My CheatSheet

1. Explain the features of Java and how it achieves platform independence.
Java is an **object-oriented**, **platform-independent** programming language. Key features include simplicity, security, robustness, multithreading, and high performance. Platform independence is achieved primarily through the Java Virtual Machine (JVM). Java code is compiled into **bytecode** (.class files), which is a machine-independent format. This bytecode can then be executed on any system that has a compatible JVM installed, regardless of the underlying hardware or operating system. The "**Write Once, Run Anywhere**" (WORA) principle is a direct result of this architecture, allowing developers to deploy Java applications across diverse environments without modification.

2. Describe JVM, JRE, and JDK and explain their roles in Java program execution.
The **JVM (Java Virtual Machine)** is an abstract machine that provides a runtime environment for executing Java bytecode. It performs **garbage collection** and acts as an interpreter for bytecode. The **JRE (Java Runtime Environment)** is a software package that includes the JVM, core classes, and supporting files necessary to run Java applications. It does not contain development tools. The **JDK (Java Development Kit)** is a superset of JRE, providing all the tools for developing, debugging, and executing Java programs. It includes the JRE, along with essential development tools like the Java compiler (javac), debugger, and archiver. In essence, JDK is for development, JRE is for running, and JVM is for executing bytecode.

3. Explain Object-Oriented Programming in Java and discuss its four main principles.
Object-Oriented Programming (OOP) in Java is a paradigm based on the concept of "objects," which can contain data and code. Its four main principles are: 1. **Encapsulation**: Bundling data (attributes) and methods (behaviors) that operate on the data within a single unit (class), hiding internal implementation details. 2. **Inheritance**: A mechanism where a new class (**subclass**) derives properties and behaviors from an existing class (**superclass**), promoting code reusability. 3. **Polymorphism**: The ability of an object to take on many forms, allowing a single interface to represent different underlying forms (e.g., method **overloading** and **overriding**). 4. **Abstraction**: Hiding complex implementation details and showing only essential features to the user, achieved through abstract classes and interfaces.

4. What is inheritance in Java? Explain its types and how it is implemented.
Inheritance in Java is an OOP concept where a class (**child/subclass**) can acquire the fields and methods of another class (**parent/superclass**). It promotes **code reusability** and establishes an **is-a relationship**. Java primarily supports **single inheritance**, where a class inherits from only one direct superclass, implemented using the `extends` keyword. **Multilevel inheritance** (A extends B, B extends C) is also supported. **Hierarchical inheritance** (multiple subclasses extending a single superclass) is allowed. However, Java does not support **multiple inheritance** (a class inheriting from multiple superclasses) directly to avoid the 'diamond problem'; this is partially achieved through interfaces. Implementation involves creating a subclass that `extends` a superclass, allowing the subclass to access public and protected members of the superclass.

5. Explain polymorphism in Java and differentiate between compile-time and runtime polymorphism.
Polymorphism in Java, meaning "many forms," allows objects to exhibit different behaviors in different contexts. It enables a single interface to represent multiple methods. There are two main types: 1. **Compile-time Polymorphism** (or Static Polymorphism): Achieved through **method overloading**, where multiple methods in the same class have the same name but different parameters (number, type, or order). The appropriate method call is resolved during compilation. Example: `add(int a, int b)` and `add(double a, double b)`. 2. **Runtime Polymorphism** (or Dynamic Polymorphism): Achieved through **method overriding**, where a subclass provides a specific implementation for a method already defined in its superclass. The actual method invoked is determined at runtime based on the object type. Example: a `draw()` method in a `Shape` class overridden by `Circle` and `Square` classes.

6. What is encapsulation? How is it achieved in Java?
Encapsulation is an **object-oriented programming concept** that binds data (attributes) and methods (functions) that operate on the data into a single unit, a class. It restricts direct access to some of an object's components, meaning internal representation is hidden from the outside. In Java, encapsulation is achieved by declaring instance variables as `private` and providing public **getter and setter methods** to access and modify these variables. This approach ensures data integrity and helps in controlling how data is accessed and updated, promoting better code organization and maintainability. It's a core principle for building robust applications.

7. Explain abstraction in Java and compare abstract classes with interfaces.
Abstraction is the process of **hiding implementation details** and showing only functionality to the user. In Java, it's achieved using abstract classes and interfaces. An **abstract class** can have abstract (no body) and concrete methods, can have constructors, static and final methods, and can have instance variables. A class can extend only one abstract class. An **interface** in Java is a blueprint of a class that can only contain abstract methods (pre-Java 8) and constants. From Java 8, it can also have default and static methods. A class can implement multiple interfaces, supporting multiple inheritance of type. Both enforce a contract but abstract classes provide a partial implementation while interfaces define a contract without implementation details.

8. Describe exception handling in Java and explain try, catch, finally, throw, and throws.
Exception handling in Java manages **runtime errors** gracefully, preventing program crashes. The `try` block encloses code that might throw an exception. The `catch` block follows, specifying the type of exception it handles and the recovery code. The `finally` block always executes, regardless of whether an exception occurred or was caught, often used for resource cleanup. The `throw` keyword is used to **explicitly throw an exception** from a method or any block of code. The `throws` keyword is used in a method signature to declare which exceptions might be thrown by that method, obliging the caller to handle them. This mechanism ensures program stability and robustness.

9. What is multithreading in Java? Explain the thread lifecycle.
Multithreading in Java allows **concurrent execution of multiple parts** of a program (threads) to maximize CPU utilization. The **thread lifecycle** includes: **New** (thread created, not yet started), **Runnable** (thread ready to run, waiting for CPU), **Running** (thread is executing), **Blocked/Waiting** (thread temporarily inactive, waiting for a resource or event), and **Terminated** (thread finished execution). Threads can transition between these states based on various factors like acquiring locks, waiting for I/O, or completing their execution. This parallel processing boosts application performance and responsiveness, especially in resource-intensive tasks.

10. Explain the Java Collections Framework and describe List, Set, and Map.
The Java Collections Framework (JCF) is a set of interfaces and classes for **representing and manipulating collections of objects**. It provides a unified architecture to store and manipulate groups of data. **`List`** is an ordered collection (sequence) that allows duplicate elements; elements are accessed by their integer index (e.g., `ArrayList`, `LinkedList`). **`Set`** is a collection that cannot contain duplicate elements; it models the mathematical set abstraction (e.g., `HashSet`, `TreeSet`). **`Map`** is an object that maps keys to values; it cannot contain duplicate keys, and each key can map to at most one value (e.g., `HashMap`, `TreeMap`). JCF offers efficient algorithms and data structures.

11. Differentiate between Array and ArrayList in Java.
An **Array** in Java is a fixed-size data structure that stores elements of the same data type. Its size is determined at creation and cannot be changed. Accessing elements is fast using an index. It can store both primitive data types and objects. An **ArrayList** is a dynamic-size collection from the `java.util` package, implementing the `List` interface. It can grow or shrink in size as needed. It stores only objects, not primitive data types directly (autoboxing handles this). ArrayList operations like add and remove can be slower due to potential resizing and shifting of elements, but it provides more flexibility.

12. Explain String handling in Java and differentiate between String, StringBuilder, and StringBuffer.
String handling in Java involves manipulating sequences of characters. The `String` class creates **immutable** objects, meaning once a String is created, its value cannot be changed. Any operation appearing to modify a String actually creates a new String object. `StringBuffer` is a **mutable** sequence of characters, designed for thread-safe operations. All its methods are synchronized, making it suitable for multi-threaded environments but potentially slower. `StringBuilder` is also a **mutable** sequence of characters, similar to StringBuffer, but it is **not thread-safe**. This makes StringBuilder generally faster than StringBuffer for single-threaded applications as it doesn't incur the overhead of synchronization.

13. What is file handling in Java? Explain how to read and write files.
File handling in Java refers to the operations performed on files, such as creating, reading, writing, and deleting them. Java provides classes like `File`, `FileReader`, `FileWriter`, `BufferedReader`, and `BufferedWriter` for this purpose. To **read** a file, you typically use `FileReader` nested within a `BufferedReader`. `BufferedReader reader = new BufferedReader(new FileReader("file.txt"));` and then `reader.readLine()` or `reader.read()`. To **write** to a file, you use `FileWriter` nested within a `BufferedWriter`. `BufferedWriter writer = new BufferedWriter(new FileWriter("file.txt"));` followed by `writer.write("content");`. Always remember to **close** readers/writers in a `finally` block or use try-with-resources to prevent resource leaks.

14. What is a constructor in Java? Explain its types and uses.
A **constructor** in Java is a special method used to initialize objects. It has the same name as its class and no return type, not even `void`. Constructors are invoked automatically when an object is created using the `new` keyword. There are two main types: the **default constructor** (provided automatically by Java if no explicit constructor is defined, providing no-argument initialization) and **parameterized constructors** (defined explicitly by the programmer, taking arguments to initialize object properties with specific values). Constructors are crucial for ensuring objects are in a valid state upon creation and for performing any necessary setup operations.

15. What are access modifiers in Java? Explain public, private, protected, and default.
Access modifiers in Java control the **visibility** and **accessibility** of classes, fields, methods, and constructors within a program, enforcing encapsulation. There are four types: **public** elements are accessible from anywhere. **private** elements are only accessible within the class itself. **protected** elements are accessible within the class, by subclasses (inheritance), and within the same package. The **default** (or package-private) modifier means elements are accessible only within the same package; if no modifier is specified, this is the default. They secure data and control interactions between different parts of a program.