Computer Scientist in A Nutshell

Miskatul Anwar

June 28, 2024

Comming up ...

- **Maths**
  - Calculus
  - Differention & Integration
  - Linear Algebra
  - Probability
- Java
  - Setup
  - Compile & Run
  - Basics
- **Python**
  - Basics
  - Virtual Environment
    * Python venv
    * Pyenv
    * Anaconda
- **R**
  - Radiant
- **Data Analysis**
  - Numpy
  - Pandas
  - Matplotlib
- **Frameworks**
  - Pytorch
- **Machine Learning**
  - Supervised
  - Unsupervised
  - Reinforcement
- **Practice**
  - Kaggle
  - CIFAR 10
  - EMNIST
- **Neural Network**
  - Natural Language Processing(NLP)
  - Recurrent Neural Network(RNN)
- **Learn Generative AI**
  - GPT Documentation
  - ChatGPT Short Courses
  - The GPT store

Chapter 1

Maths

Maths will be the foundation what you'll need the most in this journey! The most important topics are …

1.1 Permutations & Combinations

Problems Related to Counting

1. Consider the word "Permutation". In how many ways can you arrange the letters,

    - **No, rule specified,**

    $$Answer : n_t = 2$$

    $$\frac{11!}{2!}$$

    - **If, The vowels stay together,**

    $$n_{total} = 7$$

    $$n_t = 2$$

    $$TheAnswer : \frac{7!}{2!} \cdot 5!$$

    - **If, The vowels don't stay together,**

    $$\frac{11!}{2!} - \frac{7!}{2!} \cdot 5!$$

    - **If the vowels stay at their place,**

    $$\frac{6!}{2!}$$

    - **If, The order (vowels) is same,**
    "euaio" if the order of the vowels. Let's consider them as a single word.

    $$\frac{11!}{2!5!}$$

    - **If the relative order is the same,**

    $$\frac{6!}{2!} \cdot 5!$$

2. How many numbers greater than 800 and less than 4000 can be made with the digits 0,1,2,4,5,7,8,9 ? (no number occuring more than once in the same number)
   Sol. firstly, let's consider the 801 - 999 range...(all the three digit numbers available in the limit)
   The first number will definitely be 8 or 9, so the possible ways to put the first number is 2. The other numbers will be put as follows.

   $$2 \cdot 7 \cdot 6$$

   secondly, consider the 1000 - 3999 range...(all the four digit numbers available in the limit)
   The first number can only be 1 or 2 as three is not given. So, the rest will be declining in the previously shown manner.

   $$2 \cdot 7 \cdot 6 \cdot 5$$

   Now, calculation of the possible ways is as follows:

   $$2 \cdot 7 \cdot 6 + 2 \cdot 7 \cdot 6 \cdot 5 = 504$$

3. Your Home to DU there's 3 ways avalable to go. DU to BUET there's only 2 available ways. How many ways are there to visit BUET from your home?
   (Yeah, ugh…you have'nt got a chance to read in BUET)
   Sol⇒ There are 6 different ways to visit BUET from your home.

4. there are five people in a vehicle. But there is only three sits available. In how many ways we can pick 3 people to have their sits?
   Sol.⇒ The answer is:

   $$5 \cdot 4 \cdot 3 = 60$$

   If we apply the formulae, pick 3 people from 5 and arrange them.

   $$^5P_3 = 60$$

5. How many even numbers of three digits can be formed from the digits 0,1,2,3,4,5,6 ?
   Sol.1 If the last digit is 0, 2, 4 or 6 , then it can be an even number. Also, we need to keep in mind that the first number can't be zero.

   $$6 \cdot 5 \cdot 1 + 5 \cdot 5 \cdot 3 = 105$$

6. The following problems have to do with cardinalities of sets of phone numbers of certain kinds.

   - How many 7-digit phone numbers are possible, assuming that the first digit can't be a 0 or a 1?
     Sol.
     To determine the number of possible 7-digit phone numbers where the first digit cannot be 0 or 1, we can break the problem into parts:

     - First Digit: The first digit has 8 possible choices (2 through 9).
     - Remaining Digits: Each of the remaining 6 digits can be any digit from 0 to 9, giving each position 10 possible choices.

     Therefore, the total number of 7-digit phone numbers can be calculated as follows:

     $$8 \cdot 10^6$$

     Here's the step-by-step calculation:
     There are 8 choices for the first digit. There are 10 choices for each of the remaining 6 digits. So, multiplying these together:

     $$8 \cdot 10^6 = 8,000,000$$

     Thus, there are 8,000,000 possible 7-digit phone numbers under the given constraints.

- Solve the above problem again, except now assume also that the phone number is not allowed to start with 911 (since this is reserved for emergency use, and it would not be desirable for the system to wait to see whether more digits were going to be dialed after someone has dialed 911). *Sol.*
  To solve this problem under the new constraints, we need to calculate the total number of valid 7-digit phone numbers that do not start with 911.
  Steps:
  (a) Total number of 7-digit phone numbers without additional constraints: As previously calculated, the total number of possible 7-digit phone numbers where the first digit can't be 0 or 1 is:
  $$8 \times 10^6 = 8,000,000$$
  (b) Calculate the number of invalid phone numbers that start with 911:
    - If the phone number starts with 911, the first three digits are fixed as 911.
    - The remaining 4 digits can each be any digit from 0 to 9, so there are 10 choices for each of the remaining 4 digits.

    Therefore, the number of phone numbers that start with 911 is:
  $$10 \times 10 \times 10 \times 10 = 10^4 = 10,000$$
- Subtract the number of invalid phone numbers from the total:

  Total valid phone numbers = Total possible phone numbers - Invalid phone numbers.
  $$8,000,000 - 10,000 = 7,990,000$$

Thus, the total number of valid 7-digit phone numbers, ensuring the first digit can't be 0 or 1 and the number does not start with 911, is 7,990,000.

7. How many different functions f : A → B are possible in total with |A| = m and |B| = n? (m and n are, of course, positive integers.)
   Sol.2

   To determine how many different functions f:A→B are possible with ∣A∣=m and ∣B∣=n, we can consider the following:

   - Definition of a Function: A function
   $$f : A \to B$$
     maps every element in set A to an element in set B.
   - Total Number of Functions:
     For each element in set A, there are n possible choices in set B. Since there are m elements in set A, each of the m elements can independently be mapped to any of the n elements in B.

   To find the total number of such functions, we use the principle of counting:
   $$n \times n \times n \times \cdots \times n (m \ times)$$
   This can be written as:
   $$n^m$$
   Conclusion Thus, the total number of different functions f:A→B with ∣A∣=m and ∣B∣=n is
   $$n^m$$

8. - Consider the grid shown below. Starting at a point, you can go one step up or one step to the right at each move (as long as any such move keeps you within the grid). This procedure is continued until the point C is reached from point A.

(a) How many different paths are there from A to C?
(b) How many different paths from A to C pass through B?

(We say that two paths are different whenever they are not identical.)

Let's analyze the problem of finding different paths in a 5x6 grid from point A to point C, and specifically how many of those paths pass through point B(2,3). In a 5x6 grid: Starting from point A(1,1) to point C(5,6), you need to make 4 steps up and 5 steps to the right. The total number of paths from A to C can be calculated using the combination formula, since each path consists of 4 up movements (U) and 5 right movements (R):

$$TotalPaths = 9C4$$

Calculating this:

$$\frac{4\times3\times2\times1}{9\times8\times7\times6} = 126$$

So, there are 126 different paths from A to C in the grid.

- To find paths from A to C that pass through B(2,3):
  From A to B(2,3), you need 1 step up and 2 steps to the right. From B(2,3) to C(5,6), you need 3 steps up and 3 steps to the right. So, the paths from A to C passing through B(2,3) involve:

  1 step up from A to B,
  2 steps to the right from A to B,
  3 steps up from B to C, and
  3 steps to the right from B to C.
  This gives a total of:

  $$3C1 \cdot 5C2 = 30$$

  - There are 126 different paths from A to C.
  - There are 30 different paths from A to C that pass through B(2,3).

9. As a part of the population in Silverland seeks to break free, higher ups of the mafia enterprise that controls Silverland has decided to quickly form a regional subcommittee with their operatives to teach them a lesson. The subcommittee is to consist of 3 hockeystick experts (H), 2 tech-savvy social media monitors (S), and 2 specializing in effective usage of machete (M). When a pool of 8 H, 3 S, and 6 M among their regular thugs in the territory are available, in how many different ways the subcommittee can be formed if

   - **(a) 2 of the H refuse to serve together:**
     Total ways without restriction:

     $$\binom{8}{3} \times \binom{3}{2} \times \binom{6}{2}$$

     Ways with 2 specific H serving together:

     $$\binom{6}{1} \times \binom{3}{2} \times \binom{6}{2}$$

Number of ways where the 2 H do not serve together:

$$\binom{8}{3} \times \binom{3}{2} \times \binom{6}{2} - \left(\binom{6}{1} \times \binom{3}{2} \times \binom{6}{2}\right)$$

- **(b) 2 of the M refuse to serve together:**
  Total ways without restriction:
  $$\binom{8}{3} \times \binom{3}{2} \times \binom{6}{2}$$
  Ways with 2 specific M serving together:
  $$\binom{8}{3} \times \binom{3}{2} \times \binom{4}{2}$$
  Number of ways where the 2 M do not serve together:
  $$\binom{8}{3} \times \binom{3}{2} \times \binom{6}{2} - \left(\binom{8}{3} \times \binom{3}{2} \times \binom{4}{2}\right)$$
- **(c) 1 S and 1 M refuse to serve together:**
  Total ways without restriction:
  $$\binom{8}{3} \times \binom{3}{2} \times \binom{6}{2}$$
  Ways with 1 specific S and 1 specific M serving together:
  $$\binom{8}{3} \times \binom{2}{1} \times \binom{5}{1} \times \binom{5}{2}$$
  Number of ways where 1 S and 1 M do not serve together:
  $$\binom{8}{3} \times \binom{3}{2} \times \binom{6}{2} - \left(\binom{8}{3} \times \binom{2}{1} \times \binom{5}{1} \times \binom{5}{2}\right)$$

1.2 Probability

$$P(H|E) = \frac{P(H) \cdot P(E|H)}{P(E)} \tag{1.1}$$

P(H) = Probability a hypothesis is true
P(H) = Probability of seeing the evidence if the hypothesis is true
P(E) = Probability of seeing the evidence
P(H|E) = Probability a hypothesis is true

The "|" stands for 'given' P(H|E) = P (Hypothesis given Evidence)
Also,

$$P(H|E) = \frac{P(H \cap E)}{P(E)} \tag{1.2}$$

$$P(E)P(H|E) = P(H \cap E) = P(H)P(E|H) \tag{1.3}$$

Chapter 2

Java

2.1 Encapsulation

Encapsulation is a fundamental concept in object-oriented programming (OOP) that binds together the data (variables) and the methods (functions) that manipulate that data within a single unit, such as a class or module. This unit is designed to hide its internal details from the outside world while exposing only necessary information through a controlled interface.
Why Encapsulation ?

1. **Data Hiding:** Encapsulation helps to hide the implementation details of an object from the outside world, making it difficult for other parts of the program to access or modify the data directly.

2. **Abstraction:** It provides abstraction by showing only the necessary information to the outside world while hiding the internal implementation details.

3. **Code Organization:** Encapsulation promotes good code organization and structure by grouping related data and methods together.

Ways to do that...

- Java Packages
- Access Modifiers
- Java Encapsulation
- Data Hiding
- The 'static' keyword

2.1.1 Java Packages

A package is simply a container that groups related types (Java classes, interfaces, enumerations, and annotations) An example package:

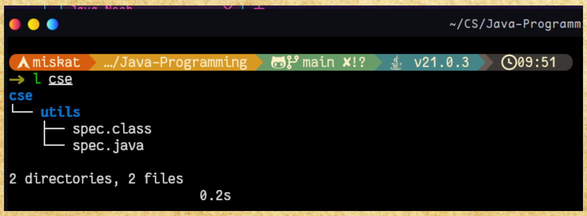

Figure 2.1: Linux Console

2.1.2 Access Modifiers

In Java, access modifiers are keywords that determine the visibility of a class, method, or variable to other parts of your program. They control who can see or interact with these elements.

- **public**: Accessible from anywhere in your program.
- **private**: Only accessible within the same class.
- **protected**: Accessible only within the same class and its subclasses.
- **default** (or no modifier): Accessible only within the same package.

Let's consider a simple example with a class 'Car':

```
public class Car {
    // public variable
    public String color = "red";

    // private method
    private void startEngine() {
        System.out.println("Vroom!");
    }

    // protected constructor
    protected Car(String make, String model) {
        this.make = make;
        this.model = model;
    }
}
```

public

- Use public classes when you want to expose a class or interface to the outside world.
- Examples:
  - A web service API that needs to be accessed by clients.
  - A utility class that provides common functionality.

```
public class MathUtils {
    public static int sum(int a, int b) {
        return a + b;
    }
}
```

private

- Use private variables or methods when you want to hide internal implementation details from the outside world.
- Examples:
  - A 'Car' class might have a private 'startEngine()' method that's not meant to be called directly by other classes.
  - An immutable class might have private setters and getters.

Access Modifiers	Same Class	Same Package Subclass	Same Package Non-subclass	Different Package Subclass	Different Package Non-subclass
default	Y	Y	Y	N	N
private	Y	Y	N	N	N
protected	Y	Y	Y	Y	N
public	Y	Y	Y	Y	Y

2.1.3 Ways to Encapsulation

Simple Class

A basic Java class that encapsulates a private integer variable `x` and provides a public method `getValue()` to retrieve its value:

```java
public class SimpleClass {
    private int x;

    public SimpleClass(int x) {
        this.x = x;
    }

    public int getValue() {
        return x;
    }
}
```

Bank Account

A Java class that encapsulates a bank account, with private fields for balance and account number, and public methods to deposit, withdraw, and get the balance:

```java
public class BankAccount {
    private double balance;
    private int accountNumber;

    public BankAccount(int accountNumber) {
        this.accountNumber = accountNumber;
        this.balance = 0.0;
    }

    public void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
        } else {
            System.out.println("Insufficient funds!");
        }
    }

    public double getBalance() {
        return balance;
    }
}
```

Student Record

A Java class that encapsulates a student's record, with private fields for name, age, and grades, and public methods to set and retrieve the information:

```java
public class StudentRecord {
    private String name;
    private int age;
    private double[] grades;

    public StudentRecord(String name, int age) {
        this.name = name;
        this.age = age;
        this.grades = new double[5];
    }

    public void setGrade(int index, double grade) {
        grades[index] = grade;
    }

    public double getAverageGrade() {
        double sum = 0.0;
        for (double grade : grades) {
            sum += grade;
        }
        return sum / grades.length;
    }
}
```

Use Cases

1. **Security:** Encapsulation ensures that sensitive data is protected from unauthorized access.
2. **Code Reusability:** By encapsulating related data and methods, you can reuse the code in different parts of your program.
3. **Improved Code Organization:** Encapsulation helps to organize your code in a logical and structured way.

Applications

1. **Database Management Systems:** Database management systems like MySQL or Oracle use encapsulation to hide their internal implementation details while providing a controlled interface for data manipulation.
2. **Financial Applications:** Banking and financial applications, such as accounting software, use encapsulation to protect sensitive customer data and ensure secure transactions.
3. **Gaming:** Games often use encapsulation to hide the internal workings of game mechanics, levels, or characters, making it easier to modify or extend the game without affecting other parts.

In summary, encapsulation is a fundamental concept in object-oriented programming that helps to organize code, protect sensitive data, and improve code reusability.

2.1.4 Data Hiding

Data hiding is a principle of encapsulation that conceals the internal representation of an object's state (data) from external objects, making it difficult for other parts of the program to access or modify the data directly. This helps to:

1. **Prevent Accidental Changes:** By hiding the internal state, you prevent accidental changes to the data from outside the class.

2. **Control Access:** You can control access to the data by providing methods (getters and setters) that allow only authorized objects to modify the data.

3. **Improve Code Quality:** Data hiding encourages good coding practices, such as using meaningful variable names and encapsulating related data.

How to Apply Data Hiding

1. **Make Data Private:** Declare the data variables as private within the class:

```java
public class Example {
    private int x;
    // ...
}
```

2. **Use Getters and Setters:** Provide public methods (getters and setters) that allow controlled access to the data:

```java
public class Example {
    private int x;

    public int getX() {
        return x;
    }

    public void setX(int value) {
        x = value; // validate or transform the value if needed
    }
}
```

3. **Avoid Direct Access:** Refrain from accessing the data directly from outside the class:

```java
public class Example {
    private int x;

    public void someMethod() {
        System.out.println(x); // bad practice, avoid direct access
    }
}
```

Best Practices

1. **Use Meaningful Variable Names:** Use descriptive names for your variables to make the code more readable.

2. **Limit Access Modifiers:** Limit access modifiers (public, private, protected) to only what's necessary.

3. **Validate Input:** Validate input data before modifying it to ensure integrity and prevent errors.

4. **Use Immutable Objects:** Consider using immutable objects when possible, especially for sensitive or critical data.

Real-World Examples

1. **Bank Account:** A bank account class should hide its internal balance and only provide methods for depositing, withdrawing, and getting the balance.
2. **Person Information:** A person information class might hide their personal details (name, age, address) and provide getter methods to access this information.

By applying data hiding principles, you can create more robust, maintainable, and secure code that is easier to understand and modify over time.

2.1.5 The ‘static‘ keyword

The ‘static‘ keyword in Java is used to declare static variables or methods that belong to a class rather than an instance of the class. Here are some key uses of the ‘static‘ keyword:

1. **Static Variables:** You can use ‘static‘ variables to store values that don’t change throughout the program’s execution.

```
    public class MyMath {
    public static int MAX_VALUE = 100;

    public static void main(String[] args) {
        System.out.println(MyMath.MAX_VALUE); // prints 100
    }
}
```

2. **Static Methods:** You can use ‘static‘ methods to perform operations that don’t depend on instance-specific data.

```
public class MathUtil {
    public static double sqrt(double num) {
        return Math.sqrt(num);
    }

    public static void main(String[] args) {
        System.out.println(MathUtil.sqrt(4)); // prints 2.0
    }
}
```

3. **Singleton Pattern:** You can use ‘static‘ variables to implement the Singleton pattern, where only one instance of a class exists throughout the program’s execution.

```
        public class Logger {
    private static Logger instance = null;

    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }

    // ...
}
```

In this example, the ‘Logger‘ class uses a ‘static‘ variable to keep track of the single instance.

4. **Utility Methods:** You can use ‘static‘ methods as utility methods that can be called from anywhere in your code without creating an instance of the class.

   Example:

```
public class StringHelper {
    public static boolean isEmpty(String str) {
        return str == null || str.trim().length() == 0;
    }

    public static void main(String[] args) {
        System.out.println(StringHelper.isEmpty("")); // prints true
        System.out.println(StringHelper.isEmpty("hello")); // prints false
    }
}
```

   In this example, the ‘isEmpty‘ method is a ‘static‘ utility method that can be called from anywhere in your code.

   These are just a few examples of how you can use the ‘static‘ keyword in Java. Remember to use it wisely and only when necessary!

2.1.6 File Handling

Copying Contents from `INPUT.DAT` to `OUTPUT.DAT`

```
public class copyPaste {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader("input.dat"));
        String str;
        if ((str = br.readLine()) != null) {
            System.out.println(str);
        }
        br.close();
        BufferedWriter bw = new BufferedWriter(new FileWriter("output.dat"));
        bw.append(str);
        bw.close();
        return;
    }
}
```

Using `DataOutputStream` and `DataInputStream`

```
import java.io.*;

public class file1 {
    public static void main(String[] args) throws IOException {
        FileOutputStream fout = new FileOutputStream("Test.txt");
        DataOutputStream out = new DataOutputStream(fout);
        out.writeDouble(98.6);
        out.writeInt(1000);
        out.writeBoolean(true);
        out.close();

        FileInputStream fin = new FileInputStream("Test.txt");
        DataInputStream in = new DataInputStream(fin);
        double d = in.readDouble();
        int i = in.readInt();
        boolean b = in.readBoolean();
```

```
        System.out.println("Here are the values: " + d + " " + i + " " + b);
        in.close();
    }
}
```

Reading from `output.txt`

```java
import java.io.*;

public class file3 {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader("output.txt"));
        String str;
        if ((str = br.readLine()) != null) {
            System.out.println(str);
        }
        br.close();
        return;
    }
}
```

2.1.7 `final` keyword in Java

Type	Description
Final Variable	Variable with 'final' keyword cannot be assigned again, similar to 'const' in C. Once initialized, its value cannot be changed.
Final Method	Method with 'final' keyword cannot be overridden by its subclasses. This ensures that the method's implementation in the superclass cannot be changed by subclasses.
Final Class	Class with 'final' keyword cannot be extended or inherited from. It serves as a final implementation that cannot be subclassed.

2.1.8 Java Inheritance

01 (Method Overriding)

```java
// Parent class (Shape)
class Shape {
    void draw() {
        System.out.println("Drawing a shape");
    }
}

// Child class (Circle) that extends the parent class (Shape)
class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a circle");
    }
}

// Child class (Rectangle) that extends the parent class (Shape)
class Rectangle extends Shape {
    @Override
    void draw() {
```

```java
        System.out.println("Drawing a rectangle");
    }
}

public class InheritanceExample1 {
    public static void main(String[] args) {
        // Create objects of the child classes
        Circle circle = new Circle();
        Rectangle rectangle = new Rectangle();

        // Use polymorphism to call the draw method on these objects
        Shape shape1 = circle;
        Shape shape2 = rectangle;

        shape1.draw();  // Output: Drawing a circle
        shape2.draw();  // Output: Drawing a rectangle
    }
}
```

02

```java
// Parent class (Vehicle)
class Vehicle {
    void start() {
        System.out.println("Starting a vehicle");
    }
}

// Child class (Car) that extends the parent class (Vehicle)
class Car extends Vehicle {
    void accelerate() {
        System.out.println("Accelerating a car");
    }
}

// Child class (Motorcycle) that extends the parent class (Vehicle)
class Motorcycle extends Vehicle {
    void revEngine() {
        System.out.println("Revving a motorcycle engine");
    }
}

public class InheritanceExample2 {
    public static void main(String[] args) {
        // Create objects of the child classes
        Car car = new Car();
        Motorcycle motorcycle = new Motorcycle();

        // Use polymorphism to call the start method on these objects
        Vehicle vehicle1 = car;
        Vehicle vehicle2 = motorcycle;

        vehicle1.start();  // Output: Starting a vehicle
        vehicle2.start();  // Output: Starting a vehicle
    }
}
```

Using `super` keyword in Inheritance

```java
class Animal {
    void sound() {
        System.out.println("The animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        super.sound();
        System.out.println("Woof!");
    }
}

public class SuperKeywordExample {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound();  // Output: The animal makes a sound, Woof!
    }
}
```

Using `this` keyword in Class

```java
class Complex {
    int a, b;

    public Complex(int a, int b) {
        this.a = a;
        this.b = b;
    }

    void print() {
        System.out.println(a + " + " + b + "i");
    }

    Complex add(Complex num2) {
        Complex newnum = new Complex(a + num2.a, b + num2.b);
        return newnum;
    }
}

public class ThisKeywordExample {
    public static void main(String[] args) {
        Complex num1 = new Complex(11, 15);
        Complex num2 = new Complex(11, 15);
        num1.print();
        Complex res = num1.add(num2);
        res.print();
    }
}
```

2.1.9 Comparison: this vs. super

this	super
this is an implicit reference variable used to represent the current class. It can invoke methods of the current class and refer to its instance and static variables. It's also used to return and pass as an argument in the context of a current class object.	super is an implicit reference variable used to represent the immediate parent class. It can invoke methods of the immediate parent class, refer to its instance and static variables, and invoke its constructor.

2.1.10 Constructors

01

```java
class Innerconstructors {
    int a, b;

    public Innerconstructors() {
        a = 4;
        b = 7;
    }

    void print() {
        System.out.println(a + b);
    }
}

public class ConstructorsExample {
    public static void main(String[] args) {
        Innerconstructors num1 = new Innerconstructors();
        num1.print();
    }
}
```

02

```java
class Innerconstructors {
    int a, b;

    public Innerconstructors() {
        a = 4;
        b = 7;
        System.out.println("New object created !");
    }

    void print() {
        System.out.println(a + b);
    }
}

public class ConstructorsExample {
    public static void main(String[] args) {
        Innerconstructors num1 = new Innerconstructors();
    }
}
```

03

```java
class Innerconstructors {
    int a, b;
```

```java

    public Innerconstructors(int real, int imaginary) {
        a = real;
        b = imaginary;
    }

    void print() {
        System.out.println(a + b);
    }
}

public class ConstructorsExample {
    public static void main(String[] args) {
        Innerconstructors num1 = new Innerconstructors(11, 15);
    }
}
```

04

```java
class Innerconstructors {
    int a, b;

    public Innerconstructors(int real, int imaginary) {
        a = real;
        b = imaginary;
    }

    void print() {
        System.out.println(a + " + " + b + "i");
    }
}

public class ConstructorsExample {
    public static void main(String[] args) {
        Innerconstructors num1 = new Innerconstructors(11, 15);
        num1.print();
    }
}
```

Method Overloading

```java
public class MethodOverloadingExample {
    void greetings() {
        System.out.println("Hello, Good Morning!");
    }

    void greetings(String name) {
        System.out.println("Hello " + name + ", Good Morning!");
    }
}
```

2.1.11 Possible ways to iterate

iterateInMap.java

```java
import java.util.ArrayList;
import java.util.HashMap;
```

```java
import java.util.Map;

public class IterateInMap {
    public static void main(String[] args) {
        System.out.println("Ways to iterate in Map:");

        Map<String, Integer> tm = Map.of(
            "one", 1,
            "two", 2,
            "three", 3);

        // Method I
        for (Map.Entry<String, Integer> entry : tm.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }

        // Method II
        for (String key : tm.keySet()) {
            System.out.println(key + " -> " + tm.get(key));
        }

        // Method III
        for (Integer val : tm.values()) {
            System.out.println("Value: " + val);
        }

        // Method IV
        tm.forEach((key, value) -> System.out.println(key + " + " + value));
    }
}
```

2.1.12 StringBuilders.java

```java
public class StringBuilders {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder();
        sb.append("Hello");
        sb.append(", ");
        sb.append("World");

        System.out.println(sb.charAt(0));
        System.out.println(sb);

        for (int i = 0; i < sb.length(); i++) {
            sb.setCharAt(i, (char) (sb.charAt(i) + 1));
        }

        System.out.println(sb);
    }
}
```

2.1.13 The Map framework

TreeMap example

```java
import java.util.TreeMap;

```

```java
public class TreeMapExample {
    public static void main(String[] args) {
        TreeMap<String, Integer> hp = new TreeMap<>();
        hp.put("Bus", 2);
        hp.put("Car", 1);
        hp.put("Zeep", 1);
        hp.put("Plane", 1);

        System.out.println(hp);

        if (hp.containsKey("Bus")) {
            hp.putIfAbsent("Bus", 2);
        }

        if (hp.containsValue(2)) {
            System.out.println("Duo");
        }

        for (HashMap.Entry<String, Integer> e : hp.entrySet()) {
            System.out.println(e);
        }

        for (String key : hp.keySet()) {
            System.out.println(key);
        }
    }
}
```

`HashMap` **example**

```java
import java.util.HashMap;

public class HashMapExample {
    public static void main(String[] args) {
        HashMap<String, Integer> hp = new HashMap<>();
        hp.put("Bus", 2);
        hp.put("Car", 1);
        hp.put("Zeep", 1);
        hp.put("Plane", 1);

        System.out.println(hp);

        if (hp.containsKey("Bus")) {
            hp.putIfAbsent("Bus", 2);
        }

        if (hp.containsValue(2)) {
            System.out.println("Duo");
        }

        for (HashMap.Entry<String, Integer> e : hp.entrySet()) {
            System.out.println(e);
        }

        for (String key : hp.keySet()) {
            System.out.println(key);
        }
    }
```

```
}
```

2.1.14 OMG ArrayList

ArrayList example

```java
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<>();
        list.add(10);
        list.add(20);
        list.add(30);
        list.add(0, 30);

        for (Integer i : list) {
            System.out.println(i);
        }

        if (list.contains(10)) {
            System.out.println(list.toString());
        }
    }
}
```

2.1.15 Binary Search

binarySearch1.java

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;

public class BinarySearchExample {
    public static void main(String[] args) {
        ArrayList<Integer> num = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 2, 1, 9, 8, 7,
        ↪  2));
        Collections.sort(num);
        int target = 7;

        System.out.println("The target 7 found at index: " + Collections.binarySearch(num,
        ↪  target));
        System.out.println("Max : " + Collections.min(num));
        System.out.println("Min : " + Collections.max(num));

        Collections.sort(num, Collections.reverseOrder());
    }
}
```

2.1.16 Hot Set

HashSet example

```java
import java.util.HashSet;

```

```java
public class HashSetExample {
    public static void main(String[] args) {
        HashSet<Integer> s = new HashSet<>();
        s.add(4);
        s.add(1);
        s.add(1);
        s.add(2);
        s.add(3);

        System.out.println(s);
        System.out.println(s.contains(3));

        s.clear();
        System.out.println(s);
    }
}
```

`TreeSet` **example**

```java
import java.util.TreeSet;

public class TreeSetExample {
    public static void main(String[] args) {
        TreeSet<Integer> set = new TreeSet<>();
        set.add(3);
        set.add(1);
        set.add(5);
        set.add(9);

        System.out.println(set);
    }
}
```

`LinkedHashSet` **example**

```java
import java.util.LinkedHashSet;
import java.util.Iterator;

public class LinkedHashSetExample {
    public static void main(String[] args) {
        LinkedHashSet<Integer> s = new LinkedHashSet<>();
        s.add(4);
        s.add(2);
        s.add(1);

        System.out.println(s);

        Iterator<Integer> it = s.iterator();
    }
}
```

2.1.17 It's me bro :) `Stack`

`Stack` **example**

```java
import java.util.Stack;

```

```java
public class StackExample {
    public static void main(String[] args) {
        Stack<String> animals = new Stack<>();
        animals.push("Lion");
        animals.push("Dog");
        animals.push("Cat");
        animals.push("Rat");
        animals.push("Bat");

        System.out.println(animals);
        System.out.println(animals.peek()); // LIFO

        animals.pop();
        System.out.println(animals.toString());
    }
}
```

2.1.18 I'd like to be dynamic! Heap

Queue example

```java
import java.util.LinkedList;

public class QueueExample {
    public static void main(String[] args) {
        LinkedList<Integer> queue = new LinkedList<>();
        queue.offer(12);
        queue.offer(24);
        queue.offer(36);

        System.out.println(queue);
        System.out.println(queue.peek());
        System.out.println(queue.element());

        queue.poll();
        System.out.println(queue);
    }
}
```

Deque example

```java
import java.util.ArrayDeque;

public class DequeExample {
    public static void main(String[] args) {
        ArrayDeque<Integer> adq = new ArrayDeque<>();
        adq.offer(1);
        adq.offer(2);
        adq.offer(3);
        adq.offer(5);

        System.out.println(adq.peekFirst());
        System.out.println(adq.peekLast());
        System.out.println(adq);

        adq.poll();
        System.out.println(adq);
```

```

        adq.pollFirst();
        System.out.println(adq);

        adq.pollLast();
        System.out.println(adq);
    }
}
```

PriorityQueue example

```
import java.util.PriorityQueue;

public class PriorityQueueExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        pq.offer(40);
        pq.offer(30);
        pq.offer(10);
        pq.offer(20);

        System.out.println(pq);
        System.out.println(pq.peek());
    }
}
```

Chapter 3

Python

Just try to learn the basics of python and that's it. Don't try to deep dive. In 2024, learning something has become really easy. Understand the basics, keep your eye on the goal.

- freecodecamp.org free course
- Tech With Tim Playlist

Chapter 4

Data Analysis

Some important *Python* libraries for 'Data Analysis'...

- Numpy
- Pandas
- Matplotlib

4.1 Matplotlib

4.1.1 Basic Plotting

```

from numpy import *
from pylab import *

x = arange(-10, 10, 0.5)
y = x**2
plot(x, y, "r.")
show()
```

4.1.2 Scatter Plot

01

```

import matplotlib.pyplot as plt
import numpy as np

ax = plt.axes(projection="3d")

x_data = np.arange(0, 10, 0.1)
y_data = np.arange(0, 10, 0.1)
z_data = x_data * y_data

# ax.scatter(x_data, y_data, z_data, alpha=0.5, color="black")
ax.plot(x_data, y_data, z_data)

plt.show()
```

02

```python
import matplotlib.pyplot as plt
import numpy as np

ax = plt.axes(projection="3d")

x_data = np.arange(0, 100, 10)
y_data = np.arange(0, 100, 10)
z_data = np.arange(0, 100, 10)

ax.scatter(x_data, y_data, z_data, color="red", marker="v", alpha=0.9)

plt.show()
```

4.1.3 Histogram

```python
import matplotlib.pyplot as plt
# fig = plt.figure()
# ax = fig.add_axes([0, 0, 1, 1])
categories = ["A", "B", "C", "D"]
values = [10, 23, 17, 5]
ax = plt.bar(categories, values)
plt.xlabel("marks", font="Maple Mono")
plt.ylabel("marks", font="Maple Mono")
plt.title("CSE 116 Marks", font="Maple Mono")
plt.legend("CSE116", loc="lower left")
# ax.bar(langs, students)
plt.show()
```

4.1.4 Graphs

Mathematical Equations

$$f(x) = \sin(x) \tag{4.1}$$

```python

import matplotlib.pyplot as plt
import numpy as np

ax = plt.axes()

# Plot for f(x) = sin(x)
# let's consider, y = f(x) = sin(x)

x_data = np.arange(0, 20, 0.1)
y_data = np.sin(x_data)

ax.set_title("f(x) = sin(x)", font="Maple Mono", color="red")
ax.set_xlabel("sin(x)")
ax.set_ylabel("sin(x)")
ax.plot(x_data, y_data)
plt.show()

```

4.1.5 3D Plotting

Shpere

```
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
from matplotlib import pyplot as plt

fig = plt.figure(figsize=(6, 6))
ax = fig.add_subplot(111, projection="3d")

# Create a sphere using numpy
u = np.linspace(0, 2 * np.pi, 100)
v = np.linspace(0, np.pi, 100)
x = np.outer(np.cos(v), np.sin(u))
y = np.outer(np.sin(v), np.sin(u))
z = np.outer(np.ones(np.size(u)), np.cos(u))
ax.plot_surface(x, y, z, color="r", rstride=4, cstride=4)

plt.show()

```

4.1.6 Surface Plot

```

import matplotlib.pyplot as plt
import numpy as np

myplot = plt.axes(projection="3d")

x_data = np.arange(0, 20, 0.1)
y_data = np.arange(0, 20, 0.1)
X, Y = np.meshgrid(x_data, y_data)
Z = X * Y

myplot.plot_surface(X, Y, Z)
plt.show()

```

Chapter 5

Practice

Start solving problems: kaggle
Also watchout,

- CIFAR 10
- EMNIST

Chapter 6

Learn Generative AI

- Deep Learning Full Course
- ChatGPT Documentation
- ChatGPT Short Courses
- GPT Store