#_ The Ultimate Java CheatSheet

1. Basic Syntax and Data Types

- Main method: `public static void main(String[] args) { }`
- Print to console: `System.out.println("Hello, World!");`
- Print without newline: `System.out.print("Hello");`
- Formatted print: `System.out.printf("%.2f", 3.14159);`
- Read user input: `Scanner scanner = new Scanner(System.in);`
- Read integer: `int num = scanner.nextInt();`
- Read string: `String str = scanner.nextLine();`
- Integer declaration: `int num = 10;`
- Long declaration: `long bigNum = 1000000000L;`
- Short declaration: `short shortNum = 100;`
- Byte declaration: `byte b = 127;`
- Float declaration: `float f = 3.14f;`
- Double declaration: `double d = 3.14159;`
- Boolean declaration: `boolean isTrue = true;`
- Character declaration: `char c = 'A';`
- String declaration: `String str = "Hello";`
- Constant declaration: `final int MAX_VALUE = 100;`
- Type casting (widening): `long l = (long) 10;`
- Type casting (narrowing): `int i = (int) 3.14;`
- Auto-boxing: `Integer num = 10;`
- Unboxing: `int value = num;`
- Binary literal: `int binary = 0b1010;`
- Octal literal: `int octal = 012;`
- Hexadecimal literal: `int hex = 0xA;`
- Scientific notation: `double sci = 1.23e2;`

2. Operators

- Addition: `int sum = a + b;`
- Subtraction: `int diff = a - b;`
- Multiplication: `int product = a * b;`
- Division: `int quotient = a / b;`
- Modulus: `int remainder = a % b;`
- Increment (prefix): `++i;`
- Increment (postfix): `i++;`

- Decrement (prefix): `--i;`
- Decrement (postfix): `i--;`
- Addition assignment: `a += b;`
- Subtraction assignment: `a -= b;`
- Multiplication assignment: `a *= b;`
- Division assignment: `a /= b;`
- Modulus assignment: `a %= b;`
- Equality: `boolean isEqual = (a == b);`
- Inequality: `boolean isNotEqual = (a != b);`
- Greater than: `boolean isGreater = (a > b);`
- Less than: `boolean isLess = (a < b);`
- Greater than or equal to: `boolean isGreaterOrEqual = (a >= b);`
- Less than or equal to: `boolean isLessOrEqual = (a <= b);`
- Logical AND: `boolean result = (a && b);`
- Logical OR: `boolean result = (a || b);`
- Logical NOT: `boolean result = !a;`
- Bitwise AND: `int result = a & b;`
- Bitwise OR: `int result = a | b;`
- Bitwise XOR: `int result = a ^ b;`
- Bitwise complement: `int result = ~a;`
- Left shift: `int result = a << 2;`
- Right shift: `int result = a >> 2;`
- Unsigned right shift: `int result = a >>> 2;`
- Ternary operator: `int result = (condition) ? trueValue : falseValue;`

3. Control Flow

- If statement: `if (condition) { }`
- If-else statement: `if (condition) { } else { }`
- If-else if-else statement: `if (condition1) { } else if (condition2) { } else { }`
- Nested if: `if (condition1) { if (condition2) { } }`
- Switch statement: `switch (variable) { case value: break; default: break; }`
- For loop: `for (int i = 0; i < 10; i++) { }`
- Enhanced for loop: `for (String item : list) { }`
- While loop: `while (condition) { }`
- Do-while loop: `do { } while (condition);`
- Infinite loop: `while (true) { }`
- Break statement: `break;`

- Labeled break:

```
outerloop:
for (int i = 0; i < 5; i++) {
      for (int j = 0; j < 5; j++) {
      if (condition) break outerloop;
      }
}
```

- Switch expression (Java 14+):

```
int result = switch (variable) {
case 1, 2 -> 0;
case 3, 4 -> 1;
default -> -1;
};
```

- Continue statement: `continue;`
- Labeled continue:

```
outerloop:
for (int i = 0; i < 5; i++) {
      for (int j = 0; j < 5; j++) {
      if (condition) continue outerloop;
      }
}
```

- Return statement: `return value;`
- Yield statement (Java 13+): `yield value;`

4. Arrays

- Single-dimensional array declaration: `int[] numbers;`
- Array initialization: `int[] numbers = {1, 2, 3, 4, 5};`
- Array instantiation: `int[] numbers = new int[5];`
- Multidimensional array: `int[][] matrix = new int[3][3];`
- Jagged array: `int[][] jagged = new int[3][];`
- Get array length: `int length = numbers.length;`
- Access array element: `int element = numbers[0];`
- Set array element: `numbers[0] = 10;`
- Iterate over array: `for (int i = 0; i < numbers.length; i++) { }`
- Enhanced for loop for array: `for (int number : numbers) { }`
- Copy array: `int[] copy = Arrays.copyOf(original, original.length);`

- Copy range of array: `int[] copy = Arrays.copyOfRange(original, fromIndex, toIndex);`
- Fill array: `Arrays.fill(numbers, 0);`
- Sort array: `Arrays.sort(numbers);`
- Binary search: `int index = Arrays.binarySearch(numbers, key);`
- Compare arrays: `boolean isEqual = Arrays.equals(array1, array2);`
- Convert array to list: `List<Integer> list = Arrays.asList(numbers);`
- Print array: `System.out.println(Arrays.toString(numbers));`
- Print multidimensional array: `System.out.println(Arrays.deepToString(matrix));`
- Parallel sort: `Arrays.parallelSort(numbers);`
- Stream from array: `Arrays.stream(numbers)`
- Find max in array: `int max = Arrays.stream(numbers).max().getAsInt();`
- Find min in array: `int min = Arrays.stream(numbers).min().getAsInt();`
- Sum of array elements: `int sum = Arrays.stream(numbers).sum();`
- Average of array elements: `double avg = Arrays.stream(numbers).average().getAsDouble();`

5. Strings

- String declaration: `String str = "Hello";`
- String concatenation: `String result = str1 + str2;`
- String builder: `StringBuilder sb = new StringBuilder();`
- Append to string builder: `sb.append("text");`
- Insert into string builder: `sb.insert(0, "prefix");`
- Delete from string builder: `sb.delete(0, 5);`
- Replace in string builder: `sb.replace(0, 5, "new");`
- Reverse string builder: `sb.reverse();`
- Convert string builder to string: `String result = sb.toString();`
- String length: `int length = str.length();`
- Get character at index: `char c = str.charAt(0);`
- Substring: `String sub = str.substring(start, end);`
- String comparison: `boolean isEqual = str1.equals(str2);`
- Case-insensitive comparison: `boolean isEqual = str1.equalsIgnoreCase(str2);`
- Compare strings lexicographically: `int result = str1.compareTo(str2);`
- Convert to uppercase: `String upper = str.toUpperCase();`

- Convert to lowercase: `String lower = str.toLowerCase();`
- Trim whitespace: `String trimmed = str.trim();`
- Strip leading and trailing whitespace: `String stripped = str.strip();`
- Replace characters: `String replaced = str.replace('a', 'b');`
- Replace substring: `String replaced = str.replace("old", "new");`
- Replace all occurrences (regex): `String replaced = str.replaceAll("\\s+", " ");`
- Split string: `String[] parts = str.split(",");`
- Join strings: `String joined = String.join(", ", strings);`
- Check if string starts with: `boolean startsWith = str.startsWith("prefix");`
- Check if string ends with: `boolean endsWith = str.endsWith("suffix");`
- Check if string contains: `boolean contains = str.contains("substring");`
- Index of substring: `int index = str.indexOf("substring");`
- Last index of substring: `int lastIndex = str.lastIndexOf("substring");`
- Convert to char array: `char[] chars = str.toCharArray();`
- Create string from char array: `String str = new String(charArray);`
- Check if string is empty: `boolean isEmpty = str.isEmpty();`
- Check if string is blank: `boolean isBlank = str.isBlank();`
- Repeat string: `String repeated = str.repeat(3);`
- Format string: `String formatted = String.format("Hello, %s", name);`

6. Methods

- Method declaration: `public int add(int a, int b) { return a + b; }`
- Method overloading: `public int add(int a, int b, int c) { return a + b + c; }`
- Variable arguments: `public int sum(int... numbers) { }`
- Recursive method: `public int factorial(int n) { if (n <= 1) return 1; return n * factorial(n - 1); }`
- Method with default value: `public void greet(String name = "World") { }`
- Static method: `public static void staticMethod() { }`
- Instance method: `public void instanceMethod() { }`
- Abstract method: `public abstract void abstractMethod();`
- Final method: `public final void finalMethod() { }`

- Native method: `public native void nativeMethod();`
- Synchronized method: `public synchronized void synchronizedMethod() { }`

7. Object-Oriented Programming

- Class declaration: `public class ClassName { }`
- Constructor: `public ClassName() { }`
- Parameterized constructor: `public ClassName(int param) { }`
- Instance variable: `private int instanceVar;`
- Class variable (static): `private static int classVar;`
- Final variable: `private final int CONSTANT = 10;`
- Getter method: `public int getInstanceVar() { return instanceVar; }`
- Setter method: `public void setInstanceVar(int value) { this.instanceVar = value; }`
- Instance method: `public void instanceMethod() { }`
- Class method (static): `public static void classMethod() { }`
- Method with parameters: `public void method(int param1, String param2) { }`
- Method overloading: `public void method(int param) { }` and `public void method(String param) { }`
- Inheritance: `public class ChildClass extends ParentClass { }`
- Method overriding: `@Override public void parentMethod() { }`
- Abstract class: `public abstract class AbstractClassName { }`
- Abstract method: `public abstract void abstractMethod();`
- Interface: `public interface InterfaceName { }`
- Implement interface: `public class ClassName implements InterfaceName { }`
- Default method in interface: `default void defaultMethod() { }`
- Static method in interface: `static void staticMethod() { }`
- Multiple inheritance with interfaces: `public class ClassName implements Interface1, Interface2 { }`
- Nested class: `public static class NestedClassName { }`
- Inner class: `public class InnerClassName { }`
- Anonymous inner class: `new InterfaceName() { public void methodName() { } };`
- Enum: `public enum EnumName { VALUE1, VALUE2, VALUE3 }`
- Enum with constructor: `public enum EnumName { VALUE1(1), VALUE2(2); private final int value; EnumName(int value) { this.value = value; } }`

- Singleton pattern:

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

- Builder pattern:

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

      private Person(Builder builder) {
      this.name = builder.name;
      this.age = builder.age;
      }

      public static class Builder {
      private String name;
      private int age;

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

      public Builder age(int age) {
            this.age = age;
            return this;
      }

      public Person build() {
            return new Person(this);
      }
      }
```

```
}
```

- Factory method pattern:

```
public interface Shape {
      void draw();
}

public class ShapeFactory {
      public Shape getShape(String shapeType) {
      if (shapeType == null) {
            return null;
      }
      if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
      } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new Rectangle();
      }
      return null;
      }
}
```

8. Exception Handling

- Try-catch block: `try { } catch (Exception e) { }`
- Multiple catch blocks: `try { } catch (Exception1 e) { } catch (Exception2 e) { }`
- Try-catch-finally: `try { } catch (Exception e) { } finally { }`
- Try-with-resources: `try (Resource res = new Resource()) { }`
- Throw exception: `throw new Exception("Error message");`
- Throws clause: `public void methodName() throws Exception { }`
- Custom exception: `public class CustomException extends Exception { }`
- Get exception message: `String message = e.getMessage();`
- Print stack trace: `e.printStackTrace();`
- Catch multiple exceptions: `catch (Exception1 | Exception2 e) { }`
- Rethrowing exceptions: `catch (Exception e) { throw e; }`
- Get cause of exception: `Throwable cause = e.getCause();`
- Assert statement: `assert condition : "Error message";`

9. Collections Framework

- ArrayList: `List<String> list = new ArrayList<>();`

- LinkedList: `List<String> linkedList = new LinkedList<>();`
- HashSet: `Set<String> set = new HashSet<>();`
- TreeSet: `Set<String> treeSet = new TreeSet<>();`
- LinkedHashSet: `Set<String> linkedHashSet = new LinkedHashSet<>();`
- HashMap: `Map<String, Integer> map = new HashMap<>();`
- TreeMap: `Map<String, Integer> treeMap = new TreeMap<>();`
- LinkedHashMap: `Map<String, Integer> linkedHashMap = new LinkedHashMap<>();`
- Queue: `Queue<String> queue = new LinkedList<>();`
- Deque: `Deque<String> deque = new ArrayDeque<>();`
- PriorityQueue: `PriorityQueue<Integer> pq = new PriorityQueue<>();`
- Stack: `Stack<String> stack = new Stack<>();`
- Add element: `list.add("element");`
- Add element at index: `list.add(0, "element");`
- Remove element: `list.remove("element");`
- Remove element at index: `list.remove(0);`
- Get element: `String element = list.get(0);`
- Set element: `list.set(0, "new element");`
- Check if contains: `boolean contains = list.contains("element");`
- Size of collection: `int size = list.size();`
- Clear collection: `list.clear();`
- Check if empty: `boolean isEmpty = list.isEmpty();`
- Iterate over collection: `for (String item : list) { }`
- Iterator: `Iterator<String> iterator = list.iterator();`
- List iterator: `ListIterator<String> listIterator = list.listIterator();`
- Sort list: `Collections.sort(list);`
- Reverse list: `Collections.reverse(list);`
- Shuffle list: `Collections.shuffle(list);`
- Binary search: `int index = Collections.binarySearch(list, "element");`
- Find min element: `String min = Collections.min(list);`
- Find max element: `String max = Collections.max(list);`
- Fill list with element: `Collections.fill(list, "element");`
- Frequency of element: `int frequency = Collections.frequency(list, "element");`
- Disjoint collections: `boolean isDisjoint = Collections.disjoint(collection1, collection2);`
- Unmodifiable list: `List<String> unmodifiableList = Collections.unmodifiableList(list);`