Java Cheatsheet

Essential operations for Java programming and development

This cheatsheet provides a quick reference to fundamental Java operations, syntax, and core features, ideal for beginners learning Java programming and building foundational coding skills.

Basic Syntax
Core Java language structure

Data Types
Variables and primitive types

Control Flow
Loops and conditional statements

OOP Concepts
Classes, objects, and inheritance

Collections
Arrays, Lists, and data structures

Program Structure & Basic Syntax

Hello World: Basic Program

The simplest Java program that displays "Hello, World!" on the screen.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Class Declaration: `public class`

A class is a template/blueprint that describes the behavior/state that objects support.

```
public class MyClass {
    // Class content goes here
    int myVariable;

    public void myMethod() {
        System.out.println("Hello from method!");
    }
}
```

Main Method: Program Entry Point

The main method is where Java program execution begins.

```
public static void main(String[] args) {
    // Program code here
    System.out.println("Program starts here");
}
```

Comments: Code Documentation

Use single-line (//) and multi-line (/* */) comments to make code more understandable and maintainable.

```
// Single-line comment
System.out.println("Hello");

/* Multi-line comment
   Can span multiple lines
   Used for detailed explanations */
```

Statements & Semicolons

Each statement in Java must end with a semicolon.

```
int number = 10;
String name = "Java";
System.out.println(name);
```

Code Blocks: Curly Braces

Blocks of code are enclosed in curly braces {}, marking the beginning and end of code sections.

```
public class Example {
    public void method() {
        if (true) {
            System.out.println("Inside if block");
        }
    }
}
```

Data Types & Variables

Variables are basic storage units that hold data that can be modified during program execution, with specific data types that dictate memory size and operations.

01

Primitive Data Types

Basic data types built into Java language.

```
// Integer types
byte smallNum = 127;        // -128 to 127
short shortNum = 32000;     // -32,768 to 32,767
int number = 100;           // -2^31 to 2^31-1
long bigNum = 10000L;       // -2^63 to 2^63-1

// Floating point types
float decimal = 3.14f;      // Single precision
double precision = 3.14159; // Double precision

// Other types
char letter = 'A';          // Single character
boolean flag = true;        // true or false
```

02

Variable Declaration & Initialization

Creating and assigning values to variables.

```
// Declaration only
int age;
String name;

// Declaration with initialization
int age = 25;
String name = "John";

// Multiple declarations
int x = 10, y = 20, z = 30;

// Final variables (constants)
final double PI = 3.14159;
```

03

String Operations

Strings represent sequences of characters and are immutable, meaning once created, their values cannot be changed.

```
String greeting = "Hello";
String name = "World";

// String concatenation
String message = greeting + " " + name;
System.out.println(message); // "Hello World"

// String methods
int length = message.length();
boolean isEmpty = message.isEmpty();
String uppercase = message.toUpperCase();
```

Control Flow Statements

Conditional Statements: `if`, `else if`, `else`

Execute different code blocks based on conditions.

```
int score = 85;

if (score >= 90) {
    System.out.println("Grade A");
} else if (score >= 80) {
    System.out.println("Grade B");
} else if (score >= 70) {
    System.out.println("Grade C");
} else {
    System.out.println("Grade F");
}
```

Switch Statement

Multi-way branching based on variable values.

```
int day = 3;
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    default:
        System.out.println("Other day");
}
```

For Loop: Counted Repetition

Repeat code a specific number of times.

```
// Standard for loop
for (int i = 0; i < 5; i++) {
    System.out.println("Count: " + i);
}

// Enhanced for loop (for-each)
int[] numbers = {1, 2, 3, 4, 5};
for (int num : numbers) {
    System.out.println("Number: " + num);
}
```

While & Do-While Loops

Repeat code while a condition is true.

```
// While loop
int i = 0;
while (i < 3) {
    System.out.println("While: " + i);
    i++;
}

// Do-while loop (executes at least once)
int j = 0;
do {
    System.out.println("Do-while: " + j);
    j++;
} while (j < 3);
```

Object-Oriented Programming

Classes & Objects

Objects have states and behaviors. An object is an instance of a class.

```
public class Car {
    // Instance variables (state)
    String color;
    String model;
    int year;

    // Constructor
    public Car(String color, String model, int year) {
        this.color = color;
        this.model = model;
        this.year = year;
    }

    // Method (behavior)
    public void start() {
        System.out.println("Car is starting...");
    }
}

// Creating objects
Car myCar = new Car("Red", "Toyota", 2022);
myCar.start();
```

Constructors

Special methods used to initialize objects.

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

    // Default constructor
    public Person() {
        name = "Unknown";
        age = 0;
    }

    // Parameterized constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Inheritance: `extends`

Inheritance enables code reuse and creates hierarchical relationships between classes.

```
public class Animal {
    protected String name;

    public void eat() {
        System.out.println(name + " is eating");
    }
}

public class Dog extends Animal {
    public Dog(String name) {
        this.name = name;
    }

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

Dog myDog = new Dog("Buddy");
myDog.eat();  // Inherited method
myDog.bark(); // Own method
```

Access Modifiers

Modifiers control access to classes, methods, and variables.

```
public class Example {
    public int publicVar;      // Accessible everywhere
    private int privateVar;    // Only within this class
    protected int protectedVar; // Within package + subclasses
    int defaultVar;            // Within package only

    private void privateMethod() {
        // Only accessible within this class
    }
}
```

Methods & Functions

Method Declaration

A method is basically a behavior where logics are written, data is manipulated and actions are executed.

```
public class Calculator {
    // Method with parameters and return value
    public int add(int a, int b) {
        return a + b;
    }

    // Method with no return value
    public void printSum(int a, int b) {
        int result = add(a, b);
        System.out.println("Sum: " + result);
    }

    // Static method (belongs to class)
    public static int multiply(int a, int b) {
        return a * b;
    }
}
```

Method Overloading

Multiple methods with same name but different parameters.

```
public class MathUtils {
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }

    public int add(int a, int b, int c) {
        return a + b + c;
    }
}
```

Method Parameters & Return Types

Pass data to methods and return results.

```
public class StringHelper {
    // Method with String parameter and return
    public String formatName(String firstName, String lastName) {
        return firstName + " " + lastName;
    }

    // Method with array parameter
    public int findMax(int[] numbers) {
        int max = numbers[0];
        for (int num : numbers) {
            if (num > max) {
                max = num;
            }
        }
        return max;
    }
}
```

Recursive Methods

Methods that call themselves to solve problems.

```
public class RecursiveExamples {
    // Calculate factorial
    public int factorial(int n) {
        if (n <= 1) {
            return 1;
        }
        return n * factorial(n - 1);
    }

    // Fibonacci sequence
    public int fibonacci(int n) {
        if (n <= 1) {
            return n;
        }
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Arrays & Collections

Arrays are containers that store multiple values of the same data type, providing efficient access to collections of data using index-based positions.

Array Declaration & Initialization

Create and initialize arrays of different types.

```
// Array declaration and initialization
int[] numbers = {1, 2, 3, 4, 5};
String[] names = {"Alice", "Bob", "Charlie"};

// Array with specified size
int[] scores = new int[10];
scores[0] = 95;
scores[1] = 87;

// Getting array length
int length = numbers.length;
System.out.println("Length: " + length);

// Loop through array
for (int i = 0; i < numbers.length; i++) {
    System.out.println("Element " + i + ": " + numbers[i]);
}
```

Multi-dimensional Arrays

Arrays of arrays for matrix-like data structures.

```
// 2D array declaration
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

// Access elements
int element = matrix[1][2]; // Gets 6

// Loop through 2D array
for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
        System.out.print(matrix[i][j] + " ");
    }
    System.out.println();
}
```

ArrayList: Dynamic Arrays

Resizable arrays that can grow and shrink dynamically.

```
import java.util.ArrayList;

// Create ArrayList
ArrayList list = new ArrayList<>();

// Add elements
list.add("Apple");
list.add("Banana");
list.add("Orange");

// Get element
String fruit = list.get(0); // Gets "Apple"

// Remove element
list.remove(1); // Removes "Banana"

// Size and iteration
System.out.println("Size: " + list.size());
for (String item : list) {
    System.out.println(item);
}
```

HashMap: Key-Value Pairs

Store data as key-value pairs for fast lookup.

```
import java.util.HashMap;

// Create HashMap
HashMap ages = new HashMap<>();

// Add key-value pairs
ages.put("Alice", 25);
ages.put("Bob", 30);
ages.put("Charlie", 35);

// Get value by key
int aliceAge = ages.get("Alice");

// Check if key exists
if (ages.containsKey("Bob")) {
    System.out.println("Bob's age: " + ages.get("Bob"));
}
```

Exception Handling

Exception handling is a mechanism to handle runtime errors, ensuring the program runs smoothly without crashing using keywords like try, catch, throw, throws, and finally.

Try-Catch Blocks

Handle exceptions to prevent program crashes.

```
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // This will throw ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
            System.out.println("Error: " + e.getMessage());
        } finally {
            System.out.println("This always executes");
        }
    }
}
```

Multiple Catch Blocks

Handle different types of exceptions separately.

```
public void processArray(String[] arr, int index) {
    try {
        int number = Integer.parseInt(arr[index]);
        int result = 100 / number;
        System.out.println("Result: " + result);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Invalid array index");
    } catch (NumberFormatException e) {
        System.out.println("Invalid number format");
    } catch (ArithmeticException e) {
        System.out.println("Cannot divide by zero");
    }
}
```

Throwing Custom Exceptions

Create and throw your own exceptions.

```
public class AgeValidator {
    public void validateAge(int age) throws IllegalArgumentException {
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative");
        }
        if (age > 150) {
            throw new IllegalArgumentException("Age seems unrealistic");
        }
        System.out.println("Valid age: " + age);
    }

    public static void main(String[] args) {
        AgeValidator validator = new AgeValidator();
        try {
            validator.validateAge(-5);
        } catch (IllegalArgumentException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Common Exception Types

Frequently encountered exceptions in Java programs.

```
// NullPointerException
String str = null;
// str.length(); // Throws NullPointerException

// ArrayIndexOutOfBoundsException
int[] arr = {1, 2, 3};
// int val = arr[5]; // Throws ArrayIndexOutOfBoundsException

// NumberFormatException
// int num = Integer.parseInt("abc"); // Throws NumberFormatException

// FileNotFoundException (when working with files)
// IOException (general I/O operations)
```

Input/Output Operations

Java File Handling enables programs to create, read, write, and manipulate files using classes from java.io packages.

Console Input: Scanner Class

Read input from the keyboard using Scanner.

```
import java.util.Scanner;

public class InputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter your name: ");
        String name = scanner.nextLine();

        System.out.print("Enter your age: ");
        int age = scanner.nextInt();

        System.out.print("Enter your height: ");
        double height = scanner.nextDouble();

        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Height: " + height);

        scanner.close();
    }
}
```

Console Output: System.out

Display output to the console in various formats.

```
public class OutputExample {
    public static void main(String[] args) {
        // Basic output
        System.out.println("Hello, World!");
        System.out.print("No newline");
        System.out.print(" continues here\n");

        // Formatted output
        String name = "Java";
        int version = 17;
        System.out.printf("Welcome to %s %d!%n", name, version);

        // Output variables
        int x = 10, y = 20;
        System.out.println("x = " + x + ", y = " + y);
        System.out.println("Sum = " + (x + y));
    }
}
```

File Reading: BufferedReader

Read text files line by line efficiently.

```
import java.io.*;

public class FileReadExample {
    public static void readFile(String filename) {
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line;
            int lineNumber = 1;

            while ((line = reader.readLine()) != null) {
                System.out.println(lineNumber + ": " + line);
                lineNumber++;
            }
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
        }
    }
}
```

File Writing: PrintWriter

Write text data to files with proper exception handling.

```
import java.io.*;

public class FileWriteExample {
    public static void writeFile(String filename, String[] data) {
        try (PrintWriter writer = new PrintWriter(new FileWriter(filename))) {
            writer.println("# Data File");
            writer.println("Generated on: " + new java.util.Date());
            writer.println();

            for (int i = 0; i < data.length; i++) {
                writer.println("Line " + (i + 1) + ": " + data[i]);
            }

            System.out.println("File written successfully!");
        } catch (IOException e) {
            System.out.println("Error writing file: " + e.getMessage());
        }
    }
}
```

Java Development Environment

To run Java, you need to set up the Java environment properly, including configuring required environment variables such as PATH and JAVA_HOME.

JDK Installation

JDK (Java Development Kit) = JRE + Development Tools. Required for developing Java applications.

```
# Download JDK from Oracle or OpenJDK
# Install JDK on your system
# Set JAVA_HOME environment variable
export JAVA_HOME=/path/to/jdk
export PATH=$JAVA_HOME/bin:$PATH

# Verify installation
java -version
javac -version
```

Compile & Run Java Programs

Use javac to compile Java source code and java to run the compiled program.

```
# Compile Java source file
javac MyProgram.java

# Run compiled Java program
java MyProgram

# Compile with classpath
javac -cp .:mylib.jar MyProgram.java

# Run with classpath
java -cp .:mylib.jar MyProgram
```

IDE Setup & Development

Popular Integrated Development Environments for Java development.

```
# Popular Java IDEs:
# - IntelliJ IDEA (JetBrains)
# - Eclipse IDE
# - Visual Studio Code with Java extensions
# - NetBeans

# Command line compilation
javac -d bin src/*.java
java -cp bin MainClass

# JAR file creation
jar cf myapp.jar -C bin .
```

Best Practices & Common Patterns

Java is case-sensitive, class names should start with uppercase letters, and following naming conventions improves code readability.

Naming Conventions

Follow Java naming standards for better code readability.

```
// Classes: PascalCase
public class StudentManager { }
public class BankAccount { }

// Methods and variables: camelCase
int studentAge;
String firstName;
public void calculateGrade() { }
public boolean isValidEmail() { }

// Constants: UPPER_CASE
public static final int MAX_SIZE = 100;
public static final String DEFAULT_NAME = "Unknown";

// Packages: lowercase
package com.company.project;
package utils.database;
```

Code Organization

Structure your Java programs for maintainability.

```
package com.example.myapp;

import java.util.ArrayList;
import java.util.Scanner;

/**
 * This class demonstrates good Java code organization
 * @author Your Name
 * @version 1.0
 */
public class WellOrganizedClass {
    // Constants first
    private static final int MAX_ATTEMPTS = 3;

    // Instance variables
    private String name;
    private int value;

    // Constructor
    public WellOrganizedClass(String name) {
        this.name = name;
        this.value = 0;
    }

    // Public methods
    public void doSomething() {
        // Implementation
    }

    // Private helper methods
    private boolean isValid() {
        return value > 0;
    }
}
```

Error Prevention

Common practices to avoid bugs and improve code quality.

```
public class BestPractices {
    public void safeDivision(int a, int b) {
        // Check for division by zero
        if (b == 0) {
            throw new IllegalArgumentException("Cannot divide by zero");
        }
        int result = a / b;
        System.out.println("Result: " + result);
    }

    public void safeStringOperations(String input) {
        // Null check before using strings
        if (input != null && !input.isEmpty()) {
            System.out.println("Length: " + input.length());
            System.out.println("Uppercase: " + input.toUpperCase());
        } else {
            System.out.println("Invalid input string");
        }
    }

    public void safeArrayAccess(int[] array, int index) {
        // Bounds checking
        if (array != null && index >= 0 && index < array.length) {
            System.out.println("Value: " + array[index]);
        } else {
            System.out.println("Invalid array access");
        }
    }
}
```

Resource Management

Properly handle resources to prevent memory leaks.

```
import java.io.*;

public class ResourceManagement {
    // Try-with-resources (automatic cleanup)
    public void readFileProper(String filename) {
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line = reader.readLine();
            System.out.println(line);
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
        // Reader is automatically closed
    }

    // Manual resource cleanup (not recommended)
    public void readFileManual(String filename) {
        BufferedReader reader = null;
        try {
            reader = new BufferedReader(new FileReader(filename));
            String line = reader.readLine();
            System.out.println(line);
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            if (reader != null) {
                try {
                    reader.close();
                } catch (IOException e) {
                    System.out.println("Error closing reader");
                }
            }
        }
    }
}
```

Reference: This cheatsheet covers essential Java programming concepts and modern practices for building robust applications and mastering Java development fundamentals.