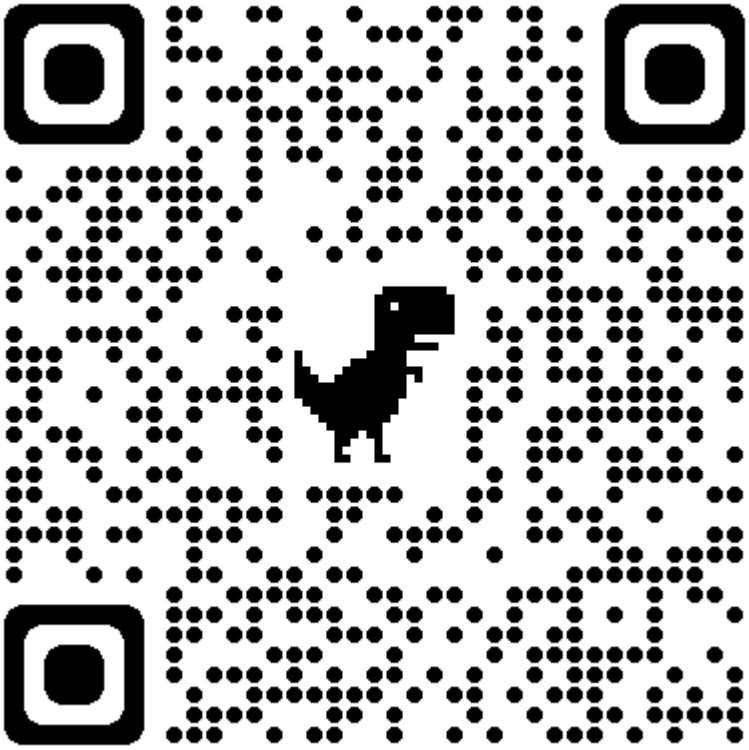
2190102 ADV COMP PROG Cheat Sheet - M

By @WasinUddy (✉ ws@prometheuzdy.cloud)



Download Example Code from Laboratory Session!!!!

Do not forget to ⭐ it

What is OOP?

Object-oriented programming (OOP) is a programming paradigm that uses "objects" to design applications and software. These objects represent real-world entities and can both store data and perform actions.

Class in Java

In Java, a class is a blueprint or template for creating objects. It defines the structure and behavior of objects of that class. Each class can have attributes (also called fields) and methods.

- ***Attributes (Fields):*** Attributes are variables declared within a class to store data. They represent the state or properties of objects created from the class. Attributes define what an object "has." For example, if you are creating a class to represent a Person, attributes could include name, age, and address.

- ***Methods:*** Methods are functions defined within a class that perform actions or operations on the class's attributes or provide some functionality. Methods define what an object "does." For example, a Person class might have methods like setName, getAge, and printDetails.

Example Implementation of Class Person

```java
public class Person {
    // Attributes (fields)
    public String name;
    public int age;

    // Constructor (a special method to initialize objects)
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getter method for name
    public String getName() {
        return name;
    }

    // Setter method for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter method for age
    public int getAge() {
        return age;
    }

    // Setter method for age
    public void setAge(int age) {
        this.age = age;
    }

    // Method to print person's details
    public void printDetails() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}
```

- ***Attributes (name and age):*** These are private fields, meaning they can only be accessed within the class itself. We use getter and setter methods to access and modify these attributes from outside the class.

- ***Constructor:*** The constructor is a special method used to initialize objects of the class when they are created. It takes parameters (in this case, ***name*** and ***age***) and assigns them to the object's attributes.

- ***Getter and Setter methods:*** These methods allow us to get and set the values of the name and age attributes, respectively. They provide controlled access to the attributes.

- ***printDetails method:*** This method is used to print the details (***name*** and ***age***) of a ***Person*** object.

Example Usage of Class Person

```
public class App {
    public static void main(String[] args) {
        // Create a Person object
        Person person1 = new Person("Alice", 25);

        // Access attributes and methods
        System.out.println("Name: " + person1.getName());
        System.out.println("Age: " + person1.getAge());
        person1.printDetails();

        // Update attributes using setters
        person1.setName("Bob");
        person1.setAge(30);
        person1.printDetails();
    }
}
```

In this ***App*** class, we create a ***Person*** object, access its attributes using getter methods, print details, and update the attributes using setter methods.

Encapsulation

Encapsulation refers to the concept of bundling the data (attributes or fields) and methods (functions) that operate on the data into a single unit, known as a class. In encapsulation, the internal state of an object is hidden from outside access, and access to that state is controlled through methods. This helps in data protection and maintaining the integrity of an object's state.

Key Concepts of Encapsulation

- ***Private Access Modifier***: In Java, encapsulation is primarily achieved by using the ***private*** access modifier for class fields (attributes). When a field is declared as ***private***, it can only be accessed within the class where it's defined.

- ***Getter and Setter Methods***: To provide controlled access to the private fields, getter and setter methods are used. Getter methods allow you to retrieve the value of a field, and setter methods allow you to modify the value of a field. By controlling access through these methods, you can implement validation logic, access control, and maintain the integrity of the object's data.

Benefits of Encapsulation

- ***Data Hiding:*** Encapsulation hides the internal details of an object's implementation, which reduces complexity and makes it easier to change the internal representation without affecting other parts of the code.

- ***Controlled Access:*** With getter and setter methods, you can control how external code interacts with an object's data. This allows you to validate input, ensure data consistency, and implement access control rules.

- ***Flexibility:*** Encapsulation allows you to change the internal implementation of a class without affecting the code that uses the class. This promotes code maintenance and evolution.

Example Implementation of Encapsulation on Class Person

```java
public class Person {
    // Attributes (fields)
    private String name;
    private int age;

    // Constructor (a special method to initialize objects)
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getter method for name
    public String getName() {
        return name;
    }

    // Setter method for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter method for age
    public int getAge() {
        return age;
    }

    // Setter method for age
    public void setAge(int age) {
        this.age = age;
    }

    // Method to print person's details
    public void printDetails() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}
```

- The ***name*** and ***age*** fields are declared as ***private***, making them inaccessible from outside the class.
- Getter methods (***getName*** and ***getAge***) allow external code to retrieve the values of ***name*** and ***age***.
- Setter methods (***setName*** and ***setAge***) provide controlled access to modify the values of ***name*** and ***age***. The ***setAge*** method includes validation logic to ensure that the age is non-negative.

Inheritance

Inheritance is one of the key principles of Object-Oriented Programming (OOP) and allows you to create new classes that are based on existing classes, inheriting their attributes and methods.

Key Concepts of Inheritance

- ***Extending a Class***: To create a subclass that inherits from a superclass, you use the ***extends*** keyword in the class declaration.

- ***Access to Superclass Members:*** A subclass has access to all the public and protected members (fields and methods) of its superclass. However, it cannot directly access private members of the superclass.

- ***Method Overriding:*** A subclass can provide its own implementation for a method that is already defined in its superclass. This is known as method overriding.

Inheritance in Java

Inheritance is a mechanism in Java that allows one class to inherit the properties (fields) and behaviors (methods) of another class. The class that is being inherited from is called the superclass or base class, and the class that inherits from it is called the subclass or derived class.

Example Implementation on Inheritance Dog, Corgi, Husky Classes

a simple example with a ***Dog*** superclass and two subclasses, ***Corgi*** and ***Husky***. The ***Dog*** class will contain common properties and methods that are shared by all dogs, while the subclasses will add specific attributes and behaviors.

Example of Super Class

```
public class Dog {
    private String name;
    private int age;

    public Dog(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public void bark() {
        System.out.println(name + " is barking!");
    }
}
```

The ***Dog*** class is the superclass that contains attributes and methods common to all dogs. It has a constructor, getter methods for ***name*** and ***age***, and a ***bark*** method.

Example of Sub Classes

```
public class Corgi extends Dog {
    public Corgi(String name, int age) {
        super(name, age); // Call the superclass constructor
    }

    // Additional method specific to Corgi
    public void shortLegs() {
        System.out.println(getName() + " has short legs!");
    }
}
```

```
public class Husky extends Dog {
    public Husky(String name, int age) {
        super(name, age); // Call the superclass constructor
    }

    // Additional method specific to Husky
    public void thickFur() {
        System.out.println(getName() + " has thick fur!");
    }
}
```

- The ***Corgi*** and ***Husky*** classes are subclasses of ***Dog***. They use the ***extends*** keyword to inherit from the ***Dog*** class and call the superclass constructor using ***super(name, age)*** .
- Each subclass adds its own specific method: ***shortLegs*** for ***Corgi*** and ***thickFur*** for ***Husky***.

Example Usage

```
public class App {
    public static void main(String[] args) {
        Corgi corgi = new Corgi("Buddy", 3);
        Husky husky = new Husky("Luna", 2);

        corgi.bark();
        corgi.shortLegs();

        husky.bark();
        husky.thickFur();
    }
}
```

In this example, we create instances of ***Corgi*** and ***Husky*** and call their methods. The subclasses inherit the ***getName***, ***getAge***, and ***bark*** methods from the ***Dog*** superclass and add their specific behaviors. This demonstrates how inheritance allows you to create a hierarchy of classes with shared and specialized characteristics.

Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common base class. It enables you to write code that can work with objects of multiple types, promoting flexibility and extensibility in your programs.

Benefits of Polymorphism

- ***Code Reusability***: Polymorphism allows you to write generic code that can work with a variety of objects, promoting code reuse.

- ***Flexibility***: It makes your code more flexible and adaptable to changes. You can easily extend your system by adding new subclasses without modifying existing code.

- ***Maintenance***: Polymorphism simplifies code maintenance. You can make changes in the superclass or add new subclasses without affecting the rest of the codebase.

- ***Enhanced Readability***: Polymorphic code is often more readable and self-explanatory, as it abstracts away specific object types and focuses on their common behaviors.

Polymorphism in Java

In Java, polymorphism can be realized in several ways, but one of the most common ways is through ***method overriding***.

Example Implementation of Polymorphism to represent Automobile

Example of Base Class

```
public class Automobile {
    void start() {
        System.out.println("Automobile is starting.");
    }
}
```

Example of Sub Classes Car and Motorcycle inherit and override method

```
public class Car extends Automobil {
    public Car () {

    }

    @Override
    public void start() {
        System.out.println("Car is starting");
    }
}
```

```
public class Motorcycle extends Automobile {
    public Motorcycle() {

    }

    @Override
    public void start() {
        System.out.println("Motorcycle is starting.");
    }
}
```

Two subclasses, ***Car*** and ***Motorcycle***, inherit from ***Automobile*** and override the ***start*** method with their specific implementations.

Abstraction

Abstraction involves simplifying complex reality by modeling classes based on the essential properties and behaviors while hiding the unnecessary details.

Benefits of Abstraction

- ***Simplicity and Ease of Use***: Abstraction simplifies complex systems or objects, making them easier to understand and use. Users can interact with abstracted interfaces without needing to know the inner workings of the underlying system.

- ***Code Reusability***: Abstraction promotes code reusability by allowing you to define common behaviors and interfaces. These abstractions can be reused in various parts of your code or in different projects.

- ***Maintenance and Evolution***: Abstraction makes it easier to maintain and evolve your code. Changes to the internal implementation of a system can be made without affecting code that uses the abstracted interface.

- ***Polymorphism***: Abstraction enables polymorphism, allowing you to treat objects of different classes as objects of a common base class. This promotes flexibility in your code and the ability to work with objects generically.

Example Implementation of Abstraction to represent TV Remote

Example of Abstract Class

```
public abstract class RemoteControl {
    private boolean poweredOn;
    private int currentChannel;
    private int currentVolume;

    public RemoteControl() {
        poweredOn = false;
        currentChannel = 0;
        currentVolume = 0;
    }

    // Turn the remote control on
    public void turnOn() {
        poweredOn = true;
        System.out.println("Remote control is now ON.");
    }

    // Turn the remote control off
    public void turnOff() {
        poweredOn = false;
        System.out.println("Remote control is now OFF.");
    }

    // Abstract method to change the channel (to be implemented by subclasses)
    public abstract void changeChannel(int channel);

    // Abstract method to adjust the volume (to be implemented by subclasses)
    public abstract void adjustVolume(int volume);
}
```

RemoteControl is an abstract class that represents the abstraction of a remote control. It includes common attributes like ***poweredOn***, ***currentChannel***, and ***currentVolume***, along with methods like ***turnOn, turnOff, changeChannel***, and ***adjustVolume*** which need to be implemented.

Example of Sub Class Implementing Abstract Method of Abstract Class

```
public class TVRemote extends RemoteControl {
    @Override
    public void changeChannel(int channel) {
        if (isPoweredOn()) {
            setCurrentChannel(channel);
            System.out.println("TV channel changed to channel " + channel);
        } else {
            System.out.println("Turn on the remote control first.");
        }
    }

    @Override
    public void adjustVolume(int volume) {
        if (isPoweredOn()) {
            setCurrentVolume(getCurrentVolume() + volume);
            System.out.println("Volume adjusted to " + getCurrentVolume());
        } else {
            System.out.println("Turn on the remote control first.");
        }
    }
}
```

TVRemote is a concrete subclass that extends ***RemoteControl***. It implements the abstract methods ***changeChannel*** and ***adjustVolume*** specific to a TV remote.

Example Usage

```
public class App {
    public static void main(String[] args) {
        TVRemote remote = new TVRemote();

        remote.turnOn();
        remote.changeChannel(5);
        remote.adjustVolume(10);

        remote.turnOff();
        remote.changeChannel(3);
        remote.adjustVolume(-5);
    }
}
```

In the ***App*** class, we create an instance of ***TVRemote*** and use it to control the TV. We don't need to know the inner details of how the remote control works; we interact with its abstracted interface.

Interface

interface is a blueprint for a class. It defines a contract of methods that a class must implement. Unlike classes, interfaces cannot contain instance variables (fields) or concrete method implementations. Instead, they specify the method signatures that implementing classes must provide.

Key Concepts of Inheritance

- ***Method Signatures***: Interfaces define method signatures without specifying the implementation. These methods are implicitly public and abstract (no need to use the ***public*** or ***abstract*** modifiers).

- ***Multiple Inheritance***: Java supports multiple inheritance through interfaces. A class can implement multiple interfaces, which is useful for achieving polymorphism and code reusability.

- ***Implementation by Classes***: A class that implements an interface must provide concrete implementations (method bodies) for all the methods declared in that interface.

- ***"implements" Keyword***: To declare that a class implements an interface, you use the ***implements*** keyword in the class declaration.

Benefits of Interface

- ***Multiple Inheritance***: Interfaces allow a class to inherit from multiple interfaces, enabling code reuse and flexibility.

- ***Method Contracts***: Interfaces enforce clear method contracts between interfaces and implementing classes, enhancing code reliability.

- ***Polymorphism***: Interfaces enable objects of different classes to be treated as objects of a common interface type, promoting code reusability.

- ***Decoupling and Design Patterns***: Interfaces support loose coupling, leading to modular and maintainable code. They are essential for design patterns and design flexibility.

Example Implementation of Interface on playable devices

Example of Interface

```
public interface Playable {
    void play();
}
```

define an interface called ***Playable***, which declares a single method ***play()***.

Example of Class implementing interface

```
public class VideoGame implements Playable {
    private String title;

    public VideoGame(String title) {
        this.title = title;
    }

    @Override
    public void play() {
        System.out.println("Playing the video game: " + title);
    }
}
```

```
public class MusicPlayer implements Playable {
    private String song;

    public MusicPlayer(String song) {
        this.song = song;
    }

    @Override
    public void play() {
        System.out.println("Playing the song: " + song);
    }
}
```

Define two classes, ***VideoGame*** and ***MusicPlayer***, both of which implement the ***Playable*** interface. Each class provides its own implementation of the ***play()*** method.

Example Usage

```
public class App {
    public static void main(String[] args) {
        Playable game = new VideoGame("Super Mario");
        Playable music = new MusicPlayer("Bohemian Rhapsody");

        game.play();  // Output: Playing the video game: Super Mario
        music.play(); // Output: Playing the song: Bohemian Rhapsody
    }
}
```

In the ***App*** class, we create instances of ***VideoGame*** and ***MusicPlayer*** and call their ***play()*** methods. This demonstrates how different classes can implement the same interface to achieve polymorphism and provide their unique "play" functionality.

Exception Handling

Exception handling is a powerful mechanism in Java that provides a way to handle runtime errors, allowing the program to continue its execution or terminate gracefully. It uses a combination of blocks and keywords to catch and manage exceptions, ensuring that the program doesn't crash unexpectedly.

Key Concepts of Exception Handling

- ***Try-Catch Block***: The try block contains code that might throw an exception. The catch block captures and handles the exception if one occurs in the try block.
- ***Throwing Exceptions***: The throw keyword is used to explicitly throw an exception, signaling an exceptional condition in the program.
- ***Exception Propagation***: If a method doesn't handle an exception, it propagates up the call stack to the previous method, and so on, until it's caught or reaches the main method.
- ***Finally Block***: The finally block contains code that is always executed, regardless of whether an exception occurred or not. It's often used for cleanup activities.
- ***Checked vs. Unchecked Exceptions***: Java categorizes exceptions as checked (must be explicitly caught or thrown) and unchecked (runtime exceptions that don't need to be declared or caught).

Benefits of Exception Handling

- ***Graceful Termination***: Instead of crashing, programs can handle exceptions gracefully, providing meaningful error messages or taking alternative actions.
- ***Separation of Error Handling Code***: Exception handling separates the error handling code from the regular code, making the program cleaner and more readable.
- ***Program Reliability***: By handling potential errors, programs become more robust and reliable, ensuring they can handle unexpected situations.
- ***Resource Management***: The finally block ensures resources, like files or network connections, are closed or released, preventing resource leaks.
- ***Controlled Propagation***: Exceptions can be propagated up the call stack, allowing higher-level methods to handle them, leading to centralized error handling.

Example Usage and Implementation of each Exception Handling

try-catch blocks

- The ***try*** block contains the code that might throw an exception.
- The ***catch*** block contains the code that will be executed if an exception is thrown in the ***try*** block.

```
try {
    // Some code...
} catch (ArithmeticException e) {
    // Handle arithmetic exceptions
} catch (NullPointerException e) {
    // Handle null pointer exceptions
}
```

You can have multiple ***catch*** blocks to handle different types of exceptions.

```
try {
    // Some code...
} catch (ArithmeticException e) {
    // Handle arithmetic exceptions
} catch (NullPointerException e) {
    // Handle null pointer exceptions
}
```

The ***finally*** block contains code that will always be executed, regardless of whether an exception was thrown or not.

```
try {
    // Some code...
} catch (Exception e) {
    // Handle exception
} finally {
    System.out.println("This will always be executed.");
}
```

Throwing Exception

You can throw an exception using the ***throw*** keyword. This is useful when you want to signal that an exceptional condition has occurred.

```
public void someMethod(int value) throws Exception {
    if (value < 0) {
        throw new Exception("Value cannot be negative");
    }
}
```

Custom Exceptions

You can create your own exception classes by extending the ***Exception*** class.

```
public class CustomException extends Exception {
    public CustomException(String message) {
        super(message);
    }
}
```

Then the ***CustomException*** can be throwed

```
throw new CustomException("This is a custom exception");
```

Strategy Pattern

The Strategy Pattern is a behavioral design pattern that defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows the algorithm to vary independently from the clients that use it.

Intents

- Define a set of algorithms, encapsulate each one, and make them interchangeable.
- Allow an object to alter its behavior when its internal state changes.

Key Concepts of Strategy Pattern

- ***Strategy (Interface/Abstract Class)***: This defines an interface common to all supported algorithms.
- ***ConcreteStrategy (Classes)***: These implement the algorithm defined by the Strategy.
- ***Context (Class)***: This class maintains a reference to a Strategy object and can switch between different strategies.

Benefits of Strategy Pattern

- ***Flexibility***: The Strategy Pattern allows you to define a family of algorithms and make them interchangeable, providing flexibility in choosing the algorithm at runtime.
- ***Decoupling***: It decouples the algorithm from the context that uses it, promoting loose coupling.
- ***Open/Closed Principle***: The system can be extended with new strategies without modifying the existing code.

Example Usage and Implementation of Strategy Pattern

Let's consider an example of a simple e-commerce system where different types of discounts (strategies) can be applied to an order.

Strategy

```
public interface DiscountStrategy {
    double applyDiscount(double price);
}
```

Concrete Strategy

```
public class NoDiscount implements DiscountStrategy {
    @Override
    public double applyDiscount(double price) {
        return price;
    }
}

public class SeasonalDiscount implements DiscountStrategy {
    @Override
    public double applyDiscount(double price) {
        return price * 0.9; // 10% discount
    }
}

public class ClearanceDiscount implements DiscountStrategy {
    @Override
    public double applyDiscount(double price) {
        return price * 0.5; // 50% discount
    }
}
```

Context

```
public class Order {
    private DiscountStrategy discountStrategy;
    private double price;

    public Order(double price, DiscountStrategy discountStrategy) {
        this.price = price;
        this.discountStrategy = discountStrategy;
    }

    public double getFinalPrice() {
        return discountStrategy.applyDiscount(price);
    }

    public void setDiscountStrategy(DiscountStrategy discountStrategy) {
        this.discountStrategy = discountStrategy;
    }
}
```

Usage

```
public class Main {
    public static void main(String[] args) {
        Order order = new Order(100, new NoDiscount());
        System.out.println("with No Discount: " + order.getFinalPrice());

        order.setDiscountStrategy(new SeasonalDiscount());
        System.out.println("with Seasonal Discount: " + order.getFinalPrice());

        order.setDiscountStrategy(new ClearanceDiscount());
        System.out.println("with Clearance Discount: " + order.getFinalPrice());
    }
}
```

Data Types

Primitive Data Types

- byte:
  - 8-bit signed integer.
  - Range: -128 to 127.
- short:
  - 16-bit signed integer.
  - Range: -32,768 to 32,767.
- int:
  - 32-bit signed integer.
  - Range: -2^31 to 2^31 -1.
- long:
  - 64-bit signed integer.
  - Range: -2^63 to 2^63 -1.
- float:
  - 32-bit floating-point number.
- double:
  - 64-bit floating-point number.
- char:
  - 16-bit Unicode character.
  - Range: 0 to 65,535.
- boolean:
  - Represents only two possible values: **true** or **false**

Reference Data Types

- Objects: Any object you create becomes a reference data type. For example, if you create a class Car, then a variable of type Car would be a reference data type.
- Arrays: Arrays are also considered reference data types in Java.

String

While String is technically a reference data type (since it's an object), it's worth mentioning separately due to its frequent use and special behavior in Java.

- Strings are immutable in Java, meaning their values cannot be changed after they're created.
- Java provides a special syntax for creating strings using double quotes.

```
String name = "John Doe";
```

Array

What is an Array

An array in Java is a homogeneous data structure that can store multiple values of the same type in contiguous memory locations. It can be thought of as a collection of variables that are accessed with an index.

Declaring Arrays

You can declare an array by specifying its type followed by square brackets.

```
int[] myArray;
```

Initializing Arrays

Arrays can be initialized in various ways:

- At the time of declaration
  ```
  int[] myArray = {1, 2, 3, 4, 5};
  ```
- Using the ***new*** keyword
  ```
  int[] myArray = new int[5];  // Allocates memory for 5 integers
  ```

Accessing Array Elements

You can access an element of an array using its index. Remember, array indices start from 0

```
int firstElement = myArray[0];
```

Modifying Array Elements

You can modify an element of an array by using its index

```
myArray[2] = 10;  // Sets the third element to 10
```

Array Length

You can find the length (number of elements) of an array using the length property:

```
int arrayLength = myArray.length;
```

Looping Through Arrays

You can loop through arrays using standard loops or the enhanced for loop

- Standard ***for*** loop

```
for (int i = 0; i < myArray.length; i++) {
    System.out.println(myArray[i]);
}
```

- Enhanced ***for*** loop

```
for (int num : myArray) {
    System.out.println(num);
}
```

Add Elements to an Array

```
public class AddToArrayExample {

    public static void main(String[] args) {
        int[] originalArray = {1, 2, 3, 4, 5};
        int newElement = 6;

        int[] newArray = addElement(originalArray, newElement);

        for (int num : newArray) {
            System.out.println(num);
        }
    }

    public static int[] addElement(int[] original, int element) {
        int length = original.length;

        // Create a new array with size increased by 1
        int[] newArray = new int[length + 1];

        // Copy elements from the original array to the new array
        for (int i = 0; i < length; i++) {
            newArray[i] = original[i];
        }

        // Add the new element to the last position of the new array
        newArray[length] = element;

        return newArray;
    }
}
```

ArrayList

the ***ArrayList*** class is a part of the Java Collections Framework and provides a dynamic array-like data structure. Unlike arrays, ***ArrayList*** can dynamically grow and shrink in size.

Importing the *ArrayList* class

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

Declaring an *ArrayList*

You can declare an *ArrayList* by specifying its type within angle brackets (< >)

```
ArrayList<String> names = new ArrayList<>();
```

Adding Elements

You can add elements to an ArrayList using the add method

```
names.add("Alice");
names.add("Bob");
names.add("Charlie");
```

Accessing Elements

You can access an element of an ***ArrayList*** using the ***get*** method

```
String firstPerson = names.get(0); // Gets the first element(index starts from 0)
```

Modifying Elements

You can modify an element of an ***ArrayList*** using the ***set*** method

```
names.set(1, "Robert");  // Changes the second element to "Robert"
```

Removing Elements

You can remove an element from an ***ArrayList*** by its index or by its value

```
names.remove(0);          // Removes the first element
names.remove("Charlie");  // Removes the element "Charlie"
```

Getting the Size

You can find the number of elements in an ***ArrayList*** using the ***size*** method

```
int size = names.size();
```

Looping Through an ArrayList

You can loop through an ArrayList using a standard loop or an enhanced for loop

- Standard ***for*** loop

```
for (int i = 0; i < names.size(); i++) {
    System.out.println(names.get(i));
}
```

- Enhanced ***for*** loop

```
for (String name : names) {
    System.out.println(name);
}
```

Checking if an ArrayList Contains an Element

You can check if an ***ArrayList*** contains a specific element using the ***contains*** method

```
boolean hasAlice = names.contains("Alice");
```

Useful Examples

Multi-Constructor Abstraction

```
// Abstract class Shape to define the common properties and methods for different shapes
abstract class Shape {
    // Color of the shape
    protected String color;

    // Constructor to initialize the color
    public Shape(String color) {
        this.color = color;
    }

    // Abstract method to get the area of the shape
    public abstract double getArea();

    // Abstract method to resize the shape
    public abstract void resize(double factor);

    // Getter method for color
    public String getColor() {
        return color;
    }
}

// Class Square extending Shape
class Square extends Shape {
    // Side length of the square
    private double sideLength;

    // Constructor to initialize the side length and color
    public Square(double sideLength, String color) {
        super(color);
        this.sideLength = sideLength;
    }

    // Overridden method to get the area of the square
    @Override
    public double getArea() {
        return this.sideLength * this.sideLength;
    }

    // Method to resize the square
    public void resize(double factor) {
        this.sideLength *= factor;
    }
}
```

2190102 ADV COMP PROG Cheat Sheet - F

By @WasinUddy (✉ ws@prometheuzdy.cloud)

Observer Pattern

The Observer Pattern is a widely used design pattern in software development, particularly useful when building systems where the state of one object affects the state of others.

Key Concepts of Observer Pattern

- ***Subject (Observable)*** : This is the entity that holds the state. When its state changes, it needs to notify its observers.

- ***Observers*** : These are the entities that need to be informed about the state changes in the subject. They implement a common interface that allows the subject to notify them of any changes.

Benefits of Observer Pattern

- ***Loose Coupling*** : The subject does not need to know details about the observers, just that they implement a specific interface.

- ***Dynamic Subscription*** : Objects can dynamically subscribe or unsubscribe from receiving updates.

Drawbacks of Observer Pattern

- ***Memory Leaks*** : Improper unsubscribe can lead to memory leaks.

Example Implementation of Observer Pattern

Define the Observer and Subject Interface

```
public interface Observer {
    void update(String message);
}
```

```
public interface Subject {
    void registerObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers();
}
```

Implement the Observer and Subject

```
public class NewsReader implements Observer {
    private String name;

    public NewsReader(String name) {
        this.name = name;
    }

    @Override
    public void update(String news) {
        System.out.println(name + " received news: " + news);
    }
}
```

```
import java.util.ArrayList;
import java.util.List;

public class NewsAgency implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String news;

    public void setNews(String news) {
        this.news = news;
        notifyObservers();
    }

    @Override
    public void registerObserver(Observer o) {
        observers.add(o);
    }

    @Override
    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(news);
        }
    }
}
```

Using the Pattern

```
public class Main {
    public static void main(String[] args) {
        NewsAgency agency = new NewsAgency();
        NewsReader reader1 = new NewsReader("Reader 1");
        NewsReader reader2 = new NewsReader("Reader 2");

        agency.registerObserver(reader1);
        agency.registerObserver(reader2);

        agency.setNews("New Java version released!");

        agency.removeObserver(reader2);

        agency.setNews("Another important news!");
    }
}
```

Decorator Pattern

The Decorator Pattern is used to extend or alter the functionality of objects at runtime by wrapping them in an object of a decorator class. This provides a flexible alternative to using inheritance to modify behavior.

Key Concepts of Decorator Pattern

- ***Component***: This is an interface or abstract class defining the methods that will be implemented. In our case, it's the object to which new functionality will be added.
- ***Concrete Component***: A class that implements the Component interface.
- ***Decorator***: This is an abstract class that implements the Component interface and has a reference to a Component object. It can also add additional functionality.
- ***Concrete Decorator***: A class that extends the Decorator class and adds extra behaviors.

Benefits of Decorator Pattern

- ***More Flexibility than Inheritance*** : It allows extending the behavior of objects without modifying the original class.
- ***Avoids Explosion*** : Instead of having many a hierarchy of classes to combine behaviors, you can mix and match decorators as needed.

Drawbacks of Decorator Pattern

- ***Complexity*** : Using Decorator Pattern can lead to a hard to read code.
- ***Instantiation Management*** : Order of decorating pattern can lead to different results.

Example Implementation of Decorator Pattern

Define the Component

```
public interface Coffee {
    String getDescription();
    double getCost();
}
```

Define the Concrete Component

```
public class SimpleCoffee implements Coffee {
    @Override
    public String getDescription() {
        return "Simple Coffee";
    }

    @Override
    public double getCost() {
        return 2.0;
    }
}
```

Define the Decorator

```
public abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee coffee) {
        this.decoratedCoffee = coffee;
    }

    public String getDescription() {
        return decoratedCoffee.getDescription();
    }

    public double getCost() {
        return decoratedCoffee.getCost();
    }
}
```

Define the Concrete Decorator

```
public class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", with milk";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.5;
    }
}

public class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", with sugar";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.2;
    }
}
```

Using the Pattern

```
public class Main {
    public static void main(String[] args) {
        Coffee simpleCoffee = new SimpleCoffee();
        System.out.println(simpleCoffee.getDescription() + " Cost: $" +
simpleCoffee.getCost());

        Coffee milkCoffee = new MilkDecorator(simpleCoffee);
        System.out.println(milkCoffee.getDescription() + " Cost: $" +
milkCoffee.getCost());

        Coffee milkSugarCoffee = new SugarDecorator(milkCoffee);
        System.out.println(milkSugarCoffee.getDescription() + " Cost: $" +
milkSugarCoffee.getCost());
    }
}
```

Factory Pattern

The Factory Pattern is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. This pattern is particularly useful when a system should be independent of how its objects are created, composed, and represented.

Key Concepts of Factory Pattern

- ***Product*** : The interface or abstract class defining the type of objects the factory method will create.
- ***Concrete Products*** : The actual implementations of the Product Interface.
- ***Factory*** : An Interface or abstract class that declares the factory that will return the Product.
- ***Concrete Factory*** : A subclass of Factory that overrides the factory method to return an instance of a Concrete Product

Benefits of Factory Pattern

- **Loose Coupling**: The Factory Pattern promotes loose coupling by reducing the dependency of the application on concrete classes.
- **Single Responsibility Principle**: The factory class handles the creation of objects, which separates the responsibility of object creation from the object's usage.
- **Open/Closed Principle**: You can introduce new types of products without disturbing the existing client code.

Drawbacks of Factory Pattern

- **Complexity**: The code can become more complicated since it introduces several new classes and interfaces.
- **Development Overhead**: More classes and objects to manage can increase the complexity of the codebase.

Example Implementation of Factory Pattern

Define the Product

```
public interface Vehicle {
    void design();
    void manufacture();
}
```

Create Concrete Products

```
public class Car implements Vehicle {
    @Override
    public void design() {
        System.out.println("Designing a Car");
    }

    @Override
    public void manufacture() {
        System.out.println("Manufacturing a Car");
    }
}

public class Bike implements Vehicle {
    @Override
    public void design() {
        System.out.println("Designing a Bike");
    }

    @Override
    public void manufacture() {
        System.out.println("Manufacturing a Bike");
    }
}
```

Create the Factory

```
public abstract class VehicleFactory {
    public abstract Vehicle createVehicle(String type);

    // Other helper methods can be added here
}
```

Implement Concrete Factory

```
public class ConcreteVehicleFactory extends VehicleFactory {
    @Override
    public Vehicle createVehicle(String type) {
        if (type.equalsIgnoreCase("car")) {
            return new Car();
        } else if (type.equalsIgnoreCase("bike")) {
            return new Bike();
        }
        return null;
    }
}
```

Using the Factory

```
public class Main {
    public static void main(String[] args) {
        VehicleFactory factory = new ConcreteVehicleFactory();

        Vehicle car = factory.createVehicle("car");
        car.design();
        car.manufacture();

        Vehicle bike = factory.createVehicle("bike");
        bike.design();
        bike.manufacture();
    }
}
```

Singleton Pattern

The Singleton Pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance. It's commonly used when exactly one object is needed to coordinate actions across the system, such as in database connections or logging.

Key Concepts of Singleton Pattern

The Singleton Pattern restricts the instantiation of a class to a single object and provides a way to access that object from anywhere in the application. This is achieved by:

- Making the constructor private to prevent the use of the new operator.
- Creating a static method that acts as a constructor. This method calls the constructor to create an object if one doesn't exist and returns the object if it already exists.

Implementation of Singleton Pattern

Define Singleton Class

```
public class Government {
    private static Government instance = new Government("Democratic");

    private String type;
    private int numberOfPoliciesEnacted;

    // Private constructor with government type
    private Government(String type) {
        this.type = type;
        this.numberOfPoliciesEnacted = 0;
        // Initialize other properties and departments
    }

    // Public method to access the single instance
    public static Government getInstance() {
        return instance;
    }

    public void createPolicy(String policyName) {
        numberOfPoliciesEnacted++;
        System.out.println("Policy enacted: " + policyName + ". Total policies: " +
numberOfPoliciesEnacted);
    }

    public String getType() {
        return type;
    }

    // Other government functions...
}
```

Using the Singleton

```
public class Main {
    public static void main(String[] args) {
        // Get the single instance of Government
        Government government = Government.getInstance();

        // Display government type
        System.out.println("Government type: " + government.getType());

        // Use the government instance to create policies
        government.createPolicy("Healthcare Reform");
        government.createPolicy("Education System Improvement");
    }
}
```

Define 2nd Singleton Class

```
public class Government {
    private static Government instance = new Government();

    private String type;

    private Government() {
        // Default type
        this.type = "Democratic";
    }

    public static Government getInstance() {
        return instance;
    }

    public void setType(String type) {
        this.type = type;
    }

    public String getType() {
        return type;
    }

    public void createPolicy(String policyName) {
        System.out.println(type + " Government Policy enacted: " + policyName);
    }
}
```

Using 2nd Singleton class

```
public class Main {
    public static void main(String[] args) {
        // Get the single instance of Government
        Government government = Government.getInstance();

        // Initially, the government is Democratic
        government.createPolicy("Free Market Policy");

        // Change the government type to Communist
        government.setType("Communist");
        government.createPolicy("Five-Year Plan");

        // The government type can be changed again if needed
        // government.setType("Democratic");
    }
}
```

Introduction to JavaScript

JavaScript is a versatile scripting language primarily used for creating interactive features on web pages. It is an essential part of web development alongside HTML and CSS. JavaScript can be used for both client-side (in the browser) and server-side (on the server, e.g., Node.js) programming.

Basic Syntax

Statements

JavaScript instructions are called statements and are separated by semicolons (;).

```
let x = 5;
let y = 6;
let sum = x + y;
console.log(sum); // Outputs: 11
```

Comments

Use **//** for single-line comments and **/* */** for multi-line comments.

```
// This is a single-line comment
/*
This is a multi-line comment
*/
```

Variables and Data Types

Variables

Declared with **var**, **let** (block scope) or **const** (constant)

```
let message = "Hello, world!";
const pi = 3.14;
```

Data Types

JavaScript is a dynamically typed language. The main data types are:

- **Number**: Both integers and floats.
- **String**: Textual data.
- **Boolean**: true or false.
- **Undefined**: A variable that has not been assigned a value.
- **Null**: Denotes a null value.
- **Object**: For more complex data structures.
- **Array**: A list-like object.

```
let age = 30;             // Number
let name = "Alice";       // String
let isAdult = true;       // Boolean
let undef;                // Undefined
let empty = null;         // Null
let user = {              // Object
  firstName: "Bob",
  lastName: "Smith"
};
let colors = ["Red", "Green", "Blue"]; // Array
```

Control Structures

Conditional Statements

```
if (age >= 18) {
    console.log("Adult");
} else {
    console.log("Minor");
}
```

Switch Statement

```
switch (color) {
    case "Red":
        console.log("Color is Red");
        break;
    case "Blue":
        console.log("Color is Blue");
        break;
    default:
        console.log("Different Color");
}
```

Loops

For Loop

```
for (let i = 0; i < 5; i++) {
    console.log(i);
}
```

While Loop

```
let i = 0;
while (i < 5) {
    console.log(i);
    i++;
}
```

Do-While Loop

```
let i = 0;
do {
    console.log(i);
    i++;
} while (i < 5);
```

Functions

Function Declaration

```
function greet(name) {
    return "Hello, " + name + "!";
}
console.log(greet("Alice")); // Outputs: Hello, Alice!
```

Arrow Functions

```
const add = (a, b) => a + b;
console.log(add(5, 3)); // Outputs: 8
```

Arrays and Objects

Arrays

```
let fruits = ["Apple", "Banana", "Cherry"];
console.log(fruits[0]); // Outputs: Apple
```

Objects

```
let person = {
    firstName: "John",
    lastName: "Doe",
    age: 30
};
console.log(person.firstName); // Outputs: John
```

Asynchronous Programming

Asynchronous programming in JavaScript is a critical concept, especially in modern web development where tasks like fetching data from a server, reading files, or executing time-consuming logic are common. It allows JavaScript to perform these tasks without blocking the main thread, ensuring a smooth user experience. Let's dive into the key concepts and techniques.

Key Concepts of Asynchronous Programming

- **Synchronous vs Asynchronous**: In synchronous operations, tasks are performed one after another. In contrast, asynchronous operations allow JavaScript to start a task and move on to the next one without waiting for the previous task to finish.

- **Event Loop**: JavaScript has a single-threaded runtime model based on an event loop. The event loop continuously checks the queue of pending tasks and executes them when possible, without blocking the main thread.

- **Callbacks**: The most basic method for asynchronous programming. A callback is a function passed into another function as an argument and is executed after a task is completed.

- **Promises**: A more advanced way of handling asynchronous operations. A Promise represents a value that may not be available yet but will be resolved at some point in the future.

- **Async/Await**: Introduced in ES8, async/await is syntactic sugar built on top of Promises, making asynchronous code easier to write and read.

Callbacks

Callbacks are functions passed as arguments to another function and are executed after a task is completed. They're the foundation of asynchronous programming in JavaScript

Basic Concepts

You provide a function (the callback) to another function, telling it to execute the callback after completing a task

Handling Asynchronous Operations

For operations like reading files, network requests, or timers, callbacks are used to continue the flow of the program once the operation is completed.

Example Implementation of Callbacks

```
function download(url, callback) {
    setTimeout(() => { // Simulate a time-consuming task
        console.log(`Downloading ${url} ...`);
        callback(url.split('/').pop());
    }, 2000);
}

download('http://example.com/file', function(fileName) {
    console.log(`Processing the downloaded file: ${fileName}`);
});
```

Promise

Promises are an evolution of callbacks, providing a more robust way to handle asynchronous operations.

Basic Concepts

- **States**: A Promise can be in one of three states: pending, resolved (fulfilled), or rejected.
- **Chaining**: Promises can be chained, allowing for sequential execution of asynchronous operations.
- **Error Handling**: **.catch()** method provides a cleaner way to handle errors compared to callbacks.

Example Implementation of Promise

```
function fetchData(url) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if(url) {
                resolve(`Data from ${url}`);
            } else {
                reject('No URL provided');
            }
        }, 1000);
    });
}

fetchData('http://example.com/data')
    .then(data => console.log(data))
    .catch(error => console.error(error));
```

Async/Await

Async/await is syntactic sugar built on top of promises, making asynchronous code look synchronous and more readable.

Basic Concepts

- **Async Function**: Declared with the async keyword. It always returns a promise.
- **Await Keyword**: Used inside an async function to wait for a promise to resolve.
- **Error Handling**: Async functions can use traditional try-catch blocks for error handling.

```
async function fetchData(url) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if(url) {
                resolve(`Data from ${url}`);
            } else {
                reject('No URL provided');
            }
        }, 1000);
    });
}

async function loadData() {
    try {
        const data = await fetchData('http://example.com/data');
        console.log(data);
    } catch (error) {
        console.error(error);
    }
}

loadData();
```

API Calls

API calls are a fundamental part of modern web development in JavaScript, allowing your application to communicate with external services and servers. Two popular ways to make these calls in JavaScript are using the **fetch()** API and the Axios library. Let's delve into each of these.

How to Use:

- **Basic GET Request**: Fetch data from a URL.
- **Handling Response**: The response of a fetch call is a stream object, which can be converted into the desired format, typically JSON.
- **Error Handling**: Use .catch() to handle network errors. However, fetch won't reject an HTTP error status even if the response is an HTTP 404 or 500.

```
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) {
        throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.error('There was a problem with the fetch operation:', error));
```

Processing the Response

```
.then(response => {
    if (!response.ok) {
        throw new Error('Network response was not ok');
    }
    return response.json();
})
```

- **.then(response => {...})**: This is a Promise handler. When the fetch request is completed, it processes the response.
- **response**: This object contains the response of the fetch request.
- **if (!response.ok)**: The ok property of the response is a boolean indicating whether the response was successful (status in the range 200–299) or not.
- **throw new Error('Network response was not ok')**: If the response was not successful (e.g., 404 or 500 HTTP status), an error is thrown. This will be caught in the .catch() block.
- **return response.json()**: If the response is successful, this line reads the body of the response and returns it as a Promise that resolves with the result of parsing the body text as JSON. This is because the data returned from the server is often in JSON format.

Handling the Data

```
.then(data => console.log(data))
```

- This **.then()** is chained to handle the resolution of the **response.json()** promise.
- **data**: This represents the parsed JSON data received from the previous **.then()**.
- **console.log(data)**: It logs the parsed data to the console. In a real application, you might do something more complex with this data, like updating the UI.

Catching Errors

```
.catch(error => console.error('There was a problem with the fetch operation:', error));
```

- **.catch(error => {...}):** This block catches any errors that occur during the fetch operation or processing in the **.then()** blocks.
- **error**: This represents the error that was caught.
- **console.error('There was a problem with the fetch operation:', error)**: This logs the error to the console. In a production environment, you might handle it by showing a message to the user or sending the error to a logging service.

DOM Manipulation

DOM manipulation in JavaScript is a fundamental concept for web development, allowing you to dynamically change the content and appearance of your web pages. The DOM (Document Object Model) is a programming interface for HTML and XML documents. It represents the page so that programs can change the document structure, style, and content. Here's a guide to some of the key aspects of DOM manipulation

Accessing Elements

JavaScript provides several methods to access and select elements from the DOM:

- **getElementById**: Selects an element by its ID.
- **getElementsByClassName**: Selects all elements that have a given class name.
- **getElementsByTagName**: Selects all elements with a specific tag name.
- **querySelector**: Uses CSS selectors to select the first matching element.
- **querySelectorAll**: Uses CSS selectors to select all matching elements.

```
const element = document.getElementById('myElement');
const classElements = document.getElementsByClassName('myClass');
const tagElements = document.getElementsByTagName('p');
const queryElement = document.querySelector('.myClass');
const queryAllElements = document.querySelectorAll('div.myClass');
```

Changing Content

- **textContent**: Sets or returns the textual content of an element and its descendants.
- **innerHTML**: Sets or gets the HTML or XML markup contained within the element.

```
element.textContent = 'New text content';
element.innerHTML = '<span>New HTML content</span>';
```

Manipulate CSS

You can manipulate the style of an element by accessing the style property

```
element.style.color = 'blue';
element.style.backgroundColor = 'yellow';
```

Adding and Removing Elements

- **createElement**: Creates a new element.
- **appendChild**: Adds a new child element to an element.
- **removeChild**: Removes a child element from an element.

```
const newElement = document.createElement('div');
newElement.textContent = 'Hello, World!';
document.body.appendChild(newElement);

const parentElement = document.getElementById('parent');
parentElement.removeChild(newElement);
```

Event Handling

Add event listeners to elements to handle user interactions like clicks, form submissions, key presses, etc.

```
element.addEventListener('click', function() {
    console.log('Element clicked!');
});
```

Attributes

Set or get attributes like **src**, **href**, **id**, etc., of an element.

```
const image = document.querySelector('img');
image.setAttribute('src', 'image.jpg');
let srcValue = image.getAttribute('src');
```

HTML (HyperText Markup Language)

HTML is the standard markup language for documents designed to be displayed in a web browser. It can be assisted by technologies like Cascading Style Sheets (CSS) and scripting languages like JavaScript.

Basic Structure of an HTML Document:

```
<!DOCTYPE html>
<html>
<head>
    <title>Page Title</title>
</head>
<body>

    <h1>My First Heading</h1>
    <p>My first paragraph.</p>

    <script>
        // JavaScript code can be placed here
    </script>

</body>
</html>
```

- **<!DOCTYPE html>**: Declares the document type and HTML version.
- **<html>**: The root element of an HTML page.
- **<head>**: Contains meta-information about the HTML document, like its title.
- **<title>**: Specifies a title for the document.
- **<body>**: Contains the visible page content.
- **<h1>, <p>**: HTML elements like headings, paragraphs, etc.
- **<script>**: Where JavaScript code is placed.