**CS2050 Technical Documentation Module 3**

Contents

[**Module 3: Objectives Abstract Classes, ArrayLists and Linear Data Structures** 3](#_Toc195560414)

[1.1 Abstract Class 3](#_Toc195560415)

[1.1.1 Defining Abstract Classes 3](#_Toc195560416)

[1.1.2 When to use an abstract class versus a concrete class 4](#_Toc195560417)

[1.1.3 Abstract Method 4](#_Toc195560418)

[*1.1.4* *Abstract Classes Rules* 4](#_Toc195560419)

[1.2 Interface 5](#_Toc195560420)

[*1.2.1* *What is an Interface?* 5](#_Toc195560421)

[1.2.2 Why Use an Interface 5](#_Toc195560422)

[1.2.3 Defining an Interface 6](#_Toc195560423)

[1.2.4 Using an Interface 6](#_Toc195560424)

[1.2.5 Interface methods 6](#_Toc195560425)

[1.2.6 Compare abstract classes and interfaces 7](#_Toc195560426)

[1.2.7 Interface vs. Implementation 7](#_Toc195560427)

[1.2.8 Interface Rules 7](#_Toc195560428)

[1.3 When to Use Interface vs. Abstract Class? 7](#_Toc195560429)

[1.4 Data Structures 8](#_Toc195560430)

[1.5 Collections 8](#_Toc195560431)

[1.5.1 Basics of Collections 8](#_Toc195560432)

[1.6 Generic Class or Method 9](#_Toc195560433)

[1.6.1 What is a Generic 9](#_Toc195560434)

[1.6.2 Why use Generics 9](#_Toc195560435)

[1.6.3 Understanding Generics 9](#_Toc195560436)

[1.6.4 Abstract Classes in Comparison to Generics 10](#_Toc195560437)

[1.7 List Interface 10](#_Toc195560438)

[1.8 ArrayList 10](#_Toc195560439)

[1.8.1 Compare ArrayLists with arrays 11](#_Toc195560440)

[1.8.2 Create ArrayList with Primitives 11](#_Toc195560441)

[1.8.3 Create ArrayList with objects 11](#_Toc195560442)

[1.8.4 Perform basic operations on an ArrayList: add, get, remove, set.. 12](#_Toc195560443)

[1.8.5 Casting work with abstract classes and interfaces 13](#_Toc195560444)

[1.8.6 Instanceof with abstract classes and interface 14](#_Toc195560445)

[1.9 Linear data structure: Stacks and Queues 14](#_Toc195560446)

[1.9.1 Understanding Linear Data Structures 14](#_Toc195560447)

[1.9.2 Stack Last In First Out (LIFO) 15](#_Toc195560448)

[1.9.3 Queues First In First Out (FIFO) 16](#_Toc195560449)

[1.9.4 Compare stacks and queues, when to use each data structure. 17](#_Toc195560450)

[1.10 Linked Lists 17](#_Toc195560451)

[1.10.1 Self-Referential Class 17](#_Toc195560452)

[1.10.2 Singly Linked List 18](#_Toc195560453)

[1.10.3 Operations on a Linked List: insert, delete 18](#_Toc195560454)

[1.10.4 Doubly linked 19](#_Toc195560455)

[1.10.5 Doubly Linked List Applications 20](#_Toc195560456)

[1.10.6 Operations on a Doubly Linked List: insert, delete 21](#_Toc195560457)

[1.10.7 Circular Doubly Linked Lists 22](#_Toc195560458)

[1.10.8 Singly Linked List vs Doubly Linked List 23](#_Toc195560459)

[1.10.9 Comparison: Array, ArrayList and Linked List 23](#_Toc195560460)

[**1.11** **Collection interface and Sorting** 24](#_Toc195560461)

[1.11.1 Implement sorting with Collections.sort() to sort custom objects 24](#_Toc195560462)

[1.11.2 Convert between LinkedList and ArrayList when needed for sorting. 25](#_Toc195560463)

[1.12 Maps – HashMap 25](#_Toc195560464)

[1.12.1 What is a HashMap 25](#_Toc195560465)

[1.12.2 How does a HashMap Work 26](#_Toc195560466)

[1.12.3 Common Map Methods 27](#_Toc195560467)

[1.12.4 Why / when to Use Maps 27](#_Toc195560468)

[1.12.5 Comparison: Maps vs Lists 27](#_Toc195560469)

# **Module 3: Objectives Abstract Classes, ArrayLists and Linear Data Structures**

## Abstract Class

A screen shot of a computer code

AI-generated content may be incorrect.

* Placeholder that represents a generic concept
* **Cannot be instantiated**

**A close up of a text

AI-generated content may be incorrect.**

* Must be extended to be of any use:
  + **Abstract class defines the behavior (the methods) for its subclasses**
  + Subclasses of the abstract class implement that behavior (methods)

**Why use Abstract Classes?**

* Sometimes it does not make sense to instantiate a class, for example, when the object has no meaning.

### Defining Abstract Classes

Include **abstract** keyword on class definition

A close up of text

AI-generated content may be incorrect.

### When to use an abstract class versus a concrete class

|  |  |
| --- | --- |
| Concrete Classes | Abstract Classes |
| Use concrete class when:   * + Specific enough to be instantiated   + Objects of this class make sense | Use abstract class when:   * + Not specific enough to create objects   + Some (or all) behaviors don’t make sense unless implemented by a specific subclass   + Need class for inheritance/polymorphism   + Use as a reference type (like above Animal array example) |

### Abstract Method

#### Define

* A method with **no body.** Contains **only a declaration. Must be overridden in subclass** to be of any use (*Dynamic Binding*)
* Defining Abstract Methods: Include **abstract** *keyword* on method definition:

|  |
| --- |
| Abstract method in Super class Vehicle:  A black text on a white background  AI-generated content may be incorrect. |
| Implemented in subclass ElectricCar |

#### Why use Abstract Methods

When a class that is meant to express only the interface, then ***the implementation for certain methods may not be known*** ***until subclasses are created***.

Great flexibility in code is created when we leave it to the subclasses to fill in the details.

* An abstract class has virtually no purpose unless it is extended
* The workers are the subclasses of the abstract class.
* Abstract class means class must be extended

*Abstract method means method must be overridden*

### *Abstract Classes Rules*

* When defining abstract classes
  + Must define a class abstract when there are 1 or more abstract methods in the class
  + Can define a class abstract even if it contains only concrete methods
    - When would you do this?
    - When you want to prevent user from making instances of the class but want to create a class that contains common functionality for its subclasses
  + Abstract classes typically contain abstract and non-abstract (concrete) methods.
* When defining abstract methods
  + Abstract methods cannot be defined as final or static.
  + final – means cannot override the method, so this doesn’t work since need to override.
  + static - means don’t need an object to use method, doesn’t work since need object to override.
* When extending abstract classes
  + A child of an abstract class must override (implement) all the abstract methods, otherwise it must be defined as abstract

## Interface

### *What is an Interface?*

A white background with black text

AI-generated content may be incorrect.

In general terms

* An interface is a description of how two things interact.
* A car’s owner’s manual contains the details of how to interact with the car.

In Java terms

* When a class contains 100% abstract methods, we call it an interface
* Note: starting with Java 8 interfaces can contain default methods and static methods
* For this class, we will focus on interfaces containing ONLY abstract methods

Interface takes the concept of an abstract class one step further!

* It is a pure abstract class
* It provides the form for a class but no method bodies
* It says: “This is what all classes that implement this particular interface will look like.”

### Why Use an Interface

To achieve abstraction / đạt tính trừu tượng

1. To provide full abstraction
   * Separates the what to do (interface) from the how to do it (implementation)
2. To create flexible code!
   * Coding to an interface, rather than an implementation, makes code easier to extend.
   * Coding to an interface means the code works with all the interface’s subclasses – even ones that haven’t been created yet.
3. To establish a protocol (contract) between classes
   * When a class implements an interface, it’s agreeing to implement all the methods

### Defining an Interface

* Include **interface** keyword instead of class keyword
* Do **not include a constructor** in interfaces
* You **can include constants** in interfaces

A screenshot of a computer

AI-generated content may be incorrect.

### Using an Interface

In most cases, we use interfaces the same way we use an abstract class

* Use as a data type for a variable
* **Subclass implements** instead of extends (since subclass will be providing all the implementation

A screenshot of a computer program

AI-generated content may be incorrect.

### Interface methods

* Interface methods are **abstract** so they **MUST end with semicolon**
* Interface methods are implicitly public and abstract, so **public and abstract** keywords **are not needed**. I include keywords in notes for clarity.

A screenshot of a computer code

AI-generated content may be incorrect.

### Compare abstract classes and interfaces

|  |  |  |
| --- | --- | --- |
| Fearture | Interfaces | Abstract Classes |
| Constructors | None | Invoked by subclasses |
| Variables | must be constants (public static final only) | can have all types of data |
| Methods | must have only a signature without implementation (public abstract methods only) | can have concrete methods and abstract methods |
| Everything else | the same | the same |

### Interface vs. Implementation

|  |  |
| --- | --- |
| Interfaces | Implementation |
| * + Describes the WHAT!   + Specifies ONLY the requests that can be made of an object     - Method names     - Parameter lists     - Return types   + To be matched with an implementation in subclass that completes the abstraction | * + Describes the HOW!   + The particular details necessary to make a workable object |

### Interface Rules

Interfaces can contain only:

* Constants
* Abstract methods
* Default and static methods as well but we will not be using these in this class

By default

* interface variables are public static final (this means they are constants)
* interface methods are public abstract

Interface variables must be initialized at time of declaration – otherwise compiler error

More rules

* An interface can extend other interfaces
* An interface cannot extend a class or implement an interface
* A class can extend one class and implement any number of interfaces

## When to Use Interface vs. Abstract Class?

|  |  |
| --- | --- |
| Abstract class | Interface |
| * Strong IS-A relationship should be modeled using classes * If related classes need to share code – put code into abstract class that will be extended | * **Weak IS-A relationship** can be modeled using interfaces (kind-of relationships) * When all methods will be implemented by classes that implement the interface * To circumvent single inheritance restriction if multiple inheritance is desired   + Design one class as a regular or abstract superclass   + Design all others classes as interfaces * When it’s possible to create your superclass without any method definition or member variables prefer interfaces to abstract classes.   + Start with interface   + Only if need to implement a method then class turn into abstract class |

## Data Structures

**What is a Data Structure?**

* Any representation of data and its associated operations.
* An organization or structuring for a collection of data items.
  + A sorted list of integer values stored in an **array** is a data structure.
  + Text editing operations stored on a **stack** is another data structure.
  + Decisions to make based on a current state stored in a **tree.**

**Why Data Structures**

* Representing information is fundamental in computer science.
* Development of commercial applications involves the development of complex and interacting structures.
* Computer programs need to store, retrieve, and manipulate data – as fast as possible!

|  |  |
| --- | --- |
| Data Structures  Learn correct data structure to use to store the data | Algorithms  Learn what algorithms to use to properly manipulate(điều khiển) the data structure |

* Data structures and algorithms are the heart of computer science!
* We will learn how to structure information to support efficient processing.

## Collections

A diagram of a programming language

AI-generated content may be incorrect.

### Basics of Collections

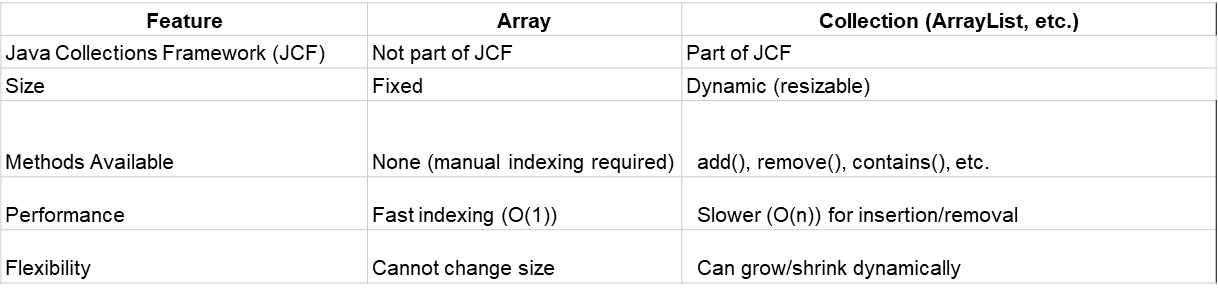
A collection in Java is a data structure used to store and manage multiple elements efficiently. Java provides the Java Collections Framework (JCF), which includes various types of collections like ArrayLists, Sets, and Maps.

***Use an Array when the number of elements is known and won’t change.***

* Arrays are lower-level structures → They store elements contiguously (liên quan/lân cận) in memory.

***Use an ArrayList when dynamic resizing is needed, and additional operations like adding/removing elements are required***.

* Collections provide additional functionality → Like resizing, sorting, and searching methods.
* Collections use interfaces (List, Set, Map) → Arrays do not implement these.



## Generic Class or Method

### What is a Generic

* A class or method defined with generic types that the compiler can replace with concrete types.

Generic Type

* Also called a parameterized data type.
* A class designed so that it stores, operates on, and manages objects whose type is not specified until the class is instantiated.

### Why use Generics

* Stronger type checking at compile time
  + If you try to use a class or method with an incompatible object, a compiler error occurs.
* Supports code reuse
  + If the implementation is identical except for the base type of the object, a generic implementation can be used.
* Improves software reliability and readability

### Understanding Generics

* Suppose we need a Box that can hold any type, so we define a class called Box to hold Object references.
* Because Box is defined to hold Object references, this now means Box can hold any object.
* But there is an issue with this approach of using Object as data type:
  + Because any type can be added to the box, we have no control over the types of elements added to the box and now must cast elements to their proper type when removed.
* Better approach
  + Instead, define a class that is based on a generic type.
  + Data type is referred to generically within the class.
  + The specific data type is specified only when the class is instantiated.

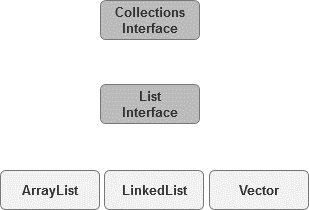
### Abstract Classes in Comparison to Generics

|  |  |  |
| --- | --- | --- |
| **Concept** | **Abstract Class** | **Generics** |
| **Purpose** | Provides a base structure but leaves details for subclasses | Allows flexible type assignment without needing multiple classes |
| **Customization** | Subclasses override abstract methods | Type parameter <T> is replaced with any type |
| **Example** | abstract class Animal { abstract void makeSound(); } | class Stack<T> { private ArrayList<T> items; } |

## List Interface

Extend the collections interface.

* Collection that allows:
  + Storing elements in sequential order
  + Duplicates
  + Storing an element in a specific location
  + Accessing an element in a specific location
* Create concrete lists with the classes:
  + ArrayList
  + Linked List



Types of lists:

* Ordered
* Unordered
* Indexed

Two ways to implement a list:

* With an array
* With a linked structure

## ArrayList

Like an array - the ***ArrayList is used to store a list of objects***. Unlike an array - ArrayList size is not fixed!

* + Its size ***grows dynamically as objects are added***.

Under the covers the implementation deals with adding and removing objects

A page of a book

AI-generated content may be incorrect.

* ArrayList is known as a **generic class**
* Because it is a generic class, we use **angle brackets** when defining
* To use an ArrayList, note that must import **java.util.ArrayList**

### Compare ArrayLists with arrays

* **Declaring**: notice the difference when declaring a *regular array* vs *ArrayList*

|  |  |
| --- | --- |
| R*egular array* | *ArrayList* |
| *Regular brackets* are used | ***Angle brackets* are used and parenthesis** |

* **Array** of objects or **ArrayList** of objects – very similar!

|  |  |
| --- | --- |
| R*egular array* | *ArrayList* |
| *must specify the size of array* | ***don’t specify size*** |

### Create ArrayList with Primitives

* When you create an ArrayList, you specify a ***concrete type*** that replaces the ***generic type (E)***
* When using ArrayList with primitive types, use wrappers for the type!

A screenshot of a computer program

AI-generated content may be incorrect.

### Create ArrayList with objects

A black and blue text

AI-generated content may be incorrect.

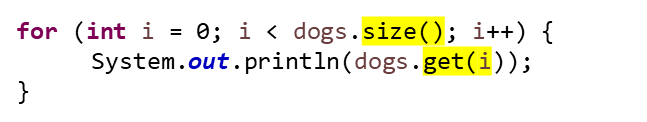
### Perform basic operations on an ArrayList: add, get, remove, set..

#### Adding to ArrayList: use the **add()** method

A computer screen shot of text

AI-generated content may be incorrect.

#### Accessing with ArrayList: Use the **size()** method to know how many to iterate. Use **get()** method to get value



#### Removing with ArrayList: Use the **remove()** method

A computer code with black text

AI-generated content may be incorrect.

#### Other methods

|  |  |
| --- | --- |
| Method | Description |
| Set() | Replace an element at a specific index |
| Clear() | Remove all elements |
| Contain() | Check if an element exists in the list |
| isEmpty() | Check if the list is empty |

#### Using for-each loop to iterate over ArrayList

***for (ElementType variable: collection)***

ElementType → The type of elements in the collection

* variable → A temporary reference to each element as we loop through the collection.
* collection → The data structure being iterated over

A screenshot of a computer code

AI-generated content may be incorrect.

#### Creating ArrayList from Array:

Write a loop

A computer code with text

AI-generated content may be incorrect.

**Or** Use methods provided by Java. Static method on Array called ***asList***

A close-up of a calendar

AI-generated content may be incorrect.

#### Creating Array from an ArrayList

Write a loop

A computer code with text

AI-generated content may be incorrect.

**Or** Use methods provided by Java. Static method on Array called ***toArray***

A close-up of a math equation

AI-generated content may be incorrect.

### Casting work with abstract classes and interfaces

#### Implicit Casting with Inheritance (Upcasting)

Upcasting happens automatically when a subclass object is assigned to a superclass reference. This is safe because the subclass is-a superclass.

Why is Upcasting Useful?

* Allows handling of multiple subtypes using a single reference type.
* Supports polymorphism, enabling dynamic method invocation at runtime.

A computer code with text

AI-generated content may be incorrect.

#### Explicit Casting with Inheritance (Downcasting)

Downcasting is manual casting from a superclass reference back to a subclass reference. This must be done carefully, as incorrect casting causes a ClassCastException.

Why Use Downcasting?

* When we know an object is actually a subclass instance and we need to access subclass-specific methods.

A close-up of a math formula

AI-generated content may be incorrect.

### Instanceof with abstract classes and interface

With interface:

A computer code with text

AI-generated content may be incorrect.

With abstract classes:

A screenshot of a computer code

AI-generated content may be incorrect.

## Linear data structure: Stacks and Queues

### Understanding Linear Data Structures

#### What are Linear Data Structures?

Linear data structures organize elements in a sequential manner, where each element is connected to its previous and next element. The order of elements is preserved, and operations like insertion and deletion are typically performed at one or both ends of the structure.

Data structures that store elements one after another in a linear fashion.

* Indexed Structure: Allows direct access to elements using an index (Array, ArrayList).
* Ordered Structure: Elements follow a specific sequence (Priority Queue, Sorted List).
* Unordered Structure: Elements do not follow a specific order (Linked List, HashSet).
* Static vs. Dynamic:
  + Static Structures: Arrays are the only static data structure, meaning their size is fixed at declaration and cannot change at runtime.
  + Dynamic Structures: ArrayLists, Linked Lists, Stacks, and Queues can grow or shrink as needed during execution.



#### When to use

* Use Arrays/ArrayLists for fast direct access.
* Use Linked Lists when frequent insertions/deletions are needed.
* Use Stacks when handling Last-In-First-Out (LIFO) operations.
* Use Queues when handling First-In-First-Out (FIFO) operations

### Stack Last In First Out (LIFO)

Stack is an ordered linear collection

Insertions and deletions are made only at the top of the stack (this is the end of the list)

Elements are added in a last in first out (LIFO) manner

A white background with black dots

AI-generated content may be incorrect.

#### Stack Applications

* For example, whenever a **function call**is made, the called function must know how to **return** to its caller, so the **return address**is pushed onto a stack.
* If a series of function calls occurs, the successive return values are pushed onto the stack in **last-in, first-out order**so that each function can return to its caller.
* Stacks support recursive function calls in the same manner as conventional non recursive calls.
* Stacks contain the space created for **automatic variables**on each invocation of a function.
* When the function returns to its caller, the space for that function's automatic variables is popped off the stack, and these variables no longer are known to the program.
* Stacks are used by compilers in the process of evaluating expressions and generating machine-language code.

#### The Stack Abstract Data Type

Reference: [Stack Abstract Data Type](https://runestone.academy/ns/books/published/javads/basic-ds_the-stack-abstract-data-type.html?mode=browsing)

Operations

* Stack() creates a new stack that is empty. It needs no parameters and returns an empty stack.



* push(item) adds a new item to the top of the stack. It needs the item and returns nothing.

A white background with blue text

AI-generated content may be incorrect.

* pop() removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.

A close-up of a white background

AI-generated content may be incorrect.

* peek() returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.

A close-up of a computer code

AI-generated content may be incorrect.

* isEmpty() tests to see whether the stack is empty. It needs no parameters and returns a boolean value.
* size() returns the number of items on the stack. It needs no parameters and returns an integer.

### Queues First In First Out (FIFO)

Reference: [Queues](https://runestone.academy/ns/books/published/javads/basic-ds_queues.html?mode=browsing)

* A queue is similar to a checkout line in a grocery store—the first person in line is serviced first, and other customers enter the line only at the end and wait to be serviced.
* Queue nodes are removed only from the head of the queue and are inserted only at the tail of the queue.
* For this reason, a queue is referred to as a first-in, first-out (F I F O) data structure.

The insert and remove operations are known as enqueue and dequeue, respectively

Chart, bar chart

Description automatically generated

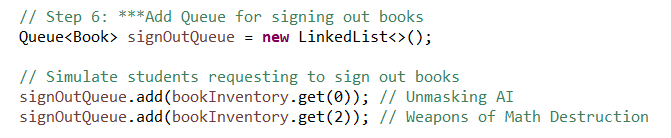
#### Application of Queues

* Information packets also wait in queues in computer networks.
* Queues are also used to support print spooling.
* For computers that have only a single processor, only one user at a time may be serviced

#### The Queue Abstract Data Type

* enqueue() – Add an element to the rear.





* dequeue() – Remove and return the front element.

A close-up of a computer code

AI-generated content may be incorrect.

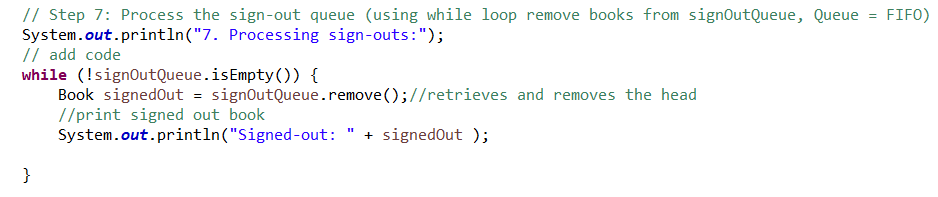
* peek() – Return the front element without removing it.

A black text on a white background

AI-generated content may be incorrect.

* isEmpty() – Check if the queue is empty.





### Compare stacks and queues, when to use each data structure.

|  |  |  |
| --- | --- | --- |
| Feature | Stack (LIFO) | Queue (FIFO) |
| Behavior | A stack is an ordered collection where the last element added is the first to be removed (LIFO). | A queue is an ordered collection where the first element added is the first to be removed (FIFO). |
| Operations | push(), pop(), peek(), isEmpty(), size() | enqueue(), dequeue(), peek(), isEmpty(), size() |
| Exception | use NoSuchElementException. | use IllegalStateException |
| Data Structure Used | Often implemented using ArrayList or LinkedList | Also implemented using ArrayList, LinkedList, or Queue interface |
| Feature | Stack (LIFO) | Queue (FIFO) |

## Linked Lists

A linked list is a sequence of data structures that are connected together, called nodes.

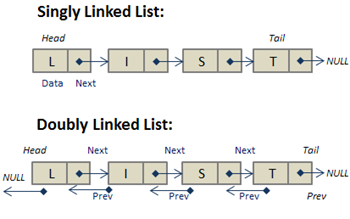
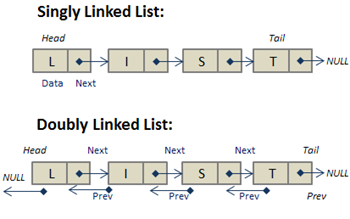
A linked list is accessed via a pointer to the first node of the list. Subsequent nodes are accessed via the link pointer member stored in each node.

The link pointer in the last node of a list is set to NULL to mark the end of the list. Data is stored in a linked list dynamically—each node is created as necessary. A node can contain data of any type including other structs.

Each node contains two parts:

Data – The actual value stored in the node.

Pointer (Reference) – A reference to the next node in the sequence

### Self-Referential Class

When a class member is a reference (pointer) to its same class type

Example:

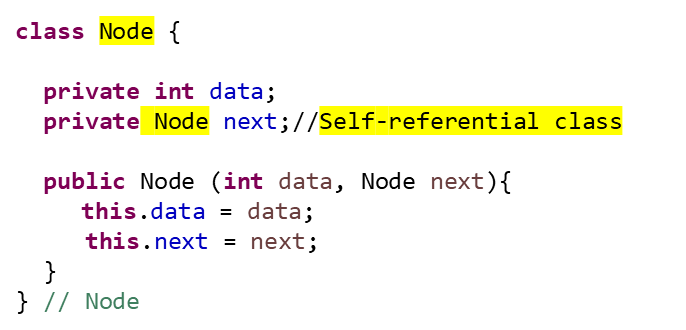
|  |  |
| --- | --- |
| Screenshot 2025-03-29 at 8.45.07 PM.png | The student object should be viewed as a box with two sections:  The 1st for the data (instance variables)  The 2nd for the reference (pointer) to another student (this is the link!) |

### Singly Linked List

A diagram of a data flow

AI-generated content may be incorrect.

* Create nodes using **self-referential** structures
* Each node represents one element in the linked list
* Each node contains two parts:
  + - Data field: The information the node holds
    - Next field: A reference (pointer) to the next node in the linked list
* Nodes are allocated when needed on the **heap**
* Nodes are objects that you link together to form linked data structures
* Class defining a node:



### Operations on a Linked List: insert, delete

When working with insertion and deletion in linked lists, we must keep track of:

current: The node we are currently examining.

previous: The node before current, which helps adjust pointers.

Tracking Pointer

previous starts as null and follows current as we traverse.

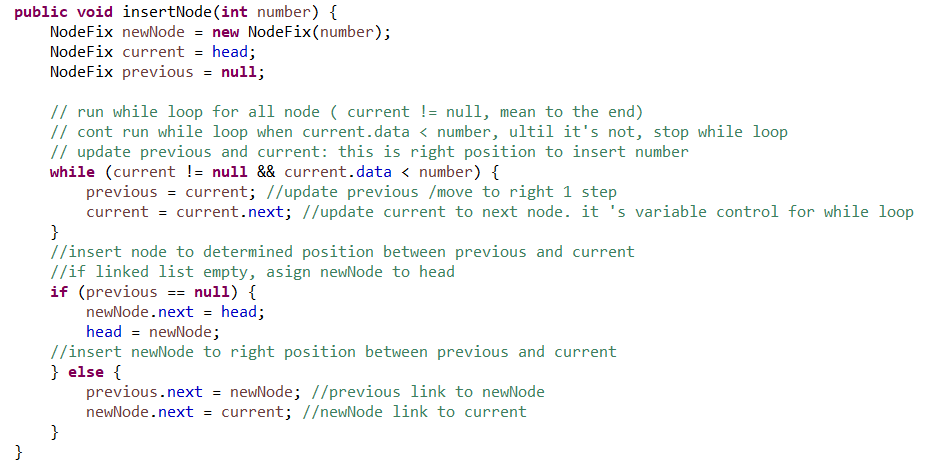
current moves forward to find the correct position

#### Traversing a linked list

Similar to iterating a 1D array, but we follow next pointers.

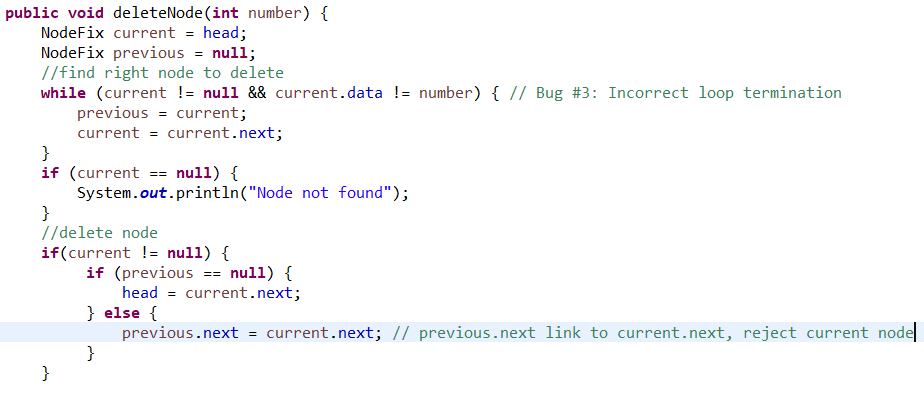
#### Insertion

* Insert at the beginning
* Insert at the end
* Insert at a specific position



#### Deletion

* Delete the first node
* Delete the last node
* Delete a node by value

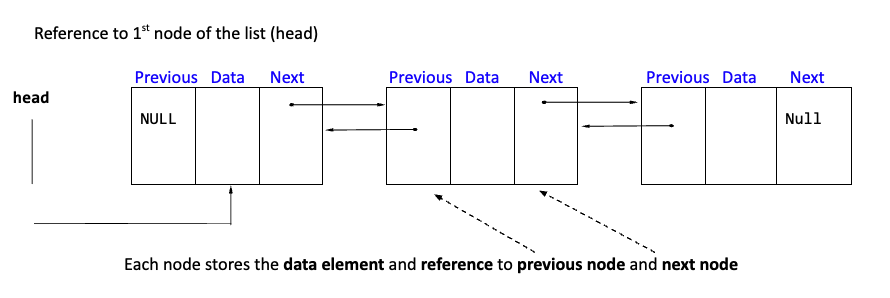


### Doubly linked

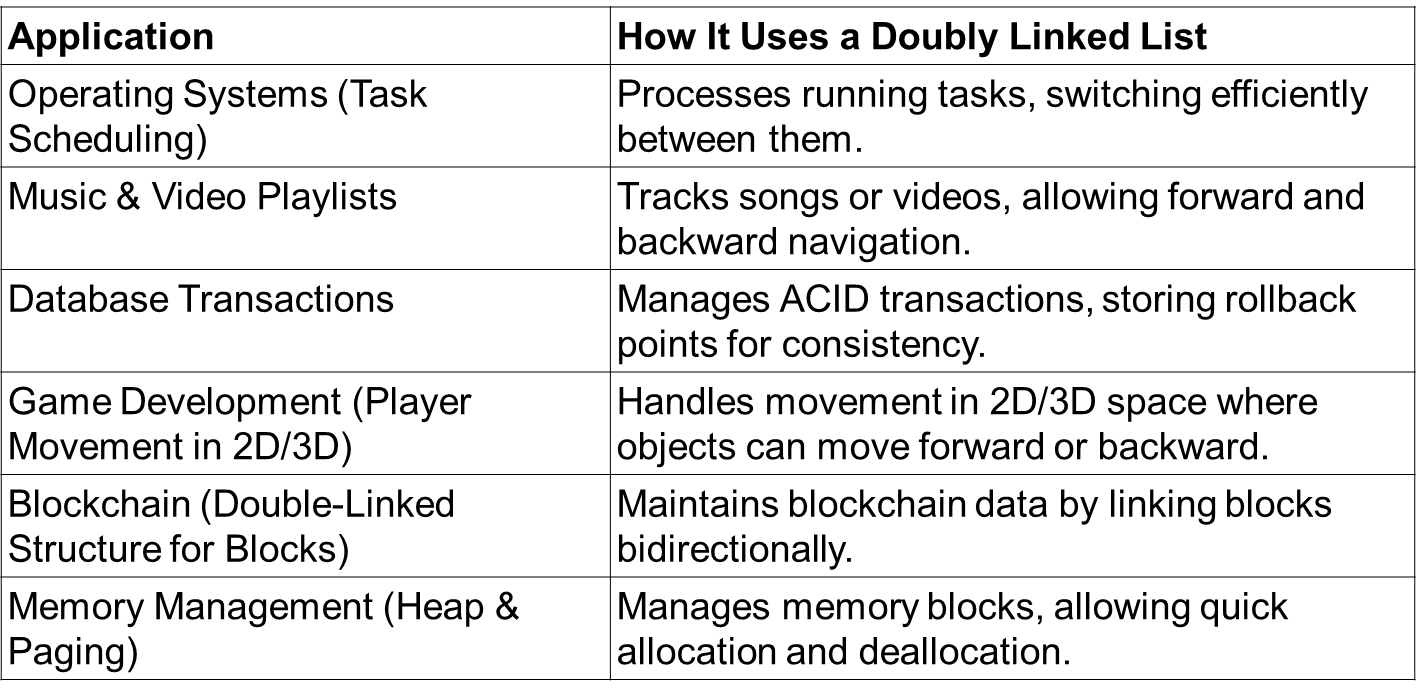
There are also doubly linked lists where each node contains three parts:

* Data field: The information the node holds
* Previous field:
  + A reference to the previous node in the list
  + First node’s previous reference is NULL
* Next field
  + A reference to the next node in the list
  + Last node’s next reference is NULL

Doubly Linked List: Allows traversal in both directions.

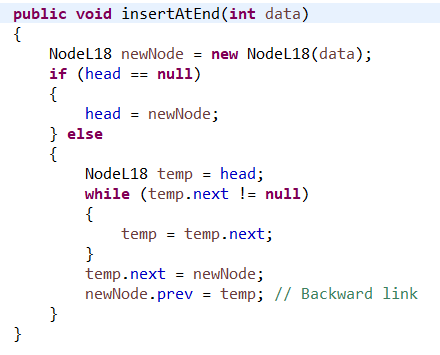


### Doubly Linked List Applications

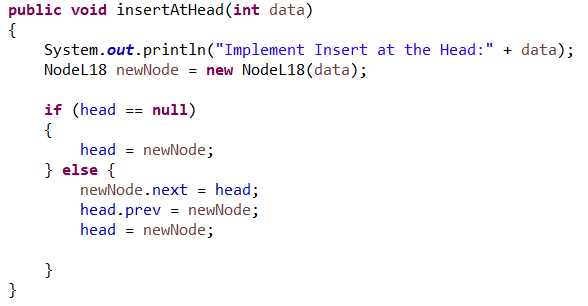


### Operations on a Doubly Linked List: insert, delete

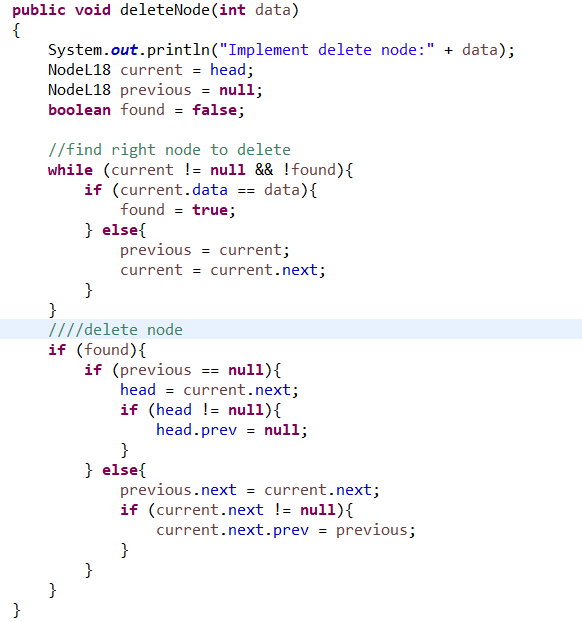
#### Insert at the end



#### insertAtHead



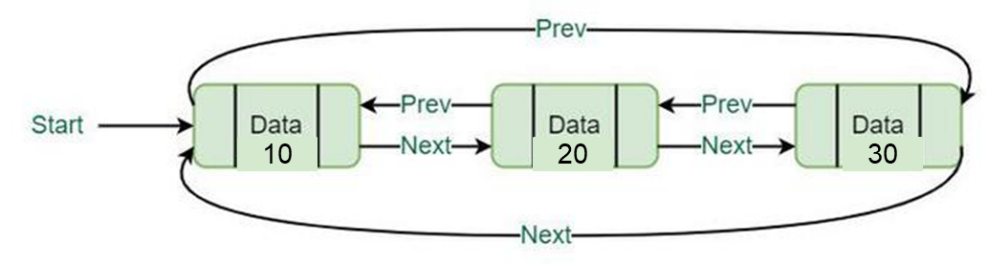
#### deleteNode



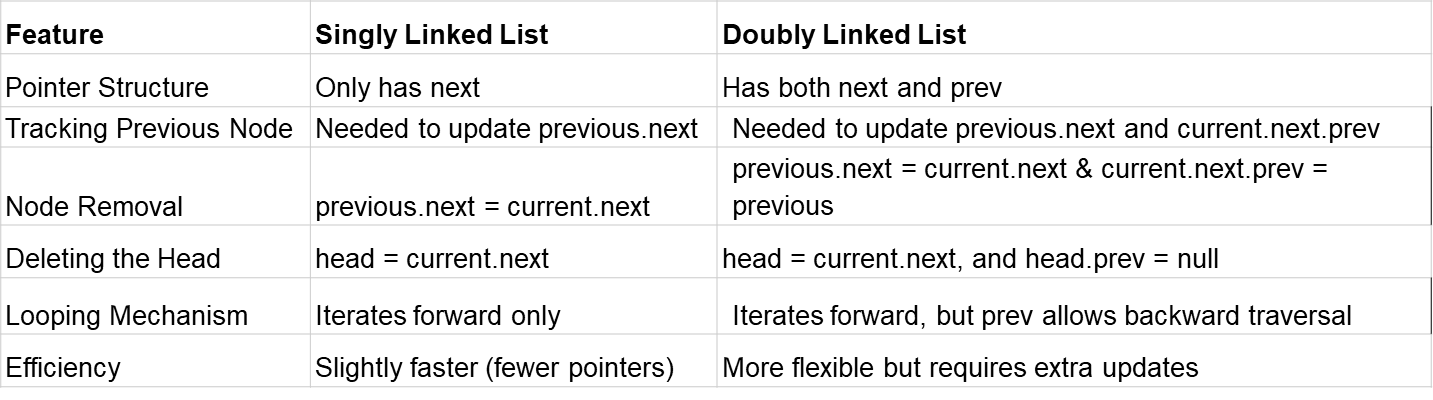
### Circular Doubly Linked Lists

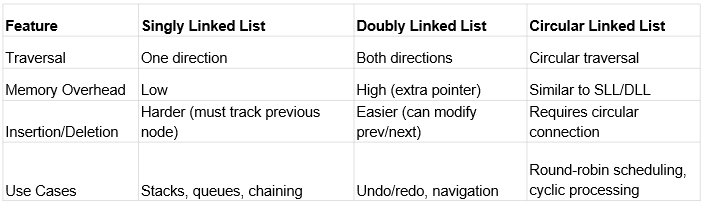
A circular doubly linked list is a doubly linked list where:

* The last node’s next pointer points back to the first node (instead of null).
* The first node’s prev pointer points to the last node.
* No null at the beginning or end (the list loops back).
* Traversal can continue infinitely in both directions.
* Efficient for cyclic operations (e.g., round-robin scheduling).



### Singly Linked List vs Doubly Linked List





### Comparison: Array, ArrayList and Linked List

|  |  |  |
| --- | --- | --- |
| Arrays | ArrayLists | LinkedLists |
| Use arrays when:   * You know exactly how many elements you'll store. * You want to store primitive types (like int, double) or objects and don’t need to resize. * You care about fast access (constant time using index) | Use ArrayLists when:   * You don’t know how many items you’ll store. * You want to add/remove items frequently. * You want automatic resizing (no need to worry about capacity). * You want to sort and search with utility methods (e.g., Collections.sort()). | Use linked lists when:   * You need to insert or remove elements in the middle of the list frequently. * You don’t care about random access (linked lists are slow to access by index). |
| Key Features:   * Fixed size. * Fast access by index (O(1)). * Cannot grow or shrink automatically. * Example: int[] nums = new int[10]; | Key Features:   * Resizable array. * Stores objects (not primitives directly). * Good for general use when order matters and size changes. * Slower than arrays for access but flexible. | Key Features:   * Can grow and shrink easily. * Efficient at adding/removing elements from the start or middle. * Poor performance for random access. * Useful for implementing queues and stacks. |

## **Collection interface and Sorting**

Collections.sort() is a utility method in Java that lets you sort elements in a List

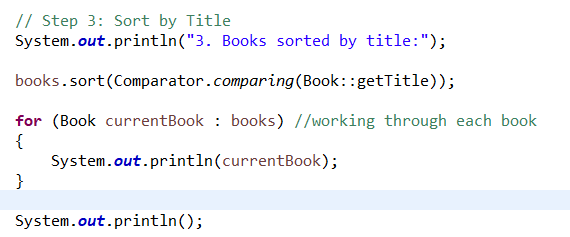
It is part of the java.util.Collections class — a toolbox of static methods to operate on collections.

### Implement sorting with Collections.sort() to sort custom objects

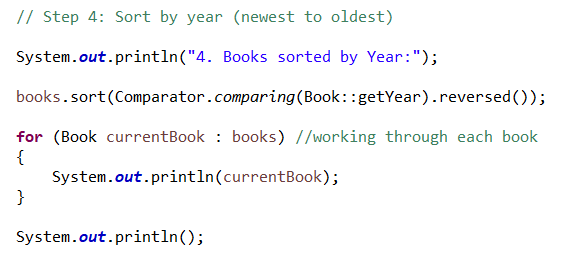
* + - Comparable<T> for natural ordering
    - Comparator<T> for custom ordering (e.g., by title, year, author)

#### Use Comparator.comparing

* Ascending order: books.sort(Comparator.comparing(Book::getTitle));

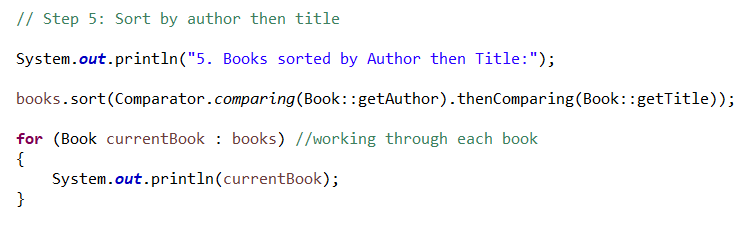


* Descending order: books.sort(Comparator.comparingInt(Book::getYear).reversed());



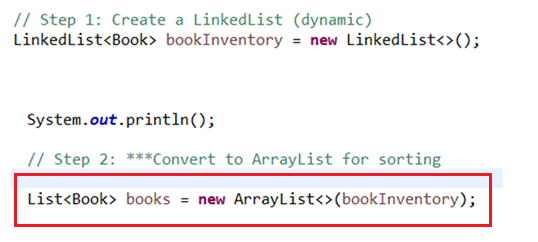
#### Sort by multiple fields

books.sort(Comparator.comparing(Book::getAuthor).thenComparing(Book::getTitle));



### Convert between LinkedList and ArrayList when needed for sorting.

LinkedList to ArrayList:



## Maps – HashMap

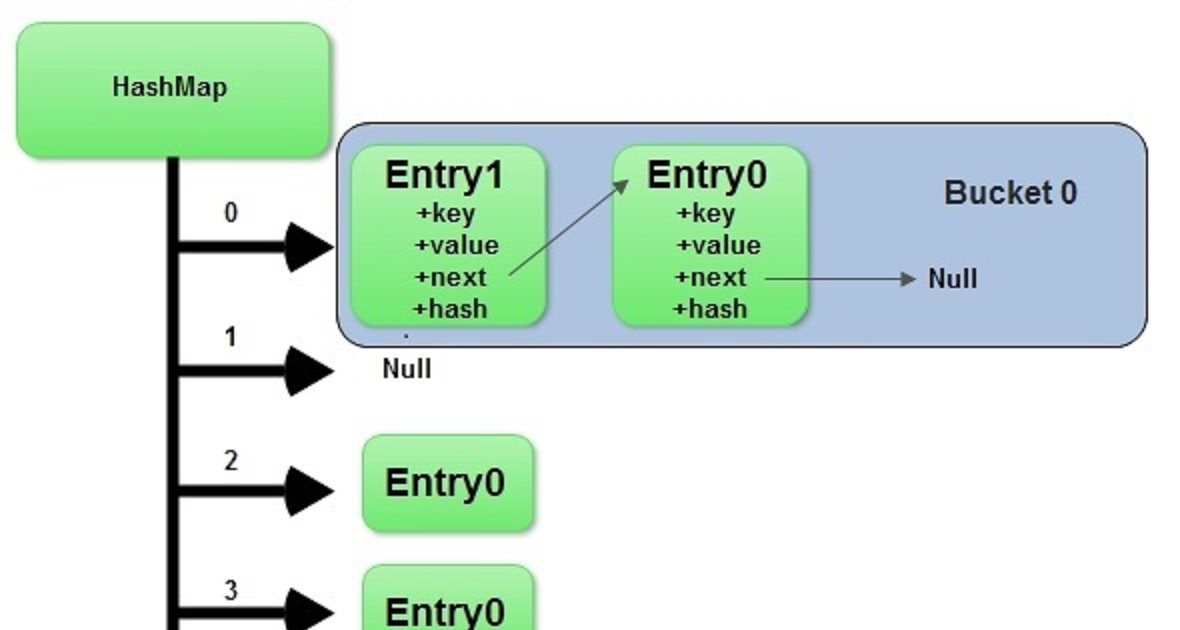
A Map is a Java collection that stores data in key–value pairs. Unlike a List or Set, which store individual elements, a Map lets you look up values using unique keys.

* **Each key is unique.**
* Each key maps to exactly one value.
* Common real-world example: a dictionary (word = key, definition = value).

Common Implementations:

* HashMap — fast lookups, no guaranteed order.
* TreeMap — sorted by keys.
* LinkedHashMap — maintains insertion order.

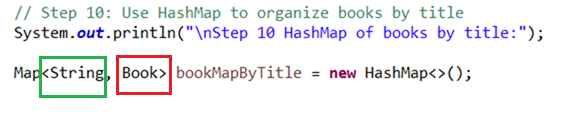
### What is a HashMap



A HashMap is a data structure that allows you to store and retrieve key-value pairs very efficiently

Key idea: HashMap<K, V> lets you store **key**-**value** pairs.

Like a dictionary — look up a word (key), get the definition (value).



### How does a HashMap Work

When you call **put(key, value)** in a HashMap, Java performs the following steps:

1. It computes a hash code for the key using the hashCode() method.
2. It uses the hash code to determine which "bucket" (internal array slot) to store the entry in.
3. If two keys have the same hash code (a "collision"), it stores them in a list or tree at that bucket.

A screenshot of a computer code

AI-generated content may be incorrect.

When you later call **get(key)**, Java:

1. Computes the hash code of the key.
2. Jumps directly to the bucket where that key should be.
3. Searches only within that bucket to find and return the matching value.

A computer code with text

AI-generated content may be incorrect.

Internally, a HashMap is built on two ideas:

1. An array of buckets (like bins).
2. Linked lists or trees inside each bucket to handle collisions.

### Common Map Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| put(key, value) | Adds or updates a key-value pair |
| get(key) | Returns the value associated with the key |
| remove(key) | Removes the key and its value |
| containsKey(key) | Returns true if the key exists |
| containsValue(value) | Returns true if the value exists |
| keySet() | Returns a Set of all keys |
| values() | Returns a Collection of all values |
| entrySet() | Returns a Set of key-value pairs (Map.Entry<K, V>) |
| isEmpty() | Checks if the map has no entries |
| size() | Returns the number of entries in the map |

### Why / when to Use Maps

Maps are ideal when you want to:

* Quickly find a value based on a key (average O(1) with HashMap).
* Avoid writing loops for lookup.
* Associate two pieces of information.

Real-World Applications of Maps:

* Track inventory (item ID → quantity)
* Store user accounts (username → user object)
* Cache recent results for fast lookup
* Associate book titles with book objects
* Track students by ID

### Comparison: Maps vs Lists

|  |  |  |
| --- | --- | --- |
| **Feature** | **List** | **Map** |
| Stores | Individual elements | Key–value pairs |
| Index-based? | Yes (e.g., list.get(0)) | No (access with map.get(key)) |
| Duplicates | Allows duplicate values | Keys must be unique |
| Access pattern | Access by index | Access by key |
| Use case | Ordered collection of elements | Fast lookup of value by key |
| Example | A List<String> might hold a list of book titles. | A Map<String, Book> might map each title to a Book object with more details. |