allows errors to be discovered and reported
- Types, type-checking and variable declarations resulted from a concern for security. So by doing type checking at compile-time, if there is a type error you'll know at compile time and you'll know at line xyz.

Extensibility

- An extensible language is a language that allows the user to add features to it.
- For example, the ability to define new data types and new declarations.
- Extend built-in features with new releases.
- Allow user defined types(C++, Java, etc)

Know the two types of programming language abstractions and the three levels of programming language abstractions

Two types of programming language abstractions

Data abstraction – Simplifies the behavior and attributes of data for humans

Control abstraction – Simplifies properties of the transfer of control (Assembly language and looping)

Three levels of programming language abstractions

Basic abstractions – Collect the most localized machine information

Structured abstractions – Collect intermediate information about the structure of a program

Unit abstractions – Collect large scale information in a program

Unit 2:

Be able to define and explain the difference between syntax and semantics
Syntax: The **structure** of the expressions, statements, and program units
Semantics: The **meaning** of the expressions, statements, and program units

Know the alternate methods for defining syntax discussed in lecture
Original BNF, extended BNF, syntax diagrams

Know why EBNF is used rather than these other methods
It's rules are often cleaner and simpler

What it means for a grammar to be context free
Context-free grammar: A formal specification of syntax which consists of a series of grammar rules

What it means for a grammar to be ambiguous
Ambiguous Grammar: A grammar is ambiguous when a single statement can have multiple grammatically valid derivations

Be able to determine if a given grammar has a finite or infinite number of legal derivations
Grammars are infinite when there is at least one recursive or iterative in the case of ENBF's rule.

If it is a finite number, be able to determine the number of legal derivations
To determine the number of legal derivations, first generate a string from said language and see what non-terminals are in the final form of the string subsequently taking the rule of products. EX. grammar below

1) sentence → noun-phrase verb-phrase
2) noun-phrase → article noun
3) article → a | and | the
4) noun → girl | competitor | win | dog | comp
5) verb-phrase → verb noun-phrase
6) verb → sees | permits | objects

Generating a sentence "a girl sees a dog", we see that the final format is of the form (article noun verb article noun). Article has 3 rules, noun has 5 rules, verb has 3 rules, so if you do the rule of products on this we have the expression 3 x 5 x 3 x 3 x 5 = 675 legal sentences that can be derived from this language.

How to add a disambiguating rule to a given EBNF grammar

How to generate both a parse tree and an abstract syntax tree if given a grammar and a valid sentence from said grammar Make sure to know all the conversion steps from parse tree to ABST

Understand the following regular expression operations: Concatenation, Repetition, Choice, Range, Optional, Wildcard

- **Range - Square Brackets**
- **Optional - Question marks**
- **Concatenation- done by sequencing the items without an explicit operation, parentheses can be included to allow for grouping of operations**
- **Repetition- indicated by an asterisk/plus after the item to be repeated**
- **Choice or selection: indicated by a vertical bar between items to be selected**

Define lexis, lexeme, and token
Lexis: Short for Lexical structure (The structure of the terminals in a language)
Lexeme: Logical groupings of characters
Token: Categories of lexemes

Be able to define the principle of longest substring
Eliminates confusion

- **This is the process of collecting the longest possible string of nonblank characters to determine lexemes**
- Ex: The longest possible string of nonblank characters is collected into a lexeme, so doif and x12 are identifiers

Understand the difference between free format and fixed format
Free format language: One in which format has no effect on program structure other than satisfying the principle of longest substring.

Fixed format language: One in which specific tokens must occur in pre-specified locations on the page

Know the two parts of syntactic analysis and know why they are two distinct parts
Syntactical analysis portion of a language processor consists of two parts:

- Lexical structure → Scanning & Tokenizing
- Syntactic structure → Parsing

Scanning Phase: The phase in which a translator collects lexemes from the input programs and associates them into tokens

Parsing Phase: The phase in which the translator processes the tokens, determining the program's syntactic structure

Be able to define and describe a lexical analyzer (i.e. a definition, not a program)
A lexical analyzer is a pattern matcher for character strings. It identifies substrings of the source program that belong together - *lexemes*

Know the relationship between a lexical analyzer and a parser
Lexical
Lexical analyzer takes a form of input and turns it into tokens / lexemes etc such that a parser can take those tokens and turn them into another form of data.

Be able to name and define the two parser classes discussed in lecture
Left-to-right, Left-most derivation or LL (Also called "Top-down" or "predictive" parsers)

- Constructs a parse tree from the root down, predicting at each step which production will be used to expand the current node, based on the next available token of input.
- Easier than LR parsers
- Can be written by hand or generated automatically

Left-to-right, Right-most derivation or LR (Also called "Bottom-up" or "shift reduce" parsers)

- They construct a parse tree from the leaves up, recognizing when a collection of leaves or other nodes can be joined together as the children of a single parent
- LR parsers are almost always constructed by a parser-generating tool

Be able to define and describe a recursive descent parser
Recursive descent parser: A LL parser whose functions correspond one-to-one, to the Nonterminals (i.e. the right-hand sides of productions) of the grammar

- Terminals are matched directly with input tokens as constructed by a scanner
- Nonterminals are interpreted as calls to the procedures corresponding to the terminals

Know the relationship between left recursive grammar rules and recursive descent parsers
Recursive descent parser: A LL parser whose functions correspond one-to-one, to the Nonterminals (i.e. the right-hand sides of productions) of the grammar

- Terminals are matched directly with input tokens as constructed by a scanner
- Nonterminals are interpreted as calls to the procedures corresponding to the terminals

Left recursive grammar rules: Curly brackets in EBNF represent left recursive removal by the use of a loop
Be able to define single-symbol lookahead and predictive parser
Single symbol-symbol lookahead: Using a single token to direct an LL parse

Predictive parsers Construct a parse tree from the root down, predicting at each step which production will be used to expand the current node, based on the next available token of input.

Know the difference between static semantics and dynamic semantics

Static Semantics

- Indirectly related to the meaning of programs during execution
- Compile time (data typing, etc)
- Static semantics define context dependent structure

Dynamic Semantics

- How and when various constructs of a language should produce behavior (What the language constructs actually do)
- Actual meaning of the language

Understand why EBNF grammars are not sufficient in describing syntax

Because EBNFs are context free and in most cases where EBNFs are used you need to use them based on context.

Be able to define attribute grammar

An **attribute grammar** is a context-free grammar with the following additions:

- Each nonterminal is associated with a set of attributes
- Each production has a set of rules that define attributes of the Nonterminals

Know the two types of attribute grammar rules discussed in lecture

Copy rules: Specify that one attribute should be a copy of another

Semantic functions: Arbitrarily complex functions specified by the language designer

- Each semantic function takes an arbitrary number of arguments, and each computes a single result, which must likewise be assigned into an attribute of a symbol in the current production

Know the two types of attributes discussed in class

Synthesized attributes: Values are calculated (synthesized) only in productions in which their symbols appears on the left-hand side

Inherited attributes: Attributes whose values are calculated when their symbol is on the right-hand side of the current production

Be able to define what translation scheme is and describe the scheme we covered in class

Translation scheme: An algorithm that decorates parse trees by invoking the rules of an attribute grammar.

- Simplest scheme is one that makes repeated passes over a tree, invoking any semantic function whose arguments have all been defined, and stopping when it completes a pass in which no values change (Such a scheme is said to be oblivious)

Know there is no single widely accepted way of formally defining dynamic semantics

Know the most common way of informally defining dynamic semantics

Language reference manual

Be able to name the three major classes of formal notation
Operational Semantics
Axiomatic Semantics
Denotational Semantics

Unit 3 – The Symbol Table & The Environment

Be able to define the concept of binding and know the difference between static and dynamic binding
Binding is the process of associating an attribute with a name. It is the association of objects and implementations with names in a programming language so that those objects and implementations can be accessed by the names.

The two categories of binding include: **Static binding,** which occurs prior to execution, and **dynamic binding,** which occurs during execution. Static attributes can be bound during translation, during linking the program with libraries, or during loading of the program. Dynamic attributes can be bound at different times during execution, such as entry or exit from a procedure or from the program.

Define the symbol table, environment, and memory in the context of binding
The symbol table expresses the bindings of attributes to names. The environment expresses the bindings of names to memory locations. The memory expresses the binding of memory locations to values.

Know and define the three principle storage allocation mechanisms
Static objects, which are given an absolute address that is retained throughout the program's execution.
Stack objects, which are allocated and deallocated in last-in, first-out order, usually in conjunction with subroutine calls and returns.
Heap objects, which may be allocated and deallocated at arbitrary times.

Know why language using static allocation exclusively cannot support recursion
Since the program cannot have more than one active subroutine to prevent runtime overhead.

Know what an activation record is and what it is comprised of
An **activation record** is a data structure containing the important state information for a particular instance of a function call (or something that resembles a function call)
It is comprised of - arguments, return values, local variables, temporaries, and bookkeeping information

Created during function runtime call.

In terms of space complexity, be able to explain why stack allocation is better than static allocation

Natural nesting within the subroutine calls make it easier to allocate space.

Know at least two reasons why a heap would be needed for storage allocation

- If you don't know how much data you will need at runtime
- If you need to allocate a lot of data

Know what a dangling reference is

A **dangling reference** is a reference to an object that no longer exists. They arise during object destruction, in the event when an object that has an incoming reference is deleted or deallocated, without the modification of the value of the pointer, so that the pointer still points to the memory location of the deallocated memory.

Know what garbage is

Garbage is memory that has been allocated in the environment but that has become inaccessible to the program. Can occur in C by failing to call free before reassigning a pointer variable.

Know and be able to define the three types of memory management discussed in class

Garbage collection, which is the process of automatically reclaiming garbage. Java is an example of a programming language that uses garbage collection to automatically reclaim garbage. Garbage collection offers increased reliability, along with less developer time chasing memory management errors.

Automatic Reference counting, which determines objects references at compile time. Injects allocation and deallocation instructions into the compiled program. Allows for efficiency of manually managed memory with security of garbage collection.

Manual memory management

Refers to the usage of manual instructions by the programmer to identify and deallocate unused objects, or garbage.

Understand the difference between static and dynamic scoping

- **Be able to determine a programs output using either static or dynamic scoping**
- **Be able to generate a syntax table at a specific point in a program's execution using either static or dynamic scoping**

- Type errors can be caught before runtime. They are reported as compilation error messages that prevent execution. Therefore, no effort is being wasted if the operations being done with those variables are invalid. The disadvantages of statically typed languages include having to wait for compiler when debugging. Compilation can take more than 10 seconds for a larger project.

- Dynamically typed languages are more succinct and less verbose along with a marginally quicker development process. Disadvantages include: If you are not careful sanitizing user input, there can be faulty errors not noticed at compile time, for example trying to add a number 5 with the string"6". It would result in the string "56", instead of 11 as you were expecting. Also, type errors are usually more difficult to locate.

Know what a lexical address is and how it is used in static scoping

- The combination of lexical depth and position is called a lexical address.
- Each name has a lexical address which is composed of a level number and offset

Be able to define the concept of a scope hole

- **Know what it means when a local variable shadows a global variable**

Local variable is used in the function instead of the global variable

Be able to define overload resolution

- **This includes what it is, how it works, and what entities it can apply to and why**

Process of choosing a unique function among many with the same name

Know the difference between simple types and complex types

Simple Types – Have no other structure than their inherent arithmetic or sequential structure
Complex Types – Types like arrays, objects, etc.

Understand what a type constructor does and how it relates to basic and complex types

Type constructors use a group of basic types to construct more complex types.

Know the different simple types discussed in lecture (POES)

Predefined – On the flashcards
Enumerated – Sets whose elements are named and listed explicitly.
Subrange – Contiguous subsets of simple types specified by giving least and greatest elements
Ordinal – Types that exhibit a discrete order on the set of values (integers, comparison operators, successor and predecessor operations)

Know the different type constructors discussed in lecture (CUAPS)

- **Cartesian Product, union, subset, function and arrays, pointers**

Be able to define type equivalence and know the difference between structural equivalence and name equivalence

Structural equivalence is when two data types are the same only if they have the same structure.

Name equivalence is when two types are the same only if they have the same name.

Understand the difference between a statically typed language and a dynamically typed language

Static – Types are determined from the text of the program and checked by the translator.
Dynamic – Type information is maintained and checked at runtime

Define type conversion

The difference between implicit and explicit conversion

Implicit – conversion inserted by the translator

Explicit – conversion directives are written into the code

The difference between widening conversion and narrowing conversion

Widening conversion – Target data type can hold all of the converted data without data loss.

Narrowing conversion – Conversion may involve a loss of data.

Know the difference between casting and conversion

Casting – Interpreting the data type as another
Conversion – Changing the data type

Does casting (as we have defined it) have a purpose in a dynamically typed language?

No because data types are managed dynamically, so there would be no point.

Unit 4 - Common Lisp:

Know what a cons cell is and what its two components are

Cons cell is a pair of values.

- Car: returns the first element of the given cons cell, the head of our list
- Cdr: returns the second element of the given cons cell, the rest of our list

Know the following about lists:

- **How cons cells are utilized to create a list**

  List L is a pointer to a cons cell, with the car storing the first value and cdr storing a pointer to the next cons cell.

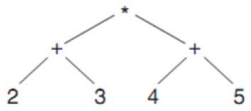
- **Write a statement which returns a built list**

  '(1, 2, 3, 4) or (list 1 2 3)

- **How a list is evaluated**

  To evaluate a list, the list must start with an atomic operator. The list will be evaluated in one of three different ways, depending on the type of atomic operator at the start of the list. If the starting atomic operator is a function, such as car, cdr, cons, +, -, we evaluate the remaining elements of the list as either literals or variables and pass the resulting variables to the named function. All list must be written in prefix form. Use applicative order evaluation – sub lists evaluate first.

- Example: `(* (+ 2 3) (+ 4 5 ))`
  - Two additions are evaluated first, then the multiplication



- Answer: `(* 5 9)` → `45`

Explain what the REPL loop is (including what the acronym stands for)
Environment that reads input, evaluates it, prints it, and waits for next user input.
Read Evaluate Print Loop

- Reads in user input
- Evaluate input
- Print Evaluation
- Waits for next user input

Know how to implement selection statements

- **Both if-else and if-else-if design patterns**

- Both if-else and if-else-if design patterns

- if else
  - (if (=3 3) (print "true") (print "false"))
- if-elseif
  - (if (condition) (true branch) (- if
    (if (condition) (true branch) - else if
    (false branch) - else
    ))

Be able to define referential transparency and side effects
Referential transparency: The property where the value of any function depends only on its arguments. The **side effects** include a change in the value of a variable that persist beyond the local scope.

Remember that functional languages lack assignment

Know how to define a function in Common Lisp
(defun name (parameters)
"Optional Documentation String"
(body))

- **Both a global function and an anonymous function**
- **Know how functions return values**

- **Know the four types of function parameter and how to utilize them in a function's parameter list (Required, optional, rest, key) RORK**

Understand what a special operator is and why they are necessary
It is a type of atomic operator defined to behave in ways a function cannot.

Understand the difference between the variable namespace and the function namespace
The first value in a list is treated as a function and is searched for in the function namespace, while the other variables are searched for in the variable namespace.

Be able to define tail recursion
When the recursive step is the last step in any function. So a function is tail recursive, when the last thing a function has to do, is the recursive call.
Know the two benefits of tail recursion relative to standard recursion
- Does not incur stack depth issues involved with regular recursion. Since tail recursions directly return the result to the parent, the stack frame is discarded when the next recursive call is made.
- Don't have to take the time to create new stack frames, so there is a speed benefit.
- Speed efficiency boost, memory boost

Know that functions are first class data values / functions as data
- In functional programming, functions are **first-class data values**
- **Functions are treated as any other value (are first-class citizens)**
  - Functions can be computed by other functions
  - Functions can be parameters to other functions
- Functions that take other functions as parameters and functions that return functions as values are called **higher-order functions**

- **Be able to implement a function which takes a function as a parameter**
  - **ex - (defun bigDome (f x) (funcall f x x))**
  - **(define (adder n) (lambda (x) (+ x n)))**
    - **((adder 3)5)**
- **Be able to implement a function which returns a function as data**

Remember that Common Lisp uses Garbage Collection

Be familiar with the let, if, cond, defvar, defun, funcall, & lambda structures
- **You will be expected to do both code analysis and basic programming in Common Lisp for the exam**

Unit 5 – Python:

Are Python 2 and Python 3 interchangeable?

No. Python 2 is depreciated and is significantly different from Python 3. Solutions which work in Python 2 will most likely not work in Python 3.

Know what a list and tuple is in Python and be able to describe the similarities and differences between the two.

Lists: Ordered and accessed by index, mutable, able to hold objects of arbitrary type, able to hold objects of different types, dynamically sized.

Tuples: Similar to lists but possess some key differences. They are immutable, support sequence packing and unpacking, support access by attribute

Similarities: They are both sequence data types that store a collection of items. They can store items of any data type.

Differences: Tuples are immutable, lists are mutable. Lists consume more memory compared to Tuples. Tuples have structure while lists have order.

Understand how Python supports multiple assignment and know sequence packing and unpacking

Multiple assignment

- x = “Dahlberg is cool”
- y = “mainframe is cool”
- x, y = y, x
- print (x) # mainframe is cool
- print (y) # Dahlberg is cool

Sequence packing and unpacking

- x, y, z = 1, 2, 3
- w = x, y, z
- print (w) # (1, 2, 3)

Know that Python is pass-by-value and statically scoped

Know how to define an object class

```
class Mainframe
    master = None # this is a class variable. (must use self for instance variables)
        def __init__(self):
            self.master = “dahlberg”
            self.dome = “quad”
```

Know the difference between class variables and instance variables

Class variables are defined within the class construction. They are shared by all instances of the class and will generally have the same value for every instance.

Instance variables are owned by instances of the class. For each object or instance of a class, the instance variables are different. Instance variables are defined within methods.

Know how to add variables and functions to an object instance or an object class "on the fly"

- setattr (variable, name, value)

- call class name with needed class instance like Dog.type = 'sometype' after class declaration. This will modify all Dog objects
- can add instance variables to particular instances of object in a similar manner.

Know the described access modifiers

x = Public – Accessible from outside the class through an object of the class
_y = Protected – Accessible from the package, the class, and are also available to its sub-classes
__z = Private – Can only be accessed inside the class

Know the difference between a deep copy and a shallow copy

In a shallow copy, object B points to object A's location in memory. In deep copy, all things in object A's memory location get copied to object B's memory location.

Deep copy – Constructs a new compound object and then recursively inserts copies into it of the objects found in the original.

Shallow copy – Constructs a new compound object and then (to an extent possible) inserts references into it to the objects found in the original.

Know how to do a deep and shallow copy in Python

- copy.copy(x)
- copy.deepcopy(x)

Know the syntax for inheritance in Python

- class Subclass (SuperClass1, SuperClass2, …):

Know about multiple inheritance and the deadly diamond of death

Deadly diamond of death is the generally used term for an ambiguity that arises when two classes B and C inherit from a superclass A, and another class D inherits from both B and C. If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?

***Be able to define Python's typing and binding paradigm (both what it is and how it works)**
Python is dynamically typed and dynamically bound. It is dynamically typed because it does not have data type defined in the variable so there is no type checking during compile time. type checking is looking at two data are compatible the program is doing operation. if it is incompatible then python will throw an exception. Dynamic bound means that all method call is looked up at runtime by name

Understand slice and range: how they work, how they are similar, how they are different.
Slice is used to slice a given sequence (string, bytes, tuple, list, or range) or any object which supports sequence protocol.

Range generates the immutable sequence of numbers starting from the given start integer to the stop integer.

In Python, ranges and slices are very similar

- Both conform to the start, stop, and step design pattern

Subtle differences between the two

- Range objects are iterable and slice objects are not
- Slices can be used as indices, ranges can not
- You can slice a range, but you cannot range a slice

Know why Python is able to concisely implement functional constructs.
Python treats functions as data

Be able to identify a closure, a decorator, and a list comprehension
Closure

```
def make_multiplier_of(n):
    def multiplier(x):
        return x * n
    return multiplier

times3 = make_multiplier_of(3)   # Multiplier of 3
times5 = make_multiplier_of(5)   # Multiplier of 5

print(times3(9))                 # Prints: 27
print(times5(3))                 # Prints: 15
print(times5(times3(2)))         # Prints: 30
```

- A decorator takes in a function, adds some functionality and returns it

```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner
def ordinary():
    print("I am ordinary")
ordinary()                          # Prints "I am ordinary"
pretty = make_pretty(ordinary)      # Prints "I got decorated"
pretty()                            # Prints "I am ordinary"
```

The function ordinary() got decorated and the returned function was given the name pretty.

- Tool for transforming an iterable into another iterable
  - Elements can be both transformed and conditionally included
- List comprehensions are composed of four parts:
  - The result
  - The transformation
  - The collection
  - The conditional

```
foo = [1,2,3,4,5,6,7,8,9,10]
doubled_odds = [n * 2 for n in foo if n % 2 == 1]
print(doubled_odds)
#[2, 6, 10, 14, 18]
```

Know the four ways software components can be modified for reuse

Extension of the data or operations

- Example: Adding new methods to a queue to allow elements to be removed from the rear and added to the front to create a double-ended queue or dequeue.
- Example: A window defined on a computer screen that is specified by its four corners extended to display text becoming a text window

Redefinition of one or more of the operations

- Example: If a square is obtained from a rectangle, an area or perimeter function may be defined to account for the reduced data needed in the computation
- Example: A window extended to display text must redefine the display operation in order to display the text as well as the window itself

Abstraction

- Example: A circle and rectangle object can be combined into an abstract object called a figure, to contain the common features of both, such as position and movement

Polymorphism

- Extending the types that an operation applies to such as the toString method, that is applicable to any object as long as it has a textual representation

Know low coupling, high cohesion. What it is and why it's important

Coupling: how related or dependent two classes are toward each other

Low Coupling: Different program units aren't highly dependent on each other

Cohesion: what the class can do

- Low cohesion: a class has a broad focus
- High cohesion: a class has a narrow focus

Why its important: Low coupling and high cohesion. This is done to ensure that you code is better maintained. If you were to change a module and it affects another module then it would not have low coupling. This makes updating or coding overall harder as you'll have to backtrace to parts for debugging. The same for high cohesion, having a class with a narrow focus helps prevent errors with classes that will have a wide focus that will affect more code.

Know the three principles of Object Oriented Programming and be able to describe them

Encapsulation

Packaging of data with a set of operations that can be performed on the data. Hides internal representation, or state, of an object from the outside.

Inheritance

A new class can be defined to borrow behavior from another class. The new class is called a subclass, and the other (the one being borrowed from) is called a superclass

Polymorphism

Characteristic of allowing an entity such as a variable, function or object to have more than one form based on the different contexts in which it is used. Gives object-oriented systems the flexibility for each object to perform an action just the way that it should be performed for that object.

Unit 6 – Java:

Know what an Anonymous Inner Class is in Java

A class without a name and which only a single object is created

Know what a functional Interface is in Java

An interface with a single abstract method.

Know the design pattern of lambda expressions in Java relative to functional interfaces.

Lambda expressions create an anonymous object which implements a functional interface

Can lambda expressions create functions as data?

No.

Describe a Java stream It is basically taking a collection of objects and applying intermediate functions on said collection and then finishing with a terminal operation.

Know that streams are computed lazily

Allows for optimization

Understand what it means for a streams behavioral parameters to be non-interfering and why this is necessary for parallel execution of streams

For a streams behavioral parameters to be non-interfering means they do not modify the streams data source which is necessary for a number of reasons, first and foremost most stream data sources do not support concurrent operation

Know how to create a thread in Java

Know how interrupting a thread works in Java
Causes an interrupted exception. (Request is made to interrupt. You can't actually stop (hence stop being deprecated)).

Even if the interrupt flag is set, the thread may ignore the interrupt flag and continue to run

Know why stop and suspend are deprecated
Suspend is deprecated since it is deadlock prone. Stop is deprecated since its considered "unsafe" (unlocks all monitors that it has locked …).

Know about the concept of Monitors in Java
Monitors are an abstract data type that run only one thread at a time through it. Resolves critical sections.

Understand how the synchronize keyword works
keyword is fundamental for programs that use multiple threads. Ordinarily, if you have multiple threads all accessing the same data concurrently, there is risk of a so-called race condition. For example, one thread could see data in an inconsistent or out-of-date state because another thread is in the process of updating it. If the two threads are attempting to perform competing operations on the data, they could even leave it in a corrupt state.

Understand how the wait and notify methods work and where they are defined
Wait() blocks a thread until another thread calls notify on the same object.
Notify – NotifyAll signals any waiting threads to resume

Be able to define Java Reflection
Java Reflection is the process of analyzing and modifying all the capabilities of a class at runtime. Reflection API in Java is used to manipulate class and its members which include fields, methods, constructor, etc. at runtime. (Java reflection is a feature which allows programmers to examine and modify the runtime behavior of a program).

Includes: Properties of a class, including its declaration and contents, members of a class, fields, methods, and constructors, the properties of array's and enumerated types.
Understand the main advantage and disadvantage of reflection

Advantages: Allows programs to be more dynamic by subverting the static typing system. Allows for objects to be modified at runtime regardless of access modifiers.

Disadvantages: People can backdoor your stuff. Subverts static typing. Makes programs unsafe.