Programming Concepts Using Java

Quiz 1 – Revision

W01:L01: Introduction

- Programming languages
  - A language is a medium for communication
  - Programming languages communicate computational instructions
  - Originally, directly connected to architecture
  - Tedious and error-prone
- Abstractions used in computational thinking
  - Assigning values to named variables, Conditional execution, Iteration, Functions / procedures, recursion, Aggregate data structures — arrays, lists, dictionaries
- Express such ideas in the programming language
  - Translate "high level" programming language to "low level" machine language – Compilers, interpreters
  - Less control over how code is mapped to the architecture
  - But fewer errors due to mismatch between intent and implementation

W01:L01: Introduction (Cont.)

- Styles of programming
  - Imperative
    - How to compute
    - Step by step instructions on what is to be done
  - Declarative
    - What the computation should produce
    - Often exploit inductive structure, express in terms of smaller computations
    - Typically avoid using intermediate variables
    - Combination of small transformations — functional programming
- Abstract datatypes, object-oriented programming
  - Collections are important
    - Arrays, lists, dictionaries
  - Abstract data types
    - Structured collection with fixed interface
    - Stack is a sequence, but only allows push and pop
    - Separate implementation from interface
  - Object-oriented programming
    - Focus on data types
    - Functions are invoked through the object rather than passing data to the functions
    - In Python, mylist.sort() vs sorted(mylist)

W01:L02: Types

- Interpreting data stored in binary in a consistent manner
- Naming concepts and structuring our computation - `Point` vs `(Float, Float)`
- Catching bugs early
- Dynamic vs static typing
  - Every variable we use has a type
  - How is the type of a variable determined?
  - Python determines the type based on the current value
    - Dynamic typing — names derive type from current value
    - Difficult to catch errors, such as typos
  - Static typing — associate a type in advance with a name

W01:L03: Memory Management

- Variables have scope and lifetime
  - Scope — whether the variable is available in the program
  - Lifetime — whether the storage is still allocated
- Memory stack
  - Each function needs storage for local variables
  - Create activation record when function is called
  - Activation records are stacked
    - Popped when function exits
    - Control link points to start of previous record
    - Return value link tells where to store result
  - Scope of a variable
    - Variable in activation record at top of stack
  - Lifetime of a variable
    - Storage allocated is still on the stack
- Passing arguments to functions
  - Call by value – copy the value – Updating the value inside the function has no side-effect
  - Call by reference – parameter points to same location as argument – Can have side-effects

W01:L03: Memory Management (Cont.)

- Heap is used to store dynamically allocated data
  - Outlives activation record of function that created the storage
  - Need to be careful about deallocating heap storage
  - Explicit deallocation vs automatic garbage collection

W01:L04: Abstraction and modularity

- Solving a complex task requires breaking it down into manageable components
  - Top down: refine the task into subtasks
  - Bottom up: combine simple building blocks
- Modular software development
  - Use refinement to divide the solution into components
  - Build a prototype of each component to validate design
  - Components are described in terms of
    - Interfaces — what is visible to other components, typically function calls
    - Specification — behaviour of the component, as visible through interface
- Programming language support for abstraction
  - Control abstraction
    - Functions and procedures
    - Encapsulate a block of code, reuse in different contexts
  - Data abstraction
    - Abstract data types (ADTs)
    - Set of values along with operations permitted on them
    - Internal representation should not be accessible
    - Interaction restricted to public interface

W01:L04: Abstraction and modularity

- Object-oriented programming
  - Organize ADTs in a hierarchy
  - Implicit reuse of implementations — subtyping, inheritance

W01:L05: Object-oriented programming

- Objects
  - An object is like an abstract datatype
    - Hidden data with set of public operations
    - All interaction through operations – messages, methods, member-functions, ...
  - Uniform way of encapsulating different combinations of data and functionality
    - An object can hold single integer — e.g., a counter
    - An entire filesystem or database could be a single object
  - Distinguishing features of object-oriented programming – abstraction, subtyping, dynamic lookup, inheritance
- Abstraction
  - Objects are similar to abstract datatypes
    - Public interface
    - Private implementation
    - Changing the implementation should not affect interactions with the object
  - Data-centric view of programming – Focus on what data we need to maintain and manipulate

W01:L05: Object-oriented programming (Cont.)

- Subtyping
  - A subtype is a specialization of a type
  - If `A` is a subtype of `B`, wherever an object of type `B` is needed, an object of type `A` can be used
    - Every object of type `A is also an object of type B`
    - `Think subset -- if` $X \subseteq Y$, `every` $x \in X$ `is also in` Y
- Dynamic lookup
  - Whether a method can be invoked on an object is a static property — type-checking
  - How the method acts is a dynamic property of how the object is implemented
- Inheritance
  - Re-use of implementations
  - Usually one hierarchy of types to capture both subtyping and inheritance
    - `A` can inherit from `B` iff `A` is a subtype of `B`
  - Philosophically, however the two are different
    - Subtyping is a relationship of interfaces
    - Inheritance is a relationship of implementations