**CS2050 Technical Documentation**

Contents

[**Overview of Developer Technical Documentation** 2](#_Toc196394629)

[**Set Up Development Environment** 3](#_Toc196394630)

[Version Control with Git and GitHub 3](#_Toc196394631)

[IDE Information 4](#_Toc196394632)

[**General Resources** 5](#_Toc196394633)

[**Module 1** 6](#_Toc196394634)

[Primitive and Non-Primitive Data Types 6](#_Toc196394635)

[Arrays 6](#_Toc196394636)

[1D arrays 6](#_Toc196394637)

[Rules for arrays 7](#_Toc196394638)

[Stack and Heap 7](#_Toc196394639)

[Encapsulation 8](#_Toc196394640)

[Constructors 8](#_Toc196394641)

[Polymorphism 9](#_Toc196394642)

[Declaring and Creating an Object 10](#_Toc196394643)

[Instance Variables and Instance Methods 10](#_Toc196394644)

[Inheritance 11](#_Toc196394645)

[Reading from a file 11](#_Toc196394646)

[Writing to file 11](#_Toc196394647)

[Exception handling 11](#_Toc196394648)

[**Module 2** 12](#_Toc196394649)

[2D Arrays 12](#_Toc196394650)

[Bubble Sort 13](#_Toc196394651)

[Selection Sort 14](#_Toc196394652)

[Insertion Sort 15](#_Toc196394653)

[Linear Search 16](#_Toc196394654)

[Binary Search 16](#_Toc196394655)

[Arrays and Math Class 17](#_Toc196394656)

[Big-O Notation 18](#_Toc196394657)

[**Module 3** 19](#_Toc196394658)

[Abstract Classes and Methods 19](#_Toc196394659)

[Dynamic Binding 20](#_Toc196394660)

[Interface 21](#_Toc196394661)

[ArrayList 21](#_Toc196394662)

[For-Each Loop 22](#_Toc196394663)

[Stack 23](#_Toc196394664)

[Queue 24](#_Toc196394665)

[Linked List 25](#_Toc196394666)

[Singly Linked List 25](#_Toc196394667)

[Doubly Linked List 26](#_Toc196394668)

[Circular Doubly Linked List 27](#_Toc196394669)

[Collections.sort() 28](#_Toc196394670)

[Hashmap 29](#_Toc196394671)

# **Overview of Developer Technical Documentation**

My technical documentation includes:

* **Setup instructions**: Steps to configure my development environment (GitHub)
* **Version control**: Key commands for Git and how they work
* **IDE information**: Setting up and using IntelliJ IDEA, including downloading, installing JDK, and creating a new project.
* **General Resources**: Links to important resources
* **Modules**: Topics covered in modules

A screenshot of a computer

Description automatically generated

# **Set Up Development Environment**

1. Login to [GitHub](https://github.com)
2. Click on the “+’ icon and select “New repository.”

A screenshot of a computer

Description automatically generated

1. Enter a name and add .gitignore file then create the repository.
2. Go to IntellJ IDEA > File > Settings > Version Control > Git
3. Make sure that the path to Git is set correctly.

## Version Control with Git and GitHub

<https://docs.google.com/document/d/1lnLZdxusq1UNoUshdf3sK64QaTrNF5Iy/edit#heading=h.gjdgxs> contains information on how to set up Git and GitHub.

Git is a distributed version control system that tracks changes in code over time. It allows multiple people to work on the same project without messing up each other’s work.

GitHub is a developer platform that allows developers to create, store, manage, and share their code. It uses Git software to track changes and manage versions of code.

Important GitHub commands:

**git init** – Makes a repository (Only done once per repository)

**git status** – Extremely useful to check now and then

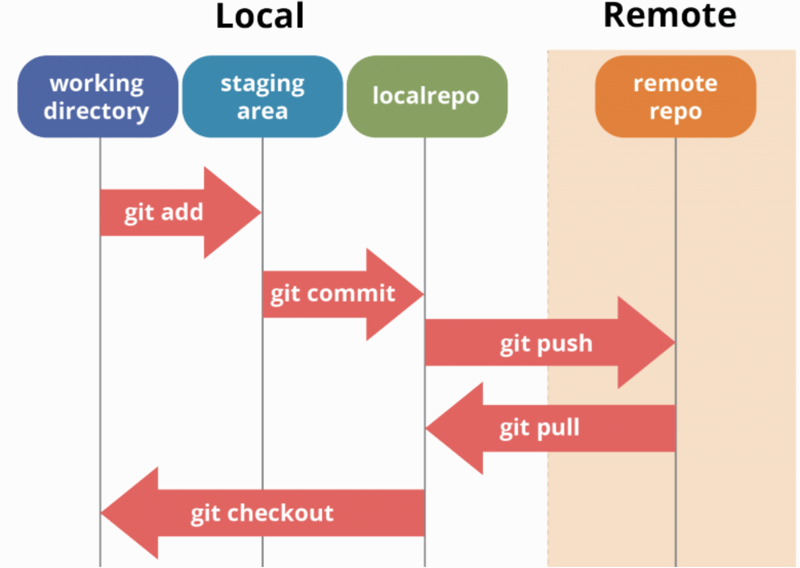
**git add** – Start tracking or include changes for a file

**git commit -m “message”** – Commit the changes with a message

**git push** – Push the branch to GitHub (remote)  
**git pull** – Pull the changes for the current branch

**git checkout -b** – Make a separate branch

**git reset - -hard** – Resets back to commit   
**git branch -D** – Deletes the branch



## IDE Information

Setting up a project with IDE

1. [Download JetBrains IntellJ IDEA](https://www.jetbrains.com/idea/download/?section=windows)  
   Choose windows  
   Complete Setup, and install required plugins
2. [Install JDK](https://www.oracle.com/java/technologies/downloads/)  
   JDK is required to compile and run Java applications
3. Create a New Project from the welcome screen  
   Click **File, New, and Project** if the welcome screen isn’t displayedA screenshot of a computer

   Description automatically generated
4. Right-click on your project and create a Java Class  
     
   A screenshot of a computer

   Description automatically generated
5. [More information (includes running, registering, updating, uninstalling, creating Java applications, features, plugins, etc.)](https://www.jetbrains.com/help/idea/run-for-the-first-time.html)

# **General Resources**

Here you can list resources that you use frequently.

* [Shared student resources containing resources, lectures and assignments](https://drive.google.com/drive/folders/1HvYY8zzSwlsH--03olvqOJooGnJkZ7F4)
* [Draft Schedule](https://docs.google.com/spreadsheets/d/1igBbmOBTXfvEVicyAggnqRIpV5Fwqh64/edit?gid=2047083326#gid=2047083326)
* [Link to join lecture in teams](https://teams.microsoft.com/l/meetup-join/19%3aklQhREluFbWiaroMMZPBYeNPhZa9AFGnTb7ATIPTUFE1%40thread.tacv2/1724008042961?context=%7b%22Tid%22%3a%2203309ca4-1733-4af9-a73c-f18cc841325c%22%2c%22Oid%22%3a%2233eb6fec-88d5-4bc1-bb67-32063f1cfacc%22%7d)
* [Syllabus](https://msudenver.instructure.com/courses/95281/assignments/syllabus)
* [Announcements](https://msudenver.instructure.com/courses/95281/announcements)
* [Modules](https://msudenver.instructure.com/courses/95281/modules)

# **Module 1**

## Primitive and Non-Primitive Data Types

Primitive data types are simple values like numbers, characters (int, double, char, boolean).

Non-Primitive data type are complex objects. (String, Arrays, Class, Objects)

You can’t put the contents from a larger variable into a smaller variable unless you do casting.

Implicit/Widening casting is a value with a smaller range to one with bigger range. Java does this automatically.

Explicit/Narrowing casting is a value with a bigger range to one with a smaller range. You must do this manually.

Example: double a = 10.85;

int b = (int) a; // Decimal part is lost so it becomes 10

## Arrays

An array is a data structure that provides a way to store more than one value. Arrays must be the same data type.   
  
There are several types of arrays:

## 1D arrays

- 1D array is used to store multiple values in a single variable  
- String[] cars;  
- To insert values, you have to place the values with commas inside curly braces  
- Int[] num = {1, 2, 3, 4, 5};  
- You can iterate these values with a loop (FOR, WHILE, DO WHILE) depending  
on what you need

This is how you declare and initialize a 1D array:   
A screen shot of a computer code

Description automatically generated

## Rules for arrays

1. When an array is created, its size is fixed. You can’t change the size of an array after it’s initialized.
2. The index for arrays starts at 0. The first element is at index 0, then 1..2..3.. etc.
3. You can get the length of an array using the .length attribute.
4. Everything inside the array needs to be the same type. For example, you can’t mix strings and integers. It can’t hold elements of mixed types.
5. If an array is created but not initialized, it will default to the value 0 for integers and null for reference types.
6. ArrayIndexOutOfBoundsException happens when the index is outside the bounds of an array (Remember the array starts at index 0).

## Stack and Heap

A stack frame is a data structure that holds information about a method’s execution in a program. When a method is called, a new stack frame is created and pushed onto the call stack, and when the method returns, the frame is popped off.

Stores local variables, and reference to objects.

Heap is used for storing objects and their instance variables.

Example:

Int a = 5; // ‘a’ is stored in the stack

String name // This is a instance variable, stored in heap

Car myCar = new Car(“Toyota”, 2020);

myCar is a reference variable stored in the stack, but it points to an object created in the heap. The car object properties like name, year, model, etc. are stored in the heap.

When an object is created, Java allocates memory for that object in the heap but the reference variable that holds the object’s memory address is stored in the stack. The object itself (its data) is stored in the heap.

## Encapsulation

Encapsulation is to make sure that “sensitive” data is hidden from users.

How to:

Declare class variables/attributes as private.

Provide getter and setter methods to access and update the value of a private variable.

Getter and setter methods:  
A screen shot of a computer code

Description automatically generated

More information: <https://www.w3schools.com/java/java_encapsulation.asp>

## Constructors

Constructors have the same name as the class and can be overloaded. It doesn’t need a return type.

For example:

class Rectangle {

double area;

# Construct a rectangle with area of 1

Rectangle() {

area = 1;

}

# Construct a rectangle with specified area

Rectangle(double newArea) {

area = newArea;

}

## Polymorphism

Polymorphism allows objects to be treated as instances of their parent class while still maintaining their own behavior.

This can be achieved in two ways, Overloading and Overriding

Overloading is when multiple methods in the same class have the same name but different parameters.

Reminder: Overloading is a core concept of how static binding works; Overriding is a core concept of how dynamic binding works.

Example:

class Animal {

void makeSound() {

System.out.println("Some generic animal sound");

}

void makeSound(String sound) {

System.out.println("Animal makes sound: " + sound);

}

}

public class Main {

public static void main(String[] args) {

Animal a = new Animal();

a.makeSound(); // Calls the method with no parameters

a.makeSound("Roar!"); // Calls the overloaded method with a String parameter

}

}

Method overwriting happens when a subclass provides a specific implementation of a method that is already defined in its superclass. The method in the subclass must have the same name, return type, and parameters as in the superclass.

Example:

public class Main {

public static void main(String[] args) {

Animal myAnimal = new Dog();

myAnimal.makeSound(); // Calls Dog's version of makeSound()

}

}

class Animal {

void makeSound() {

System.out.println("Some generic animal sound");

}

}

class Dog extends Animal {

@Override

void makeSound() {

System.out.println("Bark! Bark!");

}

}

## Declaring and Creating an Object

SimpleCircle circle1;

This is a variable for declaring object, which is a reference variable.

SimpleCircle circle1 = new SimpleCircle();

Object variable circle1 contains memory address on the heap.

## Instance Variables and Instance Methods

Instance variable is a part of an instance of a class.

Instance method can be executed only through a reference to an instance of a class.

Example:

class Car

Fields: model, make, color

These are instance variables

Brake(), Start(), Name()  
These are instance methods

## Inheritance

Inheritance allows a class (subclass) to inherit properties and methods from another class (superclass).

Example:

// Parent class (Superclass)

class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

// Child class (Subclass) inheriting from Animal

class Dog extends Animal {

// Overriding the sound method

void sound() {

System.out.println("Dog barks");

}

}

## Reading from a file

Scanner class is used to read from file, and nextLine() method allows you to read line by line.

## Writing to file

You can open, read, and write data to files from a program. A file path is the “address” of a file on the computer, showing where it’s stored. You can write to a file by creating a PrinterWriter object.

## Exception handling

1. Try Block: Code that might throw an exception.
2. Catch Block: Handles the exception if it occurs.
3. Finally Block: Executes code regardless of whether an exception was thrown or not.
4. Throw: Used to explicitly throw an exception.
5. Throws: Declares exceptions a method might throw.

Example of writing to file and try-catch:

import java.io.PrintWriter;

import java.io.FileNotFoundException;

public class PrintWriterTryCatch {

public static void main(String[] args) {

try {

// Create a PrintWriter to write to a file

PrintWriter writer = new PrintWriter("output.txt");

// Write some text to the file

writer.println("Hello, this is a test file.");

// Close the writer to save the changes

writer.close();

System.out.println("File written successfully.");

} catch (FileNotFoundException error) {

System.out.println("Error: File could not be created or opened.");

}

}

}

# **Module 2**

## 2D Arrays

A 2D array is an array of arrays, that allows you to store data in a grid-like format (rows and columns).

A computer screen shot of a program code

AI-generated content may be incorrect.

Iterating 2D array from the example above:

* printCars(cars); calls the printCars() method and passes the 2D array.
* Inside printCars(), carArray[row][col].printMake(); calls printMake() on each Car object in the array.

## Bubble Sort

Bubble Sort is a way to sort a list by comparing two items at a time and swapping them if they are in the wrong order.

**How it works:**

* Compare the first two elements. If the first is bigger, swap them.
* Move to the next pair and do the same. Keep going until the last element.
* Repeat the process for the remaining elements until no swaps are needed.

**When to use it:**

* Only for very small lists.
* It is slow for large lists because it has to keep swapping elements many times.

**Code:**

   public static void bubbleSort(int[] array) {

        int n = array.length;

        // Iterate over the array multiple times

        for (int i = 0; i < n - 1; i++) {

            // Compare adjacent elements and swap if necessary

            for (int j = 0; j < n - 1 - i; j++) {

                if (array[j] > array[j + 1]) {

                    int temp = array[j];

                    array[j] = array[j + 1];

                    array[j + 1] = temp; // Swap elements

                }

            }

        }

    }

## Selection Sort

Selection Sort is a way to sort a list by finding the smallest item and moving it to the correct position. It goes through the list, picks the smallest number, and swaps it with the first position. Then, it finds the next smallest number and places it in the next position. This keeps happening until the whole list is sorted.

**How it works:**

* Find the smallest element in the list and swap it with the first item.
* Look at the remaining list, find the next smallest item, and swap it with the second position.
* Repeat this process until all elements are in order.

**When to use it:**

* Good for small lists or when memory is limited because it doesn't use extra space.
* It is slow for big lists because it still takes too many steps.

**Code:**

    public static void selectionSort(int[] array) {

        int n = array.length;

        // Iterate over the array except the last element

        for (int i = 0; i < n - 1; i++) {

            int minIndex = i; // Assume the first element of the unsorted part is the smallest

            // Find the smallest element in the unsorted part

            for (int j = i + 1; j < n; j++) {

                if (array[j] < array[minIndex]) {

                    minIndex = j; // Update minIndex if a smaller element is found

                }

            }

            // Swap the smallest element with the first unsorted element

            int temp = array[minIndex];

            array[minIndex] = array[i];

            array[i] = temp;

        }

    }

## Insertion Sort

**Insertion Sort** is a way to sort a list by taking one item at a time and placing it in the correct position, like sorting playing cards in your hand. It goes through the list, picks an item, and shifts the other items to make space for it in the right place.

**How it works:**

* Start with the second item and compare it with the one before it. If needed, swap them.
* Move to the next item and place it in the correct position by shifting larger items to the right.
* Keep doing this for every item until the whole list is sorted.

**When to use it:**

* Good for small lists or when the list is already mostly sorted.
* It is slow for big lists.

 public static void insertionSort(int[] array) {

        int n = array.length;

        // Start from the second element (index 1), assuming the first element is sorted

        for (int i = 1; i < n; i++) {

            int key = array[i]; // The element to be inserted in the sorted portion

            int j = i - 1;

            // Shift elements of the sorted portion to the right if they are greater than the key

            while (j >= 0 && array[j] > key) {

                array[j + 1] = array[j];

                j--;

            }

            // Insert the element in the correct place

            array[j + 1] = key;

        }

    }

## Linear Search

Linear Search is a simple way to find an item in a list by checking each item one by one until you find the right one or reach the end.

**How it works:**

* Start at the first item and check if it matches what you are looking for.
* If it matches, stop the search.
* If not, move to the next item and repeat until you find the right one or reach the end.

**When to use it:**

* Good for small lists or when the list is not sorted.
* It is slow for big lists because it checks every item, so faster methods like Binary Search are better for sorted lists.

A diagram of a diagram

AI-generated content may be incorrect.

**Code:**

public static int linearSearch(int[] list, int key) {

for (int i = 0; i < list.length; i++) {

if (list[i] == key) {

return i;

}

}

return -1;

}

## Binary Search

Binary Search is a fast way to find an item in a sorted list by cutting the list in half repeatedly until the item is found.

**How it works:**

* Start in the middle of the list. If the middle item is what you are looking for, stop.
* If the item is smaller, search in the left half. If it is bigger, search in the right half.
* Keep cutting the list in half until you find the item or there is nothing left to check.

**When to use it:**

* Works best for large lists that are already sorted.
* It is much faster than Linear Search, but it doesn’t work on unsorted lists.

A screenshot of a computer

AI-generated content may be incorrect.

**Code:**

public static int binarySearch(int[] list, int key) {

int low = 0;

int high = list.length - 1;

while (high >= low) {

int mid = (low + high) / 2;

if (key < list[mid]) {

high = mid - 1;

} else if (key == list[mid]) {

return mid;

} else {

low = mid + 1;

}

}

return -low - 1;

}

## Arrays and Math Class

The Arrays class in Java is part of the java.util package and provides utility methods for working with arrays. It helps with sorting, searching, filling, and converting arrays to strings.

**Common Methods in Arrays Class**

* **Sorting:** Arrays.sort(arr); = Sorts an array in ascending order.
* **Binary Search:** Arrays.binarySearch(arr, target); = Finds an element in a sorted array.
* **To String:** Arrays.toString(arr); = Converts an array into a readable string.

The Math class in Java is part of the java.lang package and provides useful mathematical functions like calculations, rounding, and trigonometry. It contains only static methods, so you don’t need to create an object to use them.

**Common Methods in Math Class**

* **Maximum & Minimum:** Math.max(a, b);, Math.min(a, b); = Finds the larger or smaller of two numbers.
* **Random Number:** Math.random(); = Generates a random decimal between 0.0 and 1.0.

Since Arrays and Math Class is a final class (cannot be extended), and all its methods are static, you simply use them without instantiating the class.

## Big-O Notation

Big-O Notation is a theoretical approach for analyzing algorithms independent of computers and specific input.

A rating system for algorithms.

The rating (Big-O) tells you how fast you can expect the algorithm to be as the number of inputs gets larger and larger.

Growth Rates

Order of n (best to worse)

* Constant O(1)
* Logarithmic O(log n)
* Linear O(n)
* Log Linear O(n log n)
* Quadratic O(n2)
* Polynomial O(nc)
* Exponential O(2n)

A diagram of a graph

AI-generated content may be incorrect.

# **Module 3**

## Abstract Classes and Methods

Abstract class is a class that can’t be instantiated directly. It’s meant to be inherited by other classes. Abstract classes can have both abstract methods (methods without a body) and concrete methods (methods with a body)

Abstract class is declared using the abstract keyword.

Abstract method is declared in an abstract class without a body, which must be implemented by subclasses.

Example:

abstract class Animal {

// Abstract method (no body)

abstract void sound();

// Regular method (with body)

void sleep() {

System.out.println("The animal is sleeping");

}

}

class Dog extends Animal {

// Providing implementation of the abstract method

void sound() {

System.out.println("Bark");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.sound(); // Output: Bark

dog.sleep(); // Output: The animal is sleeping

}

}

So in this example, Animal is an abstract class with an abstract method sound() and a concrete method sleep(). Dog is a subclass that implements the abstract method sound().

|  |  |
| --- | --- |
| Concrete Class | Abstract Class |
| Can be instantiated (objects can be created) | Cannot be instantiated directly |
| All methods must have a body | Can have both abstract (no body) and concrete (with body) methods |
| Cannot have abstract methods | Can have abstract methods that must be implemented by subclasses |
| Can be inherited, but doesn’t require the subclass to implement anything | Must be inherited, and abstract methods in the abstract class must be implemented by subclasses |

## Dynamic Binding

Dynamic binding refers to the process where the method to be invoked is determined at runtime rather than at compile time. This typically happens when method overriding is used, meaning the method in the subclass is called instead of the one in the superclass.

Dynamic Binding occurs with overridden methods.

It allows Java to decide which method to call based on the object type, not the reference type.

Example:

class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

@Override

void sound() {

System.out.println("Dog barks");

}

}

public class Main {

public static void main(String[] args) {

Animal animal = new Dog();

animal.sound(); // Output: Dog barks (Dynamic Binding)

}

}

At runtime, Java decides to call the sound() method in the Dog class, that’s why the output is “Dog barks” instead of “Animal makes a sound".

## Interface

An interface is a description of how two things interact. For example, a car’s owner’s manual contains the details of how to interact with the car. A class that implements an interface must provide implementations for all the abstract methods of the interface (unless the class is abstract).

Interfaces provide a way to abstract out functionality without specifying how it's implemented.

A class can implement multiple interfaces, overcoming Java’s limitation of single inheritance.

Interface rules:

Interfaces can contain only: Constants, Abstract methods and Default and static methods.

By default, interface variables are public static final, this means they are constants, and interface methods are public abstract.

An interface can extend other interfaces, but can’t extend a class or implement an interface.

A class can extend one class and implement any number of interfaces.

## ArrayList

An ArrayList is a resizable array implementation of the List interface. Unlike arrays, the size of an ArrayList can grow or shrink dynamically as elements are added or removed.

* The size grows or shrinks automatically as elements are added or removed.
* You can access elements in an ArrayList using an index, just like an array.
* It can store duplicate elements.
* The order of elements is preserved.
* Known as a generic class.

Declaring an ArrayList:

ArrayList<Type> listName = new ArrayList<>();

Type: The type of elements you want to store like String, Integer, etc.).

listName: The name you give to the list.

Example:

ArrayList<String> animals = new ArrayList<>();

To add elements:

animals.add(“Dog”);

To remove elements:

animals.remove(0);

To get the size of the ArrayList:

Int size = animals.size();

## For-Each Loop

A for-each loop in Java is a simplified version of the traditional for loop. It is used to iterate over elements in an array or any iterable collection (like an ArrayList). It automatically handles the loop counter and checks for the number of elements, making it cleaner and more readable.

Syntax:

for (Type variable : collection)

Type: The type of the elements in the collection. For example, int, String, etc.).

Variable: A temporary variable that will hold each element in the collection during the iteration.

Collection: The array or collection (e.g., ArrayList) being iterated over.

Example:

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

// Creating an ArrayList of animal names

ArrayList<String> animals = new ArrayList<>();

animals.add("Lion");

animals.add("Tiger");

animals.add("Elephant");

animals.add("Zebra");

// Using for-each loop to print each animal name

for (String animal : animals) {

System.out.println(animal);

}

}

}

Output:

Lion

Tiger

Elephant  
Zebra

## Stack

A Stack is a linear data structure that follows the Last In, First Out (LIFO) principle. The last element added to the stack is the first one to be removed.

Operations:

* push(element): Adds an element to the top of the stack.
* pop(): Removes the element from the top of the stack and returns it.
* peek(): Views the element at the top of the stack without removing it.
* isEmpty(): Checks if the stack is empty.
* size(): Returns the number of elements in the stack.

Example:

import java.util.Stack;

public class StackExample {

public static void main(String[] args) {

Stack<String> stack = new Stack<>();

stack.push("Dog");

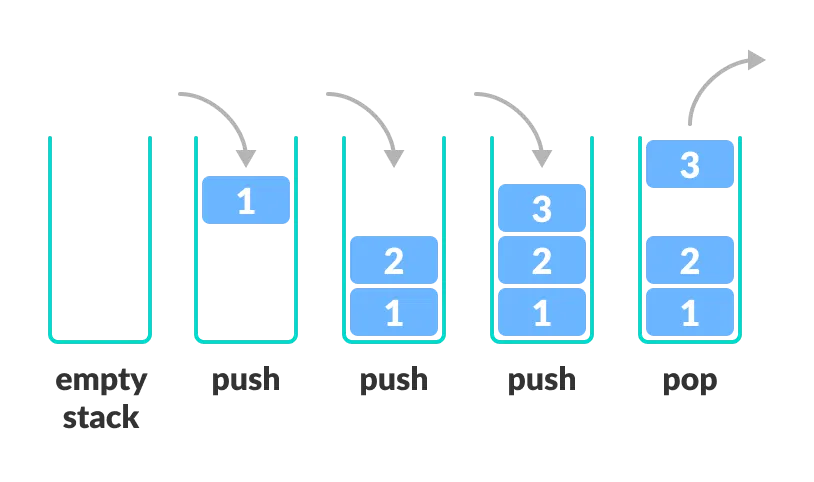
stack.push("Cat");

System.out.println(stack.pop()); // Output: Cat because it’s the last one.

System.out.println(stack.peek()); // Output: Dog because Cat was popped.

}

}



## Queue

A Queue is a linear data structure that follows the First In, First Out (FIFO) principle. The first element added to the queue is the first one to be removed. For example, imagine a coffee shop with a line of customers. The first customer comes to the cashier, gets their coffee and is the first one to leave.

Operations:

* offer(element): Adds an element to the end of the queue.
* poll(): Removes the element from the front of the queue and returns it.
* peek(): Views the element at the front of the queue without removing it.
* isEmpty(): Checks if the queue is empty.
* size(): Returns the number of elements in the queue.

Example:

import java.util.LinkedList;

import java.util.Queue;

public class QueueExample {

public static void main(String[] args) {

Queue<String> queue = new LinkedList<>();

queue.offer("Dog");

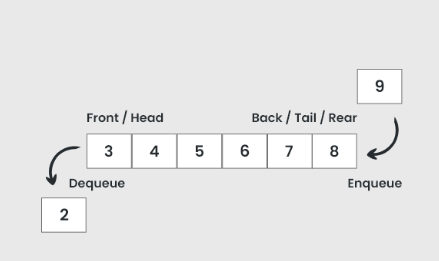
queue.offer("Cat");

System.out.println(queue.poll()); // Output: Dog because it’s the first one in queue.

System.out.println(queue.peek()); // Output: Cat because Dog is removed from queue.

}

}



## Linked List

A Linked List is a linear data structure where elements (called nodes) are stored in a non-contiguous manner. Each node contains:

* Data: The value stored in the node.
* Next: A pointer to the next node in the list (for singly linked lists).
* Previous: A reference to the previous node (for doubly linked lists).

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 the linked list dynamically and each node is created as necessary. A node can contain data of any type including other structs.

There are different types of linked lists:

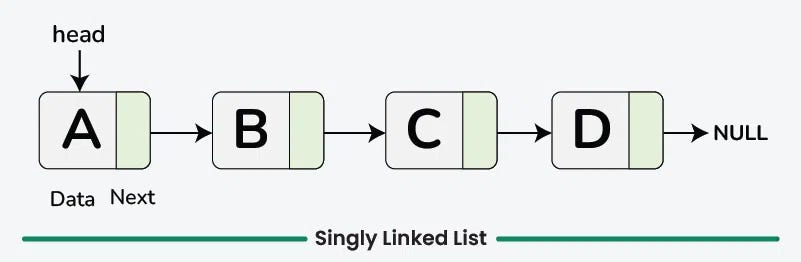
* Singly Linked List: Each node points to the next node only.
* Doubly Linked List: Each node points to both the next and the previous node.
* Circular Linked List: The last node points back to the head node, making the list circular.

### Singly Linked List

In a Singly Linked List, each node points to the next node, and the last node points to null, indicating the end of the list.

Operations:

* Insert a node (at the beginning, end, or specific position).
* Delete a node (from the beginning, end, or specific position).
* Traverse the list (print all elements).
* Search for a value.



### Doubly Linked List

A Doubly Linked List is a type of linked list in which each node contains three parts: Data, Next, and Previous.

n a Doubly Linked List, each node can be traversed in both directions:

* Forward using the next pointer.
* Backward using the previous pointer.

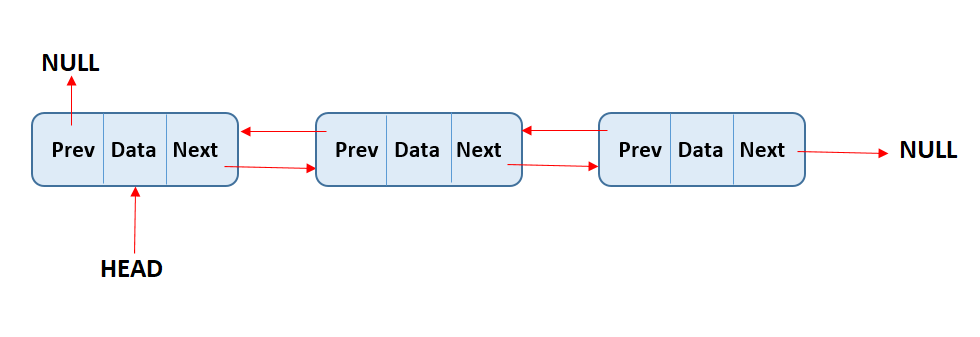
This is different from Singly Linked List, where each node only has a reference to the next node and cannot be traversed backward.

A node contains data, has a reference to the next node (next), and has a reference to the previous node (prev).

Operations:

* Insertion: Add a new node at the beginning, end, or any specific position.
* Deletion: Remove a node from the beginning, end, or any specific position.
* Traversal: Visit each node and perform operations, both forward and backward.

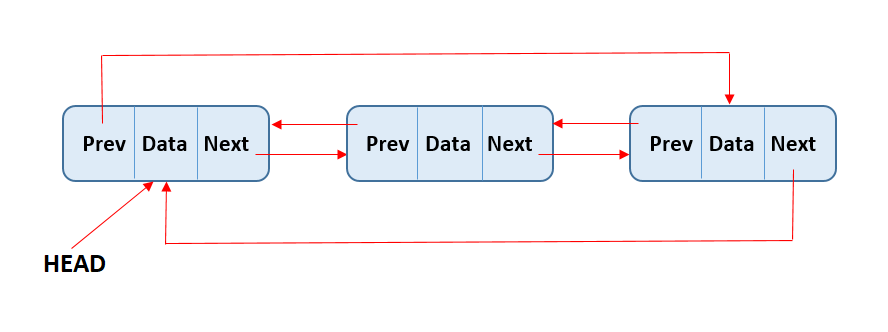
Doubly linked lists are used in situations where you need to move in both directions through the list efficiently.



### Circular Doubly Linked List

In a circular doubly linked list, the last node's next points to the head node, and the head node's prev points to the last node, creating a circular structure in both directions (forward and backward).

This means, no node has a null reference for its next or prev points (expect when the list is empty).



|  |  |  |  |
| --- | --- | --- | --- |
| Feature | Singly Linked List | Doubly Linked List | Circular Linked List |
| Traversal | One direction | Both direction | Circular traversal |
| Memory Overhead | Low | High | Similar to SLL/DLL |
| Insertion/Deletion | Harder | Easier | Requires circular connection |
| Use cases | Stacks, queues, chaining | Undo/redo, navigation | Round-robin scheduling, cyclic processing |

## Collections.sort()

This method is used to sort a List of elements. It sorts the elements in natural order (for numbers, it sorts in ascending order; for strings, it sorts alphabetically).

Use Comparator.comparing() to sort by a field.

Convert from LinkedList to ArrayList for sorting.

Syntax:

Collections.sort(List<T> list)

List: The list to be sorted. This list should implement the List interface, and its elements must be comparable (they must implement the Comparable interface).

Example:

import java.util.\*;

public class SortNumbersExample {

public static void main(String[] args) {

List<Integer> numbers = new ArrayList<>();

numbers.add(15);

numbers.add(30);

numbers.add(5);

numbers.add(25);

// Sort the numbers in ascending order using Comparator.comparing()

Collections.sort(numbers, Comparator.comparingInt(Integer::intValue));

// Print the sorted list

System.out.println("Sorted Numbers in Ascending Order: " + numbers);

}

}

Comparator.comparingInt(Integer::intValue): This creates a comparator that compares the integers by their values in natural (ascending) order. Here, Integer::intValue is a method reference that extracts the integer value from each Integer object.

Collections.sort(): Sorts the list using the comparator created by Comparator.comparingInt().

Output:

Sorted Numbers in Ascending Order: [5, 15, 25, 30]

You can sort the numbers in descending order by using the reversed() method:

import java.util.\*;

public class SortNumbersExample {

public static void main(String[] args) {

List<Integer> numbers = new ArrayList<>();

numbers.add(15);

numbers.add(30);

numbers.add(5);

numbers.add(25);

// Sort the numbers in descending order using Comparator.comparingInt()

Collections.sort(numbers, Comparator.comparingInt(Integer::intValue).reversed());

// Print the sorted list

System.out.println("Sorted Numbers in Descending Order: " + numbers);

}

}

Comparator.comparingInt(Integer::intValue).reversed(): This sorts the integers in descending order by first comparing them in ascending order and then reversing the order.

Output:

Sorted Numbers in Descending Order: [30, 25, 15, 5]

## Hashmap

A HashMap is part of the Java Collections Framework, and it stores key-value pairs. It allows you to store data where each element is associated with a unique key.

The HashMap does not guarantee any specific order of the elements, meaning it does not maintain the order in which elements are inserted.

Basic Operations:

* put(key, value): Adds a key-value pair to the map.
* get(key): Retrieves the value associated with the specified key.
* remove(key): Removes the key-value pair associated with the specified key.
* containsKey(key): Checks if the map contains the specified key.
* containsValue(value): Checks if the map contains the specified value.
* size(): Returns the number of key-value pairs in the map.
* clear(): Removes all entries from the map.

Syntax:

HashMap<KeyType, ValueType> map = new HashMap<>();

Example using Hashmap to search:

import java.util.\*;

public class BookSearchExample {

public static void main(String[] args) {

// Create a HashMap to store book titles and their publication years

HashMap<String, Integer> books = new HashMap<>();

books.put("Java Basics", 2024);

books.put("Advanced Java", 2025);

books.put("Data Structures", 2023);

String searchBook = "Advanced Java";

if (books.containsKey(searchBook)) {

System.out.println("Publication year of '" + searchBook + "': " + books.get(searchBook));

} else {

System.out.println("Book '" + searchBook + "' not found in the collection.");

}

}

containsKey(): This method checks if the HashMap contains a key (book title in this case).

If the book exists, it retrieves the value (publication year) using get().

If the book does not exist, it prints a message indicating that the book was not found.