Penn Engineering
ONLINE LEARNING

CIT 594 Data Structures and Software Design

Module 7: Tree Variations

Module Learning Objectives

- Understand the structure of a Max-Heap.
- Understand how a Max-Heap can be implemented in Java using an array representation.
- Understand the structure of a Trie.

Module Glossary

- **Max-Heap:** a binary tree data structure in which the value of any node is always greater than that of its children, ensuring that the largest value node is the root of the tree.
- **Array representation:** a way of representing the values in a tree using an array instead of nodes and links, such that the locations of a node's parent and children can be determined based on the node's index.
- **PriorityQueue:** the Java Collections Framework implementation of a Min-Heap.
- **Trie:** a tree-like data structure that is efficient at storing strings, in which each node holds a single character, and the nodes that are children of a node represent characters that follow that character in strings/values in the collection.

Key Concepts & Examples

A **Max-Heap** is a complete binary tree

- Each level of the tree is filled before the next
- Each level of the tree is filled in from left to right with no gaps

It also satisfies the heap property: each node has a greater value than its children, and thus all of its descendants

Finding the largest value in the Max-Heap can be done in O(1) time.

Adding a value and removing the largest value can be done in O(log n) time.

A Max-Heap can be implemented using array representation:

- A heap with max height h will have an array size of 2^h-1
- The root of the tree is in array index 0
- If a node is at index k…
    - Its left child is at index 2k+1
    - Its right child is at index 2k+2
    - Its parent is at index (k-1)/2

When a value is added to a Max-Heap, it is always placed in the rightmost available spot of the deepest level of the tree, i.e. in the first open spot of the array representation, and it “bubbles up” until it reaches a point at which the heap property is satisfied.

When the largest value is removed from a Max-Heap, it is replaced with the rightmost value of the deepest level of the tree, i.e. in the last spot of the array representation, and it “bubbles down” until it reaches a point at which the heap property is satisfied.

A **Trie** is a tree-based data structure for storing strings in order to support fast pattern matching. It satisfies the following properties:

1. Each node, except the root is labeled with a character from the alphabet
2. The children of an internal node have distinct labels
3. Each leaf node is associated with a string, such that the concatenation of the labels of the nodes on the path from root to leaf yields that particular string

Strings can be added or removed by starting at the root and following paths to affected nodes

Special nodes can be used to indicate the termination of strings in case one is a prefix of others

All operations are O(m) where m is the length of the string, assuming a node stores its child nodes in an O(1) data structure.

Penn Engineering
ONLINE LEARNING

CIT 594 Data Structures and Software Design

Module 8: Graphs

Module Learning Objectives

- Understand the structure of a graph.
- Implement a graph in Java.
- Apply Breadth-First and Depth-First Search algorithms on a graph.
- Evaluate the Big-Oh complexity of graph operations.

Module Glossary

- **Adjacency List:** One of the common ways of representing a graph in code. A mapping of nodes to a list of the nodes with which they share an edge.
- **Adjacency Matrix:** One of the common ways of representing a graph in code. A 2D array in which the value of [x,y] represents the weight of the edge from x to y. In an unweighted graph, it will typically be simply a 1 to indicate the presence of an edge.
- **Breadth First Search (BFS):** A graph traversal algorithm where all immediate neighbors are visited before any more distant neighbors (e.g. neighbors of neighbors) are visited. The order in which nodes/vertices are visited is typically maintained using a Queue in BFS implementations.
- **Connected Component:** A subgraph (part of a larger graph) in which each node is reachable from any other node in the same subgraph.
- **Connected Graph:** A graph in which all nodes are reachable from all other nodes (either directly or through some path in the graph). The opposite of a disconnected graph, in which one or more nodes are not reachable from every other node.
- **Cycle:** A path where the start and end node are the same.
- **Cyclic Graph:** A graph that contains at least one cycle. The opposite of an acyclic graph, which contains no cycles.
- **Degree of a Node:** The number of each edges that a node has. In a directed graph these are split into in-degree (the number of edges pointing towards the node) and out-degree (the number of edges pointing away from the node).
- **Depth First Search (DFS):** A graph traversal algorithm in which all univisited neighbors of a node N are recursively visited before moving on to the next unvisited node in the graph and applying the same strategy.
- **Directed Graph:** A graph in which the edges between nodes encode a specific direction to the relationship between the two nodes. The opposite of an undirected graph.
- **Edge:** An object representing a link between two vertices (nodes) in a graph. These edges can be weighted (have some value attached to the link between two nodes) or unweighted. They can also be directed (point from one node to another) or undirected. Attributes such as weight and direction are typically stored as fields of the Edge object.

- **Edge Set:** One of the common ways of representing a graph in code. A set of objects representing edges in the graph.

- **Graph:** A data structure that consists of two main types of components, nodes (vertices) and edges connecting them. In a graph, each node can have an arbitrary number of edges to other nodes.
- **In-Edge of a Node:** In a directed graph, this is an edge that points towards this node from another node.
- **Neighbor:** In a graph, a neighbor to a node A is any node that shares an edge with that node A. In a directed graph, a node is only considered a neighbor of node A if it is the destination node of one of the out-edges of node A.
- **Node/Vertex:** One of the components of a graph. They generally contain some fields to store information about the elements contained in the graph.
- **Out-Edge of a Node:** In a directed graph, this is an edge that points away from this node to another node.
- **Path: I**n a graph, a path is a list of nodes from a start node to an end node in which each successive pair of nodes is connected by an edge.
- **Weighted Graph:** A graph in which all edges have an associated numerical value (weight). The opposite of an unweighted graph, in which all edges are assumed to have the same (or no) weight.

Key Concepts & Examples

Graphs are useful for representing relationships and distances between elements.

- A graph is composed of **nodes** and **edges**
- **Nodes** typically stores information about the objects contained in a graph. Each node can have an arbitrary number of edges to other nodes.
- **Edge:** an object representing a relationship between two nodes in a graph. These can be weighted or unweighted, and directed or undirected
- A **path** is a list of nodes in which each successive pair of nodes is connected by an edge
- A **cycle** is a path where the start and end node are the same
- A **connected graph** is a graph in which all nodes are reachable from all other nodes

Representing Graphs:

- Graphs can be represented in various ways including:
    - Adjacency Matrix: 2-dimensional array in which value of [x,y] represents the weight of the edge from x to y.
    - Edge Set: A set of objects representing the edges in a graph.
    - Adjacency List: A mapping of nodes to a lists of other nodes with which they share an edge.

Traversing Graphs:

- Breadth First Search (BFS) is a systematic way of traversing a graph in which we:
    - Start with a source node
    - Then visit all its neighbors
    - Then visit all their neighbors
    - And so on, until the target node is found

- Depth First Search (DFS)
    - considers one of the starting node's neighbors, then one of its neighbors, then one of its neighbors…
    - And then considers other neighbors only when it can't go any "deeper"

Penn Engineering
ONLINE LEARNING

CIT 594 Data Structures and Software Design

Module #9: Introduction to Software Design

Module Learning Objectives

- Identify the aspects of software quality.
- Convert a plain-English description of requirements into a domain model.
- Represent a model using a UML class diagram.
- Convert a UML class diagram into a Java implementation.
- Identify the aspects of internal software quality.
- Understand how design principles relate to software quality.
- Apply design principles in order to create high quality software.

Module Glossary

- **Abstraction:** This refers to the notion that components (classes) should be able to use other components with minimal knowledge of the details of their implementation.
- **Aggregation**: A relationship between two classes that indicates that one class is part of the other, but can exist independently on its own.
- **Analyzability**: An aspect of software internal quality that refers to the ease with which a programmer can read, understand, and reason about code.
- **Association**: One of the possible relationships in a UML diagram (denoted by a solid line). It indicates that the two classes it connects go together in some way.
- **Changeability**: An aspect of software internal quality that refers to the ease with which a programmer can change codes.
- **Class Diagram**: A visual representation of classes, their attributes and operations, and their relationships.
- **Cohesion**: The notion of how well parts of a class (such as attributes and methods) "go together", and whether it makes sense for them to be grouped together.
- **Composition**: A relationship between two classes that indicates that one class is part of the other, but cannot exist without the class to which it belongs.
- **Controllability**: An aspect of testability that refers to the ability to put code into the state you want to test.
- **Dependency**: One of the possible relationships in a UML diagram (denoted by a dashed line). It indicates that a class A uses the attributes/operations of another class B, but B is not a part of the class A.
- **Design**: In a software development process model, design is typically used to refer to the process of converting the requirements of the system to the implementation (e.g., plain English to Java or Pseudocode).
- **Domain Modeling**: An activity that occurs early in the design stage, where the concepts (classes) that are necessary to represent the program/system are identified. In this process, the concepts' properties, behaviors, and relationships are also identified.

- **External Quality**: This refers to the quality of the software as it runs on hardware in some execution environment (i.e., speed, memory, security, etc.).
- **Functional Independence**: This refers to the notion that components (classes/modules) should have minimal dependence on other components.
- **Generalization**: A relationship between two Objects that indicates that one Object is a specialization of another. (Example: A Person is a generalization of Professor, and a Professor may have additional attributes/operations).
- **Grammatical Parsing**: An approach in which roles in the design are based on the parts of speech used to specify the system in the English description. For example, Nouns tend to be translated to classes or attributes, Adjective to subclasses or attributes, and Verbs to operations or relationships.
- **Has-A Relationship**: One of the possible relationships between classes in a UML diagram. There are two types of such a relationship which are: aggregation (denoted by a line ending with an empty diamond-shaped arrowhead), and composition (denoted by a line ending with a filled diamond-shaped arrowhead).
- **Internal Quality**: This refers to the quality of the code as text that humans need to read and write (i.e., analyzability, changeability, stability, testability, and reusability).
- **Is-A Relationship**: One of the possible relationships between classes in a UML diagram. There are two types of such a relationship which are: generalization (denoted by a solid line ending with a triangular arrowhead), and realization (denoted by a dashed line ending with a triangular arrowhead).
- **Modularity**: A concept in software quality that refers to the idea that each module (or class) is only responsible for a single part of the functionality of the overall system.
- **Monolith**: This refers to a software architecture in which a single piece of code (a single main function, for example) is responsible for the functionality of the entire program.
- **Observability**: An aspect of testability that refers to the ability to examine the state of the code you're testing.
- **Realization**: A relationship between two Objects that indicates that one Object is an actualization/realization of another. An Object A that is realized by another Object B is generally an abstract concept, meaning that Object A has no notion of existence without being a specific type of object Object B (Example: Dog, Cat, and Human are all realizations of the abstract type Animal).
- **Reusability**: An aspect of software internal quality that refers to the ease with which code can be reused in other applications.
- **Software Architecture**: The process of dividing a software system into subsystems that address major parts of functionality.
- **Software Development Life Cycle (SDLC):** A set of 5 stages that are typically part of creating a software product. These stages are Requirements Elicitation, Design, Implementation, Testing, and Maintenance.
- **Stability**: An aspect of software internal quality that refers to the extent to which the code is tolerant of changes to other parts of the code.
- **Testability**: An aspect of software internal quality that refers to the ease with which code can be tested.

- **Three-Tier Architecture**: An example of software architecture in which the system is split up into three subsystems. These tiers are the Data, Logic, and Presentation tiers.
- **Unified Modeling Language (UML):** A language/notation used for modeling and visualizing software design. UML has been an industry standard since the late 90s.

Key Concepts & Examples

Process Models typically define

- The relative amount of time spent in each stage of the SDLC
- The roles of individuals and organizations
- The input and output for each stage
- The iteration between stages

Domain Modeling is an activity that occurs early in the design stage, where the concepts (classes) that are necessary to represent the program/system are identified. In this process, the concepts' properties, behaviors, and relationships are also identified. This is typically achieved using

- **Grammatical Parsing** where roles in the design are assigned based on the parts of speech used to specify the system in the English description. Typically, we translate them using
    - Nouns : classes or attributes
    - Adjective :subclasses or attributes
    - Verbs : operations or relationships

UML Class Diagrams are used to present a visual representation of classes, their attributes and operations, and their relationships. These diagrams can then be easily translated to working code.

There are 4 main types of relationships between classes.

- "Is-a" relationships
    - **Generalization**
    - **Realization**
- "Has-a" or "contains" relationships
    - **Composition**
    - **Aggregation**
- "Uses" relationships
    - **Dependency**
- "Association" relationships
    - **Association**

Relationships between classes can also have an associated **multiplicity** which indicates how many instances of one class A can be related to another class B and vice versa.

- *N* : denotes a multiplicity of exactly *N* objects/instances

    - * : denotes a multiplicity of zero or more objects/instances
- *M..N* : denotes a multiplicity with a minimum of *M* and a maximum of *N* objects.

Software Internal Quality refers to the quality of the code as text that humans need to read and write, aspects of it include

- **Analyzability** which refers to the readability and understandability of code. This is necessary so that a programmer can easily
    - Identify and fix defects
    - Add /improve functionality
    - Understand how an algorithm/solution is implemented
- **Changeability** which refers to the ease with which code can be changed. Highly changeable code allows a programmer to easily
    - Fix defects
    - Add/improve functionality
    - Incorporate new code
    - Improve other aspects of software quality
- **Reusability** which refers to the ease with which code can be reused in other applications. Reusable code allows a programmer to
    - Reduce time spent developing new code
    - Reduce the likelihood that new code will contain defects
- **Stability** which refers to the idea that modifying one part of the code should have a limited effect on others. High stability allows a programmer to
    - Make changes in one part of the code without being required to make changes in other parts. This has the added benefit of reducing the time/effort required to change code.
- **Testability** which refers to the controllability and observability of code. High testability allows a programmer to easily
    - Create test cases
    - Identify and fix defects
        - Increase confidence that software is working correctly

Software Architecture refers to the process of dividing a software system into a subsystems that address major parts of functionality.

- Each subsystem should be able to do its work with minimal dependency on other subsystems and minimal knowledge of their implementation.

Three-Tier Architecture (TTA) is when the system is split up into three subsystems.

- **Data:** The bottom tier of TTA, it contains all the code that handles functionality related to storing and retrieving data. It doesn't interact with the user or do any processing of the data, it just makes it possible for the rest of the program to access whatever data it needs to use.
- **Logic:** The middle tier of TTA, this is where the program performs calculations, makes decisions, etc. This part of the code doesn't interact with the user either, and it doesn't do anything with storing and retrieving data.
- **Presentation:** The top tier of TTA and is also known as the user interface tier. This is where all user interaction occurs, no data processing or storage is handled in this tier.

Modularity is one of three important design concepts that help achieve high internal quality. It refers to the notion that each module (or class) is only responsible for a single part of the functionality of the overall system.

- By splitting up the functionality into separate modules, we make the code easier to read and understand, easier to test and debug, easier to reuse, and most of all, easier to change.

Function Independence is one of three important design concepts that help achieve high internal quality. It refers to the notion that modules should be able to perform their own tasks with minimal dependence on other modules.

- By my allowing a module to perform its tasks with minimal dependence on other modules, we make it easier to understand, test, and reuse on its own, but most of all, it makes the class easier to change without having to change other modules.

Functional Independence is achieved by

- Minimizing the complexity of interfaces between modules
- Minimizing the control that one module has over another

Abstraction is one of three important design concepts that help achieve high internal quality. It refers to the notion that a module should be able to use other modules with minimal knowledge of the details of their implementation. I.e. a module should only care what another module does if it will be using it, and not how it is done.

- This mostly helps with improving stability, as changes to the internal details of other modules should have no effect on the module using them, since it never depended on those details anyway.

Penn Engineering
ONLINE LEARNING

CIT 594 Data Structures and Software Design

Module 10: Software Design Patterns

Module Learning Objectives

- Understand the structure of common software design patterns in Java.
- Apply software design patterns to solve problems in Java.

Module Glossary

- **Design Pattern:** a general reusable solution to a commonly occurring problem; a description or template for how to solve a problem that can be used in many different situations.
- **Creational Patterns:** design patterns that seek to separate the code that creates an object from the code that uses it.
- **Structural Patterns:** design patterns that address problems such as:
    - how do we assemble the constituent parts of an object?
    - how do we define the relationships between classes?
    - how do we combine objects into larger structures?
- **Behavioral Patterns:** design patterns that address problems such as:
    - how do objects communicate?
    - how are responsibilities assigned to different objects?

Key Concepts & Examples

Singleton Pattern: Creational pattern that ensures that there is only one instance of a class, and that it can be accessed easily

- make the class' constructor private
- expose a public static method that returns the singleton instance

Static Factory Method: Creational pattern that provides a static method to create instances of a class

- other classes can call static factory methods as an alternative or in addition to calling constructors
- unlike constructors, a static factory method can return an object of any subtype of their return type

Bridge Pattern: Structural pattern that maintains separate inheritance hierarchies of concepts (abstractions) and things that uses them (implementors) and "bridges" them using aggregation so that Client only needs to work with abstraction

Strategy Pattern: Behavioral pattern in which we create classes to specify certain “strategies” that can be used as part of a larger algorithm

Observer Pattern: Behavioral pattern in which a Subject has a set of Observers that can be notified when an event occurs. Observers can be added and removed as the program runs.

Penn Engineering
ONLINE LEARNING

CIT 594 Data Structures and Software Design

Module 11: Writing Good Code

Module Learning Objectives

- Determine the ways in which the readability and understandability of code can be improved.
- Apply techniques to improve the understandability of code.

Module Glossary

- **Code Smells:** A term used to describe indications of bad design or bad internal quality in code.
- **Dictionary Words:** Words that occur in everyday language. These are good to use as variable names, as they tend to be easier to read/recognize, in comparison to words that are made up.
- **Duplicate Code:** One of the most common code smells that occurs when we use the same (or very similar) code in multiple places by copy and pasting it in multiple locations.
- **Extract Method:** A refactoring pattern used to address duplicate code in which we pull out, or extract, the code that's duplicated and put it in a new method, and then invoke that method wherever we had the original duplicate code.
- **Extract Superclass:** A refactoring pattern where we create a new common superclass that contains a method that can be used in multiple subclasses
- **Readability:** In code, readability is the ease with which the reader can identify and differentiate tokens and their syntactic purpose.
- **Refactoring:** An activity by which we modify code but don't change its functionality. Generally, the purpose of refactoring is to improve readability/understandability, or reduce the number of lines of code.
- **Syntactic Purpose:** For a token this is defined as what the function of the token is in that particular position in the statement of code.
- **Understandability:** In code, understandability is the ease with which a reader can identify the semantic meaning of code (i.e. what the code does).

Key Concepts & Examples

Code readability is the ease with which the reader can identify and differentiate tokens and their syntactic purpose. It is important because people need to read code in order to:

- Fix bugs
- Add features
- Understand how a problem was solved
- Improve quality: efficiency, security, etc.

Code readability is affected by the following factors:

        - Use of whitespace
    - Identifier length
    - Use of dictionary words
    - Variation among identifiers

Code understandability is the ease with which a reader can identify the semantic meaning of code.

To improve code understandability

- Use meaningful identifier names
- Use indentation and spacing
- Use punctuation
- Use comments

Refactoring an activity by which we modify code but don't change its functionality.

Why should we refactor?

- Improve the internal quality of code
- Simplify future code changes
- Reduce the amount of code to maintain

When should we refactor?

- Constantly!
- When you see code (or are writing code) that doesn't adhere to the intended design
- As a way of understanding the code

How should we refactor?

- Carefully!
- Make small changes before big ones
- Be careful about:
    - Introducing bugs
    - Breaking "published" interfaces
    - Unnecessarily decreasing external quality

What should we refactor?

- Code smells: indication of bad design
- Code that is hard to read or hard to understand
- Code that does not adhere to conventions
- Code that is unnecessary or in the wrong place

Duplicate Code is the most common code smell. Two ways of dealing with duplicate code include:

- Extract Method: Create a new method that can be used in multiple places
    - Assumes duplicate code is in the same class
- Extract Superclass: Create a new common superclass that contains a method that can be used in multiple subclasses

Penn Engineering
ONLINE LEARNING

CIT 594 Data Structures and Software Design

Module 12: Software Efficiency

Module Learning Objectives

- Understand the tradeoffs between efficiency and other aspects of quality.
- Accurately evaluate the execution time of a piece of software.
- Apply software programming techniques to improve the efficiency of code.

Module Glossary

- **Wall-Clock Time:** elapsed "real" time between two events, as would be measured by a wall clock, stopwatch, etc. This can lead to inaccuracies when measuring software execution time because of other factors, e.g. operations being performed by the Java Virtual Machine, operating system, hardware, etc.
- **Lazy evaluation:** programming technique in which we evaluate expressions only when we know we'll need the results
- **Lazy initialization:** programming technique in which we only create objects when we know we're going to need them
- **Memoization:** programming technique in which we keep a Map of inputs (arguments) to outputs (results) and return the result from the Map if the same input is seen more than once. This will lead to more memory utilization but may greatly reduce execution time.
- **Method Inlining:** micro-optimization technique in which we replace method call with method body
- **Loop Unrolling:** micro-optimization technique in which we replace for-loop with multiple instances of body
- **Constant Folding:** micro-optimization technique in which we replace literal numerical expression with result
- **Strength Reduction:** micro-optimization technique in which we replace complex operations with simpler ones

Key Concepts & Examples

Software quality is about **tradeoffs**: improving one aspect of external quality may harm other aspects of external quality and/or internal quality. If we make our code faster, we may also make it:

- harder to understand
- harder to change
- less likely to provide the correct functionality
- less secure

        - likely to use more memory
    - more time-consuming to develop

Software efficiency rules of thumb:

- Always use the right data structure.
- Make the common case fast.
- Don't do work unnecessarily.
- Be careful about accidentally harming other aspects of quality.

Execution time can be measured by various mechanisms:

- Use Java System methods to measure execution time of small pieces of code.
- Use UNIX "time" command to measure execution time of entire program.
- Use profiling tools for more detailed information.

To avoid doing unnecessary work:

- Don't check conditions unnecessarily
- Don't call methods unnecessarily
- Don't redo work you've already done
- Use the Java API when possible
- Use lazy evaluation, lazy initialization, and memoization
- Take advantage of short-circuit boolean operators
- Use micro-optimizations sparingly, especially if they may harm understandability

Penn Engineering
ONLINE LEARNING

CIT 594 Data Structures and Software Design

Module 13: Concurrency in Java

Module Learning Objectives

- Implement concurrent code using Java threads.
- Understand the behavior of concurrent code and how it can lead to race conditions.
- Apply synchronization in order to address race conditions.

Module Glossary

- **Concurrency:** A feature of some programming languages that allows a program to execute different parts of the code at the same time. In Java, this is achieved using Threads.
- **Critical Section:** A piece of code where a race condition could occur.
- **Heap:** The area of memory where objects are stored. It is also where static and global variables are stored.
- **Mutual Exclusion:** The process of ensuring that critical sections of code are only ever executed by a single thread at a time (i.e. making sure that access to that critical section is mutually exclusive for different threads). This is necessary to avoid inaccurate results being caused by race conditions.
- **Race Condition:** In threading, a race condition can occur when multiple threads are trying to modify the same piece of data. This this can lead to not only non-deterministic results, but also incorrect behavior if the results of operations are lost.
- **Stack:** The area of memory where local variables and method arguments are stored. A stack also holds its own program counter, which holds the address of the next instruction to execute.
- **Synchronized Method:** One of the ways of ensuring mutual exclusion in Java. This ensures that only one thread at a time can execute this method for a given object.
- **Synchronized Blocks:** A more fine-grained approach to ensure mutual exclusion than synchronizing a whole method, by only *locking* a section of the function. This allows us to have a critical section that's smaller than an entire method, and also to have multiple critical sections that can be run simultaneously. Each synchronized block relies on an object as its lock. If an object is locked, then no other thread can enter a synchronized block that is using it. But another thread *can* enter a synchronized block that's using a different object.

Key Concepts & Examples

Some rules of thumb for **software efficiency** include:

- Always use the right data structure!
- Don't do work unnecessarily.

        - Don't allocate memory unnecessarily.
    - Do more than one thing at a time

Doing more than one thing at a time in a program is called **concurrency**. For example,

- Performing two parts of a calculation simultaneously
- Reading data from one source while writing to another
- Blocking/waiting for some input while performing some calculation

To achieve concurrency, we use **threads**.

- **Threads** are separate lines of execution in the same process (running program)
- Each thread has its own
    - stack (local variables, method arguments)
    - program counter (address of next instruction to be executed)
- Threads in the same process share:
    - heap (space in memory where objects are stored)
    - static variables

In Java,

- Thread classes can extend the `java.lang.Thread` class or implement the `java.lang.Runnable` interface
- Represents the starting point of a separate line of execution in the running program
    - `start()` : launch a new line of execution
    - `run()` : entry point to new line of execution

Using threads in Java:

- We use threads to solve "embarrassingly parallel" processes.
- Typically, N threads each do 1/Nth of the work, and then the sub-solutions are combined to create the solution
- Care must be taken to wait for all the threads to finish!

Thread non-determinism

- Multiple threads can be used to perform different operations simultaneously, even on the same object
- This can lead to **non-deterministic** results because the ordering of the execution of the threads is not guaranteed

Thread race conditions

- When multiple threads operate on the same piece of data, this can lead to not only non-deterministic results, but also incorrect behavior if the results of operations are lost. These are called **race conditions.**

Thread synchronization

- Synchronization, or **mutual exclusion**, can be used to address race conditions.
    - **Mutual exclusion**: only allow one thread at a time to execute the **critical section** where a race condition could occur
    - **Synchronized method:** only one thread at a time may execute that method (or any other synchronized methods) on that object
    - If a thread is executing a synchronized method on an object, any other thread that tries to execute a synchronized method on the same object will need to wait for the first thread to finish
- Synchronized methods ensure that only one thread at a time can enter those methods.
- Synchronized blocks provide more fine-grained access.