ItsRunTym

JAVA CheatSheet

Topic	Subtopics
1. JVM, JRE, JDK	
2. Java Data Types	Primitive, Reference Data Types
3. Operators in Java	Arithmetic, Relational, Logical, Bitwise
4. Control Statements	If-Else, Switch, For loop, While loop
5. Arrays	Multidimensional Arrays
6. Exception Handling	Try-Catch, Throw
7. Java OOP Concepts	Encapsulation, Inheritance, Polymorphism, Abstraction
8. Additional OOP Topics	Constructor, this & super keyword
9. Static and Final keyword	
10. String Handling in Java	String Class, Common Methods, StringBuilder and StringBuffer
11. Java Memory Management	Stack vs Heap Memory, Garbage Collection, Memory Leaks
12. Static Keyword	Static Variable, Static Method, Static Block
13. Inner Classes	Nested Class, Static Nested Class, Inner Class, Local Inner Class, Anonymous Inner Class
14. Java Collections Framework	Collection Interface, List Interface, Set Interface, Map Interface
15. Generics	
16. Java Multithreading	Creating Threads, Thread Methods
17. Synchronization in Java	
18. Java 8 Features	Lambdas and Functional Programming, Streams API, Optional Class
19. Design Patterns in Java	Singleton Pattern, Factory Pattern

Topic	Subtopics
20. Exception Handling (Expanded)	**Checked vs Unchecked Exceptions, Custom Exceptions**
21. Input/Output (I/O) in Java	**Streams, File Handling, BufferedReader and FileReader, PrintWriter**

Java Cheat Sheet

1. JVM, JRE, JDK

- **JVM (Java Virtual Machine)**:
    - Runs the Java bytecode generated by the compiler.
    - Converts the bytecode into machine-specific instructions.
    - Manages memory via Garbage Collection (GC) and ensures platform independence.
- **JRE (Java Runtime Environment)**:
    - Provides libraries and JVM to run Java applications.
    - Includes everything required to run Java programs but doesn't include development tools like a compiler.
- **JDK (Java Development Kit)**:
    - Contains JRE plus tools like a compiler (javac), debugger, and libraries necessary for Java development.
    - Without JDK, you can't develop Java applications but can run them with JRE.

Key Commands:

```
javac MyClass.java  # Compile source code
java MyClass        # Run the bytecode
```

---

2. Java Data Types

Primitive Data Types

- **byte**: 8-bit, stores whole numbers from -128 to 127.
- **short**: 16-bit, stores whole numbers from -32,768 to 32,767.
- **int**: 32-bit, stores whole numbers from -2^31 to 2^31-1.
- **long**: 64-bit, stores whole numbers from -2^63 to 2^63-1.

- **float**: 32-bit floating-point numbers.
- **double**: 64-bit floating-point numbers.
- **char**: 16-bit Unicode character (e.g., 'A', '9', '#').
- **boolean**: Stores true/false values.

Example:

int age = 25;

float price = 10.99f;

boolean isJavaFun = true;

char grade = 'A';

Reference Data Types

- **Classes**: User-defined blueprints (objects are instances).
- **Arrays**: Collection of similar type elements.
- **Interfaces**: Contracts specifying methods a class must implement.

Example:

String name = "Karan";

int[] scores = { 85, 90, 95 };

---

3. Operators in Java

Arithmetic Operators:

- +, -, *, /, %: For performing basic mathematical operations.

Example:

java

int sum = 10 + 5; // 15

int remainder = 10 % 3; // 1

Relational Operators:

- ==, !=, >, <, >=, <=: For comparing values.

Example:

java

boolean isEqual = (10 == 10); // true

Logical Operators:

- && (AND), || (OR), ! (NOT): Used to combine multiple conditions.

Example:

boolean result = (5 > 3) && (7 < 9); // true

Bitwise Operators:

- &, |, ^, ~, <<, >>, >>>: Perform operations at the bit level.

Example:

java

int x = 6 & 3; // Bitwise AND -> 2

---

4. Control Statements

If-Else:

- Used for conditional execution.

Example:

java

if (age > 18) {
   System.out.println("Adult");
} else {
   System.out.println("Minor");
}

Switch:

- Used when multiple conditions depend on a single variable.

Example:

java

switch (day) {
   case 1: System.out.println("Sunday"); break;
   case 2: System.out.println("Monday"); break;
   default: System.out.println("Invalid day");

}

Loops:

- **For loop**: Used when the number of iterations is known.

java

```
for (int i = 0; i < 5; i++) {
   System.out.println(i);
}
```

- **While loop**: Used when the condition is checked before each iteration.

java

```
int i = 0;
while (i < 5) {
   System.out.println(i);
   i++;
}
```

---

5. Arrays

- A container object that holds a fixed number of values of a single type.

Example:

java

```
int[] arr = {1, 2, 3, 4, 5};
System.out.println(arr[0]);  // Output: 1
```

- **Multidimensional Arrays**:

java

```
int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
System.out.println(matrix[1][2]);  // Output: 6
```

---

6. Exception Handling

- **Try-Catch**: For handling runtime errors and ensuring smooth program flow.

Example:

java

```
try {
   int division = 10 / 0;
} catch (ArithmeticException e) {
   System.out.println("Cannot divide by zero!");
} finally {
   System.out.println("Cleanup resources if any");
}
```

- **Throw**: Used to throw an exception manually.

java

```
throw new ArithmeticException("Division by zero");
```

Java OOP (Object-Oriented Programming) Concepts

1. Encapsulation

- **Definition**: Encapsulation is the bundling of data (variables) and methods that operate on that data within a class.
- **Access Modifiers**: Used to control the visibility of class members.
    - **Private**: Only accessible within the class.
    - **Public**: Accessible from any class.
    - **Protected**: Accessible in the same package or subclasses.
    - **Default**: Package-private access.

Example:

java

```
class Person {
   private String name;
   public void setName(String name) {
```

```
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

2. Inheritance

- **Definition**: Allows a class to inherit properties and behavior (methods) from another class.
- **Super keyword**: Used to refer to parent class's constructor or methods.

Example:

java

```
class Animal {
    void sound() { System.out.println("Animal sound"); }
}
class Dog extends Animal {
    void sound() { System.out.println("Bark"); }
}
Dog dog = new Dog();
dog.sound();  // Output: Bark
```

3. Polymorphism

- **Definition**: Ability of a class to take on many forms.
- **Compile-time (Method Overloading)**: Same method name but different signatures.

java

```
class Calculator {
    int add(int a, int b) { return a + b; }
    double add(double a, double b) { return a + b; }
}
```

- **Runtime (Method Overriding)**: Subclass provides specific implementation for a method defined in a parent class.

java

```
class Parent {
   void show() { System.out.println("Parent class"); }
}
class Child extends Parent {
   void show() { System.out.println("Child class"); }
}
```

4. Abstraction

- **Definition**: Hides implementation details and shows only functionality.
- **Abstract Class**: Can have abstract and non-abstract methods.

java

```
abstract class Vehicle {
   abstract void start();
}
class Car extends Vehicle {
   void start() { System.out.println("Car starts with key"); }
}
```

- **Interface**: A contract that enforces a set of methods for the implementing class.

java

```
interface Animal {
   void eat();
}
class Dog implements Animal {
   public void eat() { System.out.println("Dog eats"); }
}
```

---

5. Additional OOP Topics- Constructor, this & super keyword

Constructors:

- **Default constructor**: No arguments, provided by the compiler if not defined.
- **Parameterized constructor**: Constructor with parameters to initialize object properties.

Example:

java

```
class Employee {
   String name;
   Employee(String name) {
      this.name = name;
   }
}
```

this keyword:

- Refers to the current instance of a class.

java

```
class Car {
   String model;
   Car(String model) {
      this.model = model;  // Using 'this' to refer to current object
   }
}
```

super keyword:

- Refers to the parent class's members (variables/methods).

java

```
class Animal {
   String name = "Animal";
}
class Dog extends Animal {
   void display() {
      System.out.println(super.name);  // Output: Animal
```

```
    }
}
```

---

6. Static and Final keyword

Static keyword:

- **Static methods and variables**: Belong to the class rather than instances.

java

```
class Company {
    static String name = "itsruntym";
}
```

Final keyword:

- **Final variable**: Constant value.
- **Final method**: Cannot be overridden.
- **Final class**: Cannot be inherited.

---

7. String Handling in Java

- **String Class**: Immutable class that represents sequences of characters.
- Common Methods:
    - length(): Returns the length of the string.
    - charAt(int index): Returns the character at the specified index.
    - substring(int beginIndex, int endIndex): Extracts a portion of the string.
    - toLowerCase(), toUpperCase(): Converts to lower/uppercase.
    - equals(), equalsIgnoreCase(): Compares two strings.
    - concat(String str): Concatenates strings.

Example:

java

```
String str = "Java";
System.out.println(str.length());  // 4
```

System.out.println(str.substring(1, 3)); // "av"

System.out.println(str.toUpperCase()); // "JAVA"

8. StringBuilder and StringBuffer

- **StringBuilder**: Mutable sequence of characters (not thread-safe).
- **StringBuffer**: Similar to StringBuilder but is synchronized (thread-safe).

Example:

java

StringBuilder sb = new StringBuilder("Hello");

sb.append(" World");

System.out.println(sb); // "Hello World"

---

9. Java Memory Management

Stack vs Heap Memory:

- **Stack**: Stores local variables and method call details. Follows LIFO (Last In, First Out).
- **Heap**: Stores objects created using new. Managed by the garbage collector.

Garbage Collection:

- **Automatic memory management** that reclaims memory used by objects that are no longer referenced.
- **System.gc()** can be used to suggest garbage collection, but it's not guaranteed.

Memory Leaks:

- Occur when objects are no longer needed but are not collected due to lingering references.

---

10. Static Keyword

- **Static Variable**: Shared across all instances of the class.
- **Static Method**: Can be called without creating an instance of the class.
- **Static Block**: Executes when the class is loaded, typically used for static variable initialization.

Example:

java

class Demo {

```
    static int counter = 0;
    static void increment() { counter++; }
}
```

---

11. Final Keyword

- **Final Variable**: Constant, value cannot be changed.
- **Final Method**: Cannot be overridden by subclasses.
- **Final Class**: Cannot be extended.

Example:

java

```
final class Constants {
    static final double PI = 3.14159;
}
```

---

12. Inner Classes

- **Nested Class**: A class defined within another class.
- **Types**:
    - **Static Nested Class**: Can be instantiated without an outer class instance.
    - **Inner Class**: Requires an instance of the outer class.
    - **Local Inner Class**: Defined inside a method.
    - **Anonymous Inner Class**: A class without a name, used to override methods in-place.

Example:

java

```
class Outer {
    class Inner {
        void show() {
            System.out.println("Inside inner class");
        }
    }
```

```
}
Outer.Inner inner = new Outer().new Inner();
inner.show();
```

---

13. Java Collections Framework

- **Collection Interface**: Root interface for working with groups of objects.

List Interface:

- **ArrayList**: Resizable array, allows duplicate elements, dynamic resizing.
- **LinkedList**: Doubly linked list, better for frequent insertions/removals.

Example:

java

```
List<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
System.out.println(list.get(0));  // "Apple"
```

Set Interface:

- **HashSet**: Unordered, no duplicates, uses a hash table for storage.
- **TreeSet**: Ordered set, sorted elements.

Example:

java

```
Set<String> set = new HashSet<>();
set.add("Apple");
set.add("Apple");  // Duplicate, ignored
System.out.println(set);  // "Apple"
```

Map Interface:

- **HashMap**: Key-value pairs, unordered, allows null keys.
- **TreeMap**: Sorted key-value pairs.

Example:

java

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "One");
map.put(2, "Two");
System.out.println(map.get(1));  // "One"
```

---

14. Generics in Java

- **Generics**: Allow classes, interfaces, and methods to operate on any data type while providing type safety.

Example:

java

```
class Box<T> {
   private T value;
   public void set(T value) { this.value = value; }
   public T get() { return value; }
}
Box<Integer> integerBox = new Box<>();
integerBox.set(10);
System.out.println(integerBox.get());  // 10
```

---

15. Java Multithreading

Creating Threads:

- **By Extending Thread class**:

java

```
class MyThread extends Thread {
   public void run() {
      System.out.println("Thread running");
   }
}
```

```
MyThread thread = new MyThread();
thread.start();
```

- **By Implementing Runnable Interface**:

java

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread running");
    }
}
Thread thread = new Thread(new MyRunnable());
thread.start();
```

Thread Methods:

- start(), run(), sleep(), join(), yield().

---

16. Synchronization in Java

- **Synchronized**: Ensures that only one thread can access a resource at a time, preventing race conditions.

Example:

java

```
class Counter {
    private int count = 0;
    public synchronized void increment() {
        count++;
    }
    public int getCount() {
        return count;
    }
}
```

---

17. Java 8 Features

Lambdas and Functional Programming:

- **Lambda Expressions**: Simplified syntax for implementing functional interfaces.

Example:

java

```
List<Integer> numbers = Arrays.asList(1, 2, 3);
numbers.forEach(n -> System.out.println(n));
```

Streams API:

- **Stream**: A sequence of elements supporting sequential and parallel operations.

Example:

java

```
List<String> names = Arrays.asList("John", "Paul", "George", "Ringo");
names.stream().filter(n -> n.startsWith("J")).forEach(System.out::println);
```

Optional Class:

- **Optional**: Used to avoid null references.

Example:

java

```
Optional<String> optional = Optional.of("Hello");
optional.ifPresent(System.out::println);  // Prints "Hello"
```

---

18. Design Patterns in Java

- **Singleton Pattern**: Ensures that only one instance of a class is created.

Example:

java

```
class Singleton {
    private static Singleton instance = null;
    private Singleton() {}
```

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

- **Factory Pattern**: Provides an interface for creating objects but lets subclasses alter the type of objects that will be created.

Example:

java

```
interface Shape {
    void draw();
}
class Circle implements Shape {
    public void draw() { System.out.println("Drawing Circle"); }
}
class ShapeFactory {
    public Shape getShape(String shapeType) {
        if (shapeType.equals("CIRCLE")) {
            return new Circle();
        }
        return null;
    }
}
ShapeFactory factory = new ShapeFactory();
Shape shape = factory.getShape("CIRCLE");
shape.draw();  // "Drawing Circle"
```

---

19. Exception Handling (Expanded)

Checked vs Unchecked Exceptions:

- **Checked Exceptions**: Checked at compile time (e.g., IOException, SQLException).
- **Unchecked Exceptions**: Occur at runtime (e.g., NullPointerException, ArithmeticException).

Custom Exceptions:

- You can create custom exceptions by extending Exception or RuntimeException.

Example:

java

```
class CustomException extends Exception {
   CustomException(String message) {
      super(message);
   }
}
throw new CustomException("Custom error occurred");
```

---

20. Input/Output (I/O) in Java

- **Streams**: Java I/O is built around the concept of streams (InputStream, OutputStream).

File Handling:

- **File Class**: Used to represent file and directory pathnames.

Example:

java

```
File file = new File("file.txt");
if (file.exists()) {
   System.out.println("File exists");
}
```

BufferedReader and FileReader:

- For reading text from a file.

Example:

java

BufferedReader reader = new BufferedReader(new FileReader("file.txt"));

String line = reader.readLine();

PrintWriter:

- For writing text to a file.

Example:

java

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

writer.println("Hello, World!");

writer.close();