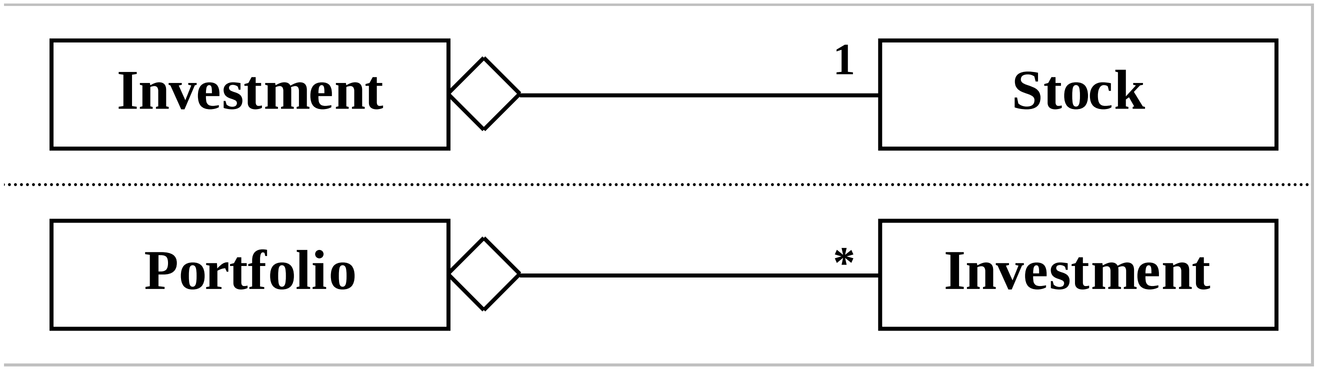
Java By Abstraction: Chapter 8

Aggregation

Some examples and/or figures were borrowed (with permission)
from slides prepared by Prof. H. Roumani

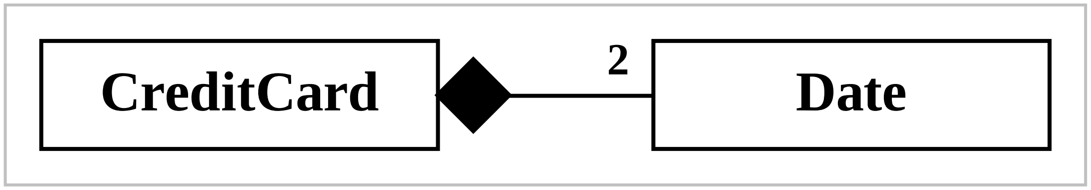
Aggregation

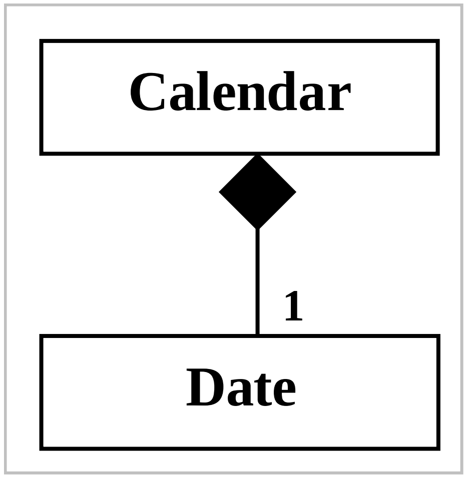
- Represents a "has-a" relationship between two classes
- A class *C* is an aggregate if it has an attribute of type *T* and *T* is NOT a primitive type OR a String
- Attribute *T* is called the "aggregated part", "part", "aggregated component", or "component"
- UML diagram (e.g., Investment **has a** Stock):



Composition

- Aggregate and aggregate part are created together (and reclaimed by the GC together)
- Client holds no reference to aggregate part
- UML diagram (e.g., CreditCard has two Dates):





Aggregation-Composition Distinction

- Camera and film (text p. 293)
- Computer and monitor
  - Desktop:
    - Aggregation: computer and monitor can be purchased/replaced separately
  - Laptop:
    - Composition: computer and monitor form a cohesive unit; cannot be separated and still considered a laptop

Constructors

- For aggregates:
  - Client instantiates attribute object (that will serve as aggregate part) and retains reference to it
  - Client instantiates aggregate by passing aggregate part as a parameter to constructor
- For compositions:
  - Instantiating composition class also instantiates the attribute object (the "part")
  - If client passes attribute object as constructor parameter, object state is copied to a new object; this way, the client still does not hold any reference to the "part"

Accessors

- Format: get*NameOfAttribute*()
- For aggregates:
  - Returns reference to the aggregate part
- For compositions:
  - Remember composition rule (from slide 3): "Client holds no reference to aggregate part"
  - Creates a copy/clone of the aggregate part and returns a reference to the copy/clone

Copy or Reference?

- Call accessor twice, save returned references
- Compare the references’ memory addresses using the == relational operator
- If true → aggregation returned references
- If false → composition returned copies

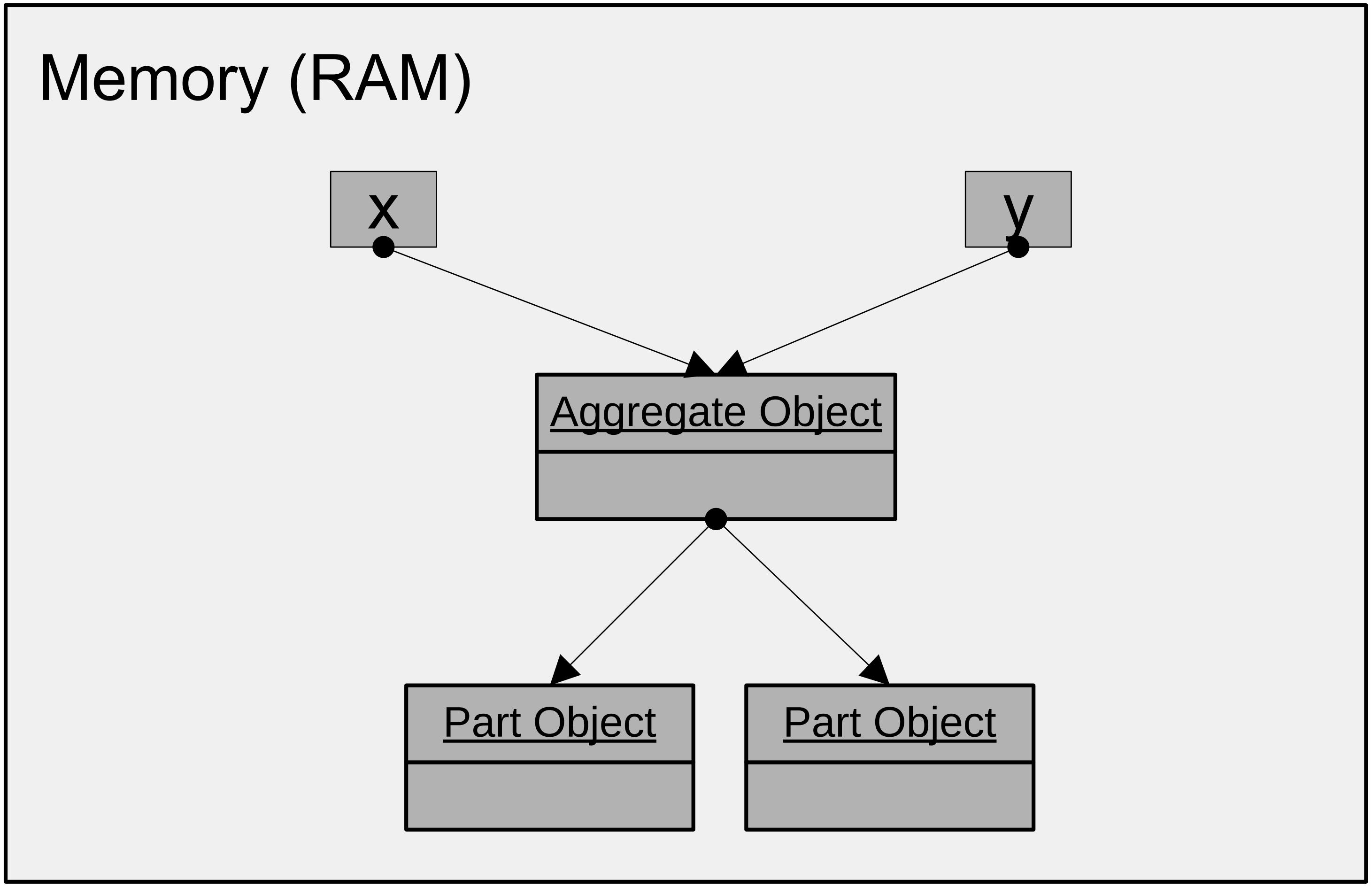
Mutators

- Format: set*NameOfAttribute*(*newInstance*)
- Changes where the attribute's reference points
  - Changes to the attribute's state handled by mutators in the attribute's class
- For aggregates:
  - Reference to the aggregate part is changed to point to the passed instance (i.e., the method parameter)
- For compositions:
  - None (Aggregate part's reference cannot change!)

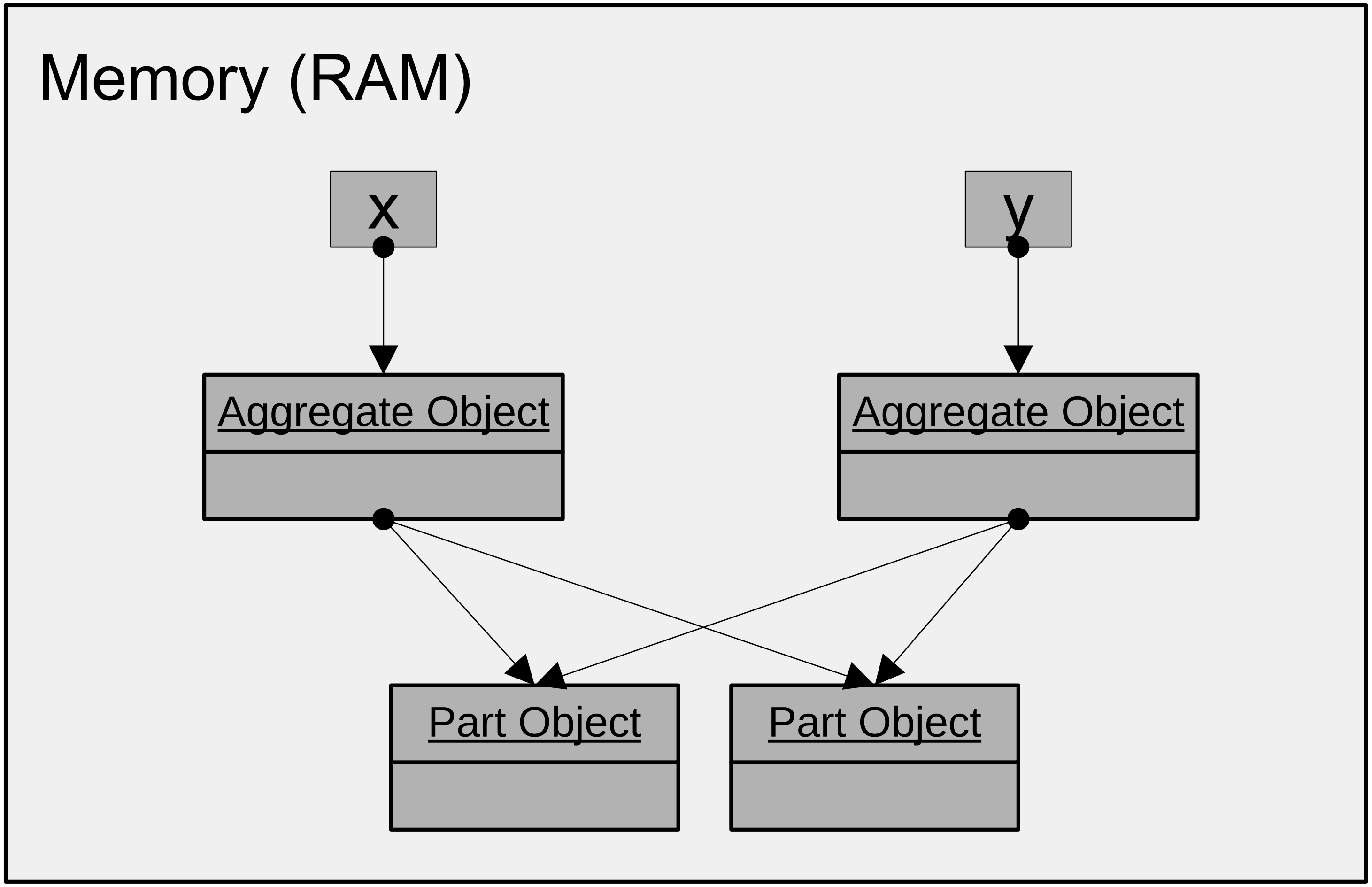
Aggregate Cloning

- Aggregate attributes could also be aggregates
- When making a copy of an aggregate, how should the attributes be copied?
  - Aliasing: copy references only
  - Shallow copy: create copies of attribute objects
  - Deep copy: create copies of attribute objects, and create copies of the copies' attribute objects

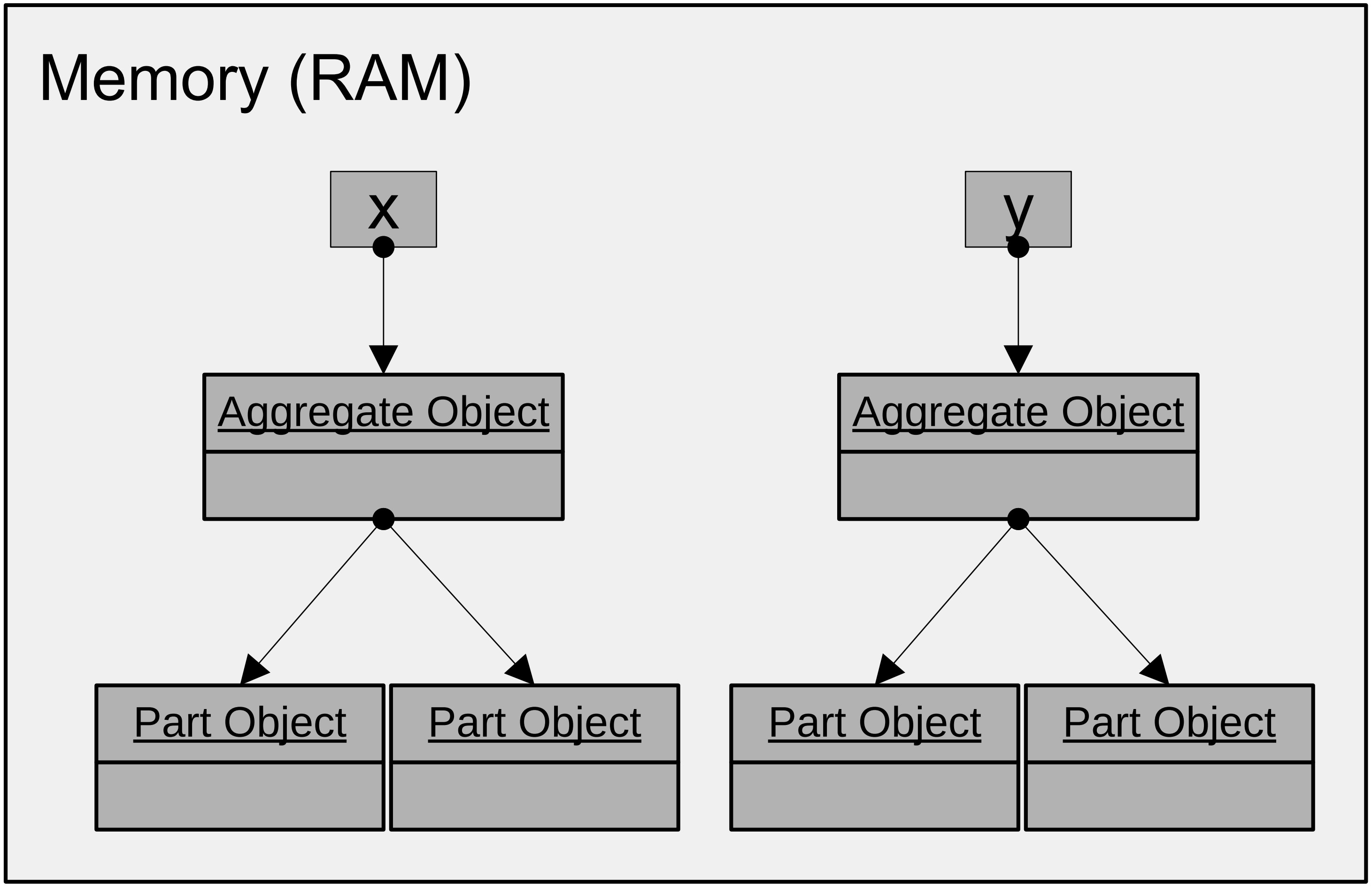
Aliasing



Shallow Copy



Deep Copy

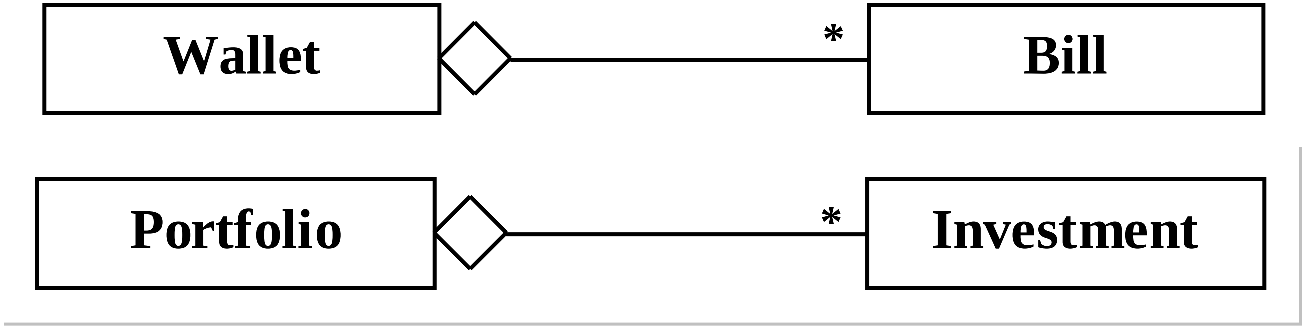

Memory (RAM)
x
y
Aggregate Object
Aggregate Object
Part Object
Part Object
Part Object
Part Object

Collections

Some examples and/or figures were borrowed (with permission)
from slides prepared by Prof. H. Roumani

What is a Collection

- Aggregate class with variable multiplicity



- Each instance of the aggregate class is called an element in the collection
  - Wallet is a collection of Bill elements
  - Portfolio is a collection of Investment elements
- Chapter 8: collections in type.lib
- Chapter 10: Java's collection framework

Creation

- Constructor creates an empty collection
- Collection capacity can be static (i.e., fixed) or dynamic (i.e., able to change)
- Fixed capacity
  - Easy for Java (and implementer) to manage memory
  - Collection can become full during run-time
- Dynamic capacity
  - Collection capacity can grow (or shrink) during run-time to efficiently accommodate various number of elements

Adding Elements

- Method typically called add(*element*)
- Two possible problems can occur:
  - Collection is full (only with fixed capacity collections)
  - Element already present (some collections require all elements to be unique)
- Return type:
  - boolean: if addition can fail (due to full capacity or duplicate element)
  - void: if no possible problems

Indexed Traversal

- Possible if elements are indexed (0..size-1)
- Use method size() to determine max index
- Use method get(*index*), getElement(*index*), etc. to access element at given index
- Access elements “randomly”

Chained Traversal

- Elements accessible only in some pre-defined order
- Use method getFirst() to get the “first” element
- Use method getNext() to access subsequent elements in the collection
- End of collection → getNext() returns null
- Can call getFirst() to return to the first element

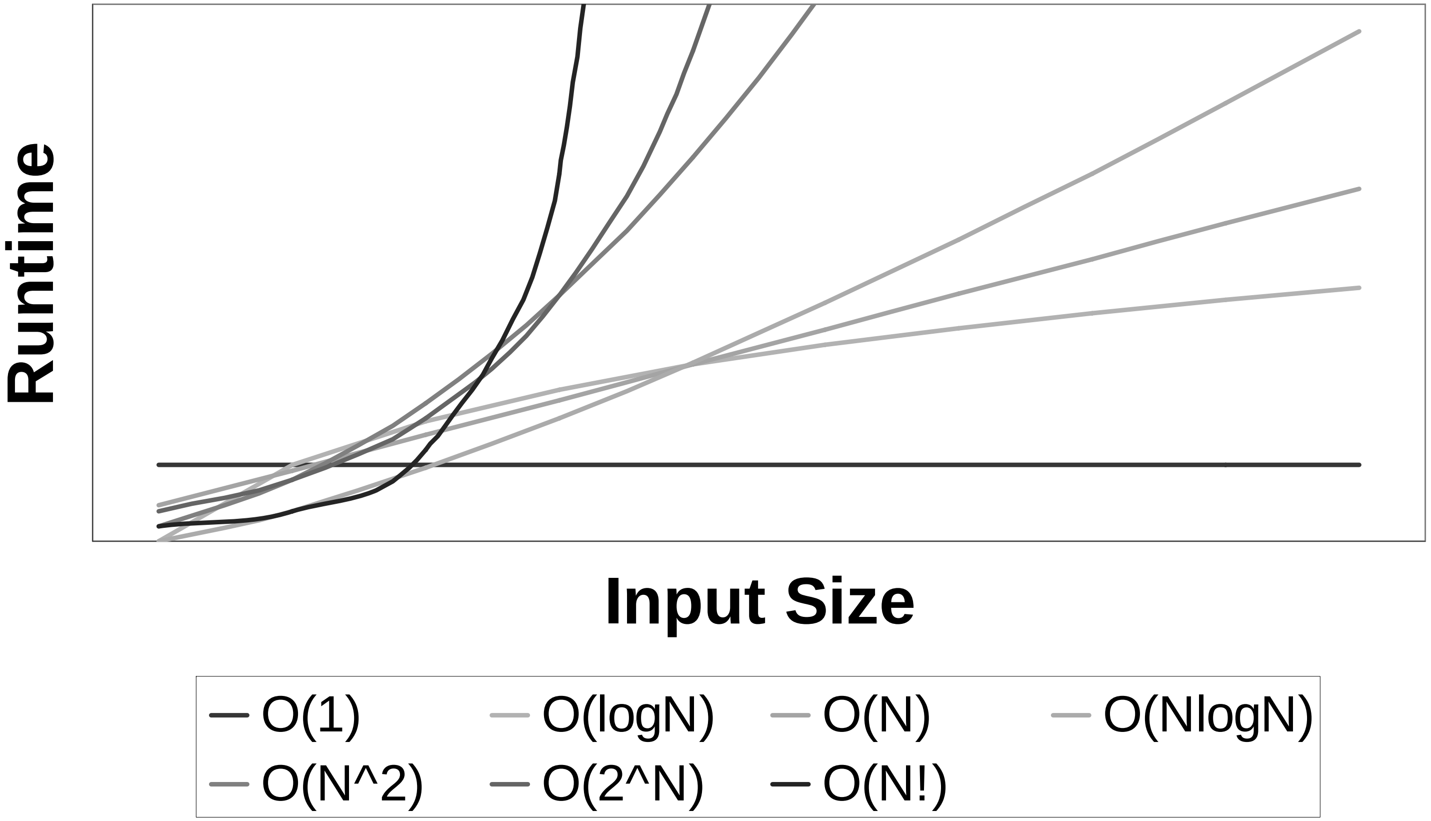
Searching

- Common task: search for element(s) in a collection matching a target value
- Time to search for an element can vary based on:
  - Number of elements (determined by user)
  - Search technique (determined by programmer)
- How to choose a search algorithm?
- How does the search time grow with respect to increases in number of elements?

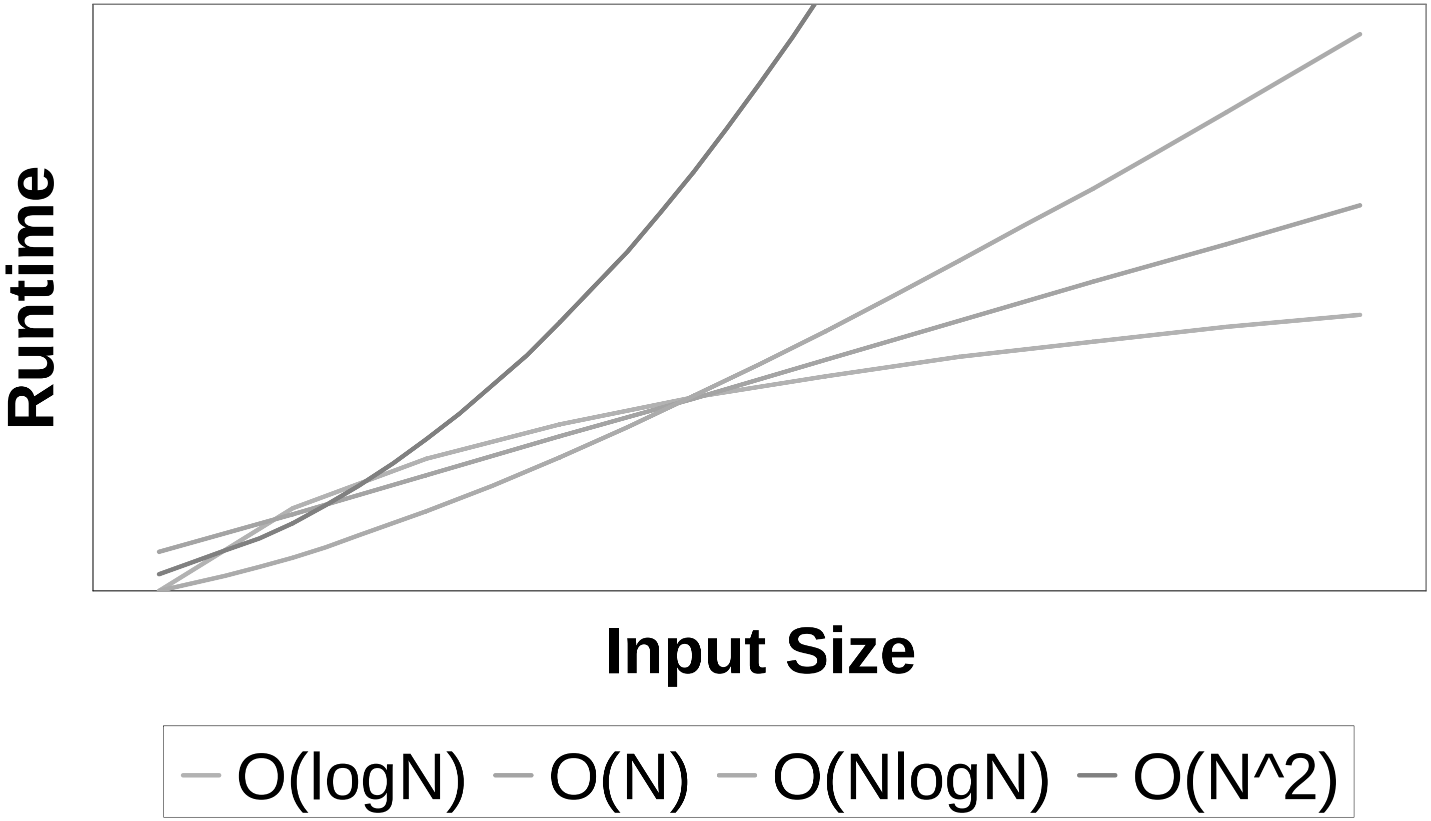
Runtime Complexity

- In the worst-case condition, how does the runtime of an algorithm grow with respect to the size of input (N)?
- Expressed in Big-O notation
  - O(1): the runtime varies by a constant factor
  - O(N): the runtime grows proportionally with N
  - O(2^N): the runtime grows exponentially with N
  - …

Runtime Complexity



Runtime Complexity



Search Complexity

- Task: search for all matching elements
- Elements in no order
  - Requires linear search (i.e., check each element)
  - Best case: O(N)
- Elements in sorted order
  - Can use binary search
    - Pick the middle element
    - Target element bigger or smaller than middle element?
    - If bigger look at “top” half; if smaller look at “bottom” half
  - Best case: O(logN)
- Element values are indexed
  - Access any element directly
  - Best case: O(1)