Computer Scientist in A Nutshell

Miskatul Anwar

June 26, 2024

Comming up …

- **Maths**
  - Calculus
  - Differention & Integration
  - Linear Algebra
  - Probability
- Java
  - Setup
  - 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 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 ?

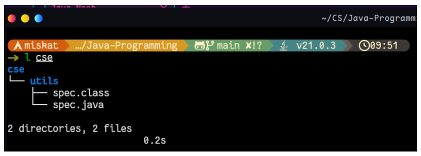
1. **Data Hiding:** Encapsulation helps to hide the implementation details of an object from the outside world, making it difficult for other parts of the program to access or modify the data directly.

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



```
~/CS/Java-Programm
miskat  …/Java-Programming  main ✘!?  v21.0.3  09:51
→ l cse
cse
└── utils
    ├── spec.class
    └── spec.java

2 directories, 2 files
                  0.2s
```

Figure 2.1: Linux Console

2.1.1 Java Packages

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

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 sub-classes.
- **default** (or no modifier): Accessible only within the same package.

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

```java
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 S
default	Y	Y	Y	N
private	Y	Y	N	N
protected	Y	Y	Y	Y
public	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:

```
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:

```
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:

```
    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`

```
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() {
        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

```
// 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

```
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

```
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

```
class Innerconstructors {
    int a, b;

    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

```
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

```
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

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

```
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

```
import java.util.TreeMap;

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

```
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

```
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

```
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,
        Collections.sort(num);
        int target = 7;

        System.out.println("The target 7 found at index: " + Collections.binarySearch(
        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**

```
import java.util.HashSet;

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**

```
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**

```
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**

```
import java.util.Stack;

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

```
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

```
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());
    }
}
```