Core Java

TABLE OF CONTENTS

PREFACE

This cheatsheet is designed to provide you with concise and practical information on the core concepts and syntax of Java. It's intended to serve as a handy reference that you can keep at your desk, save on your device, or print out and pin to your wall.

INTRODUCTION

Java is one of the most popular programming languages in the world, used in million of devices from servers and workstations to tablets, mobile phones and wearables. With this cheatsheet we strive to provide the main concepts and key aspects of the Java programming language. We include details on core language features such as lambda expressions, collections, formatted output, regular expressions, logging, properties as well as the most commonly used tools to compile, package and execute java programs.

JAVA KEYWORDS

Let's start with the basics, what follows is a list containing some of the most commonly used Java keywords, their meanings, and short examples. Please note that this is not an exhaustive list of all Java keywords, but it covers many of the commonly used ones. So be sure to consult the documentation for the most up-to-date information.

Keyword	Meaning	Example
`abstract`	Used to declare abstract classes	`abstract class Shape { /*...*/ }`
`assert`	Used for testing assertions	`assert (x > 0) : "x is not positive";`
`boolean`	Data type for true/false values	`boolean isJavaFun = true;`
`break`	Used to exit a loop or switch statement	`for (int i = 0; i < 5; i++) { if (i == 3) break; }`
`byte`	Data type for 8-bit integers	`byte b = 42;`
`case`	Used in a switch statement	`switch (day) { case 1: /*...*/ break; }`
`catch`	Used to catch exceptions	`try { /*...*/ } catch (Exception e) { /*...*/ }`
`char`	Data type for character data	`char grade = 'A';`
`class`	Declares a class	`class MyClass { /*...*/ }`
`continue`	Skips the current iteration in a loop	`for (int i = 0; i < 5; i++) { if (i == 3) continue; }`
`default`	Used in a switch statement	`switch (day) { case 1: /*...*/ default: /*...*/ }`
`do`	Starts a do-while loop	`do { /*...*/ } while (condition);`
`double`	Data type for double-precision floating-point numbers	`double pi = 3.14159265359;`
`else`	Used in an if-else statement	`if (x > 5) { /*...*/ } else { /*...*/ }`
`enum`	Declares an enumerated type	`enum Day { MONDAY, TUESDAY, WEDNESDAY }`
`extends`	Indicates inheritance in a class	`class ChildClass extends ParentClass { /*...*/ }`
`final`	Used to make a variable or method final	`final int maxAttempts = 3;`
`finally`	Used in exception handling	`try { /*...*/ } catch (Exception e) { /*...*/ } finally { /*...*/ }`
`float`	Data type for single-precision floating-point numbers	`float price = 19.99f;`
`for`	Starts a for loop	`for (int i = 0; i < 5; i++) { /*...*/ }`

Keyword	Meaning	Example
`if`	Conditional statement	`if (x > 5) { /…/ }`
`implements`	Implements an interface in a class	`class MyClass implements MyInterface { /…/ }`
`import`	Used to import packages and classes	`import java.util.ArrayList;`
`instanceof`	Checks if an object is an instance of a class	`if (obj instanceof MyClass) { /…/ }`
`int`	Data type for integers	`int count = 42;`
`interface`	Declares an interface	`interface MyInterface { /…/ }`
`long`	Data type for long integers	`long bigNumber = 1234567890L;`
`native`	Used in native method declarations	`native void myMethod();`
`new`	Creates a new object	`MyClass obj = new MyClass();`
`null`	Represents the absence of a value	`Object obj = null;`
`package`	Declares a Java package	`package com.example.myapp;`
`private`	Access modifier for private members	`private int age;`
`protected`	Access modifier for protected members	`protected String name;`
`public`	Access modifier for public members	`public void displayInfo() { /…/ }`
`return`	Returns a value from a method	`return result;`
`short`	Data type for short integers	`short smallNumber = 100;`

Keyword	Meaning	Example
`static`	Indicates a static member	`static int count = 0;`
`strictfp`	Ensures strict floating-point precision	`strictfp void myMethod() { /…/ }`
`super`	Calls a superclass constructor/method	`super();`
`switch`	Starts a switch statement	`switch (day) { case 1: /…/ }`
`synchronized`	Ensures thread safety	`synchronized void myMethod() { /…/ }`
`this`	Refers to the current instance	`this.name = name;`
`throw`	Throws an exception	`throw new Exception("Something went wrong.");`
`throws`	Declares exceptions thrown by a method	`void myMethod() throws MyException { /…/ }`
`transient`	Used with object serialization	`transient int sessionId;`
`try`	Starts an exception-handling block	`try { /…/ } catch (Exception e) { /…/ }`
`void`	Denotes a method that returns no value	`public void doSomething() { /…/ }`
`volatile`	Indicates that a variable may be modified by multiple threads	`volatile int sharedVar;`
`while`	Starts a loop	`while (in.hasNext()) process(in.next());`

JAVA PACKAGES

In Java, a package is a mechanism for organizing and grouping related classes and interfaces into a single namespace. Below you can find some of the most commonly used and the functionality their classes provide. Java has a rich ecosystem of libraries and frameworks, so the packages you use may vary depending on your specific application or project. Remember to import the necessary packages at the beginning of your Java classes to access their functionality.

Package	Description
`java.lang`	Provides fundamental classes (e.g., `String`, `Object`) and is automatically imported into all Java programs.
`java.util`	Contains utility classes for data structures (e.g., `ArrayList`, `HashMap`), date/time handling (`Date`, `Calendar`), and more.
`java.io`	Provides classes for input and output operations, including reading/writing files (`FileInputStream`, `FileOutputStream`) and streams (`InputStream`, `OutputStream`).
`java.net`	Used for network programming, including classes for creating and managing network connections (`Socket`, `ServerSocket`).
`java.awt`	Abstract Window Toolkit (AWT) for building graphical user interfaces (GUIs) in desktop applications.
`javax.swing`	Part of the Swing framework, an advanced GUI toolkit for building modern, platform-independent desktop applications.
`java.sql`	Provides classes for database access using JDBC (Java Database Connectivity).
`java.util.concurrent`	Contains classes and interfaces for handling concurrency, including thread management and synchronization (`Executor`, `Semaphore`).
`java.nio`	New I/O (NIO) package for efficient I/O operations, including non-blocking I/O and memory-mapped files (`ByteBuffer`, `FileChannel`).
`java.text`	Offers classes for text formatting (`DateFormat`, `NumberFormat`) and parsing.
`java.security`	Provides classes for implementing security features like encryption, authentication, and permissions.
`java.math`	Contains classes for arbitrary-precision arithmetic (`BigInteger`, `BigDecimal`).
`java.util.regex`	Supports regular expressions for pattern matching (`Pattern`, `Matcher`).
`java.awt.event`	Event handling in AWT, used to handle user-generated events such as button clicks and key presses.
`java.util.stream`	Introduces the Stream API for processing collections of data in a functional and declarative way.

Package	Description
`java.time`	The Java Date and Time API introduced in Java 8, for modern date and time handling (`LocalDate`, `Instant`, `DateTimeFormatter`).
`java.util.logging`	Provides a logging framework for recording application log messages.
`javax.servlet`	Contains classes and interfaces for building Java-based web applications using the Servlet API.
`javax.xml`	Offers XML processing capabilities, including parsing, validation, and transformation.

JAVA OPERATORS

Operators are a cornerstone aspect to every programming language. They allow you to manipulate data, perform calculations, and make decisions in your programs. Java provides a wide range of operators. Following are the ones most commonly used. The descriptions provide a brief overview of their purposes and usage. The operators at the top of the table have higher precedence, meaning they are evaluated first in expressions. Operators with the same precedence level are evaluated left to right. Be sure to use parentheses to clarify the order of evaluation when needed.

Operator	Example	Description
`()`	`(x + y)`	Parentheses for grouping expressions.
`++ --`	`x++`, `--y`	Increment and decrement operators.
`+ -`	`x + y`, `x - y`	Addition and subtraction.
`* / %`	`x * y`, `x / y`, `x % y`	Multiplication, division, and modulo.
`+` (unary)	`+x`, `-y`	Unary plus and minus.
`!`	`!flag`	Logical NOT (negation).
`~`	`~x`	Bitwise NOT (complement).
`<< >> >>>`	`x << 1`, `x >> 2`, `x >>> 3`	Bitwise left shift, right shift (with sign extension), and right shift (zero extension).
`< <= > >=`	`x < y`, `x <= y`, `x > y`, `x >= y`	Relational operators for comparisons.
`== !=`	`x == y`, `x != y`	Equality and inequality comparisons.
`&`	`x & y`	Bitwise AND.
`^`	`x ^ y`	Bitwise XOR (exclusive OR).
`&&`	`x && y`	Conditional AND (short-circuit).
`? :`	`result = (x > y) ? x : y`	Conditional (Ternary) Operator.
`=`	`x = y`	Assignment operator.
`+= -= *= /= %=`	`x += y`, `x -= y`, `x *= y`, `x /= y`, `x %= y`	Compound assignment operators.
`<<= >>= >>>=`	`x <<= 1`, `x >>= 2`, `x >>>= 3`	Compound assignment with bitwise shifts.

PRIMITIVE TYPES

In Java, primitive types (also known as primitive data types or simply primitives) are the simplest data types used to represent single values. These types are built into the language itself and are not objects like instances of classes and reference types. Java’s primitive types are designed for efficiency and are used to store basic values in memory. Below is a table listing all of Java’s primitive data types, their size in bytes, their range, and some additional notes. Keep in mind that you can convert from `byte` to `short`, `short` to `int`, `char` to `int`, `int` to `long`, `int` to `double` and `float` to `double` without loosing precision since the source type is smaller in size compared to the target one. On the other hand converting from `int` to `float`, `long` to `float` or `long` to `double` may come with a precision loss.

Data Type	Size (bytes)	Range	Notes
`byte`	1	-128 to 127	Used for small integer values.
`short`	2	-32,768 to 32,767	Useful for integers within this range.
`int`	4	-2^31 to 2^31 - 1	Commonly used for integer arithmetic.
`long`	8	-2^63 to 2^63 - 1	Used for larger integer values.
`float`	4	Approximately ±3.40282347E+38	Floating-point with single precision.
`double`	8	Approximately ±1.7976931348623157E+308	Floating-point with double precision.
`char`	2	0 to 65,535 (Unicode characters)	Represents a single character.
`boolean`		`true` or `false`	Represents true/false values.

LAMBDA EXPRESSIONS

Lambda expressions provide a way to define small, self-contained, and unnamed functions (anonymous functions) right where they are needed in your code. They are primarily used with functional interfaces, which are interfaces with a single abstract method. Lambda expressions provide a way to implement the abstract method of such interfaces without explicitly defining a separate class or method.

The syntax for lambda expressions is concise and consists of parameters surrounded by parenthesis (`()`), an arrow (`->`), and a body. The body can be an expression or a block of code.

```
// Using a lambda expression to
define a Runnable
Runnable runnable = () -> System.
out.println("Hello, Lambda!");
```

FUNCTIONAL INTERFACES

A functional interface is an interface that has exactly one abstract method, which is used to define a single unit of behavior. Below is an example of such an interface:

```
@FunctionalInterface
interface Calculator {
    int calculate(int a, int b);
}
```

Implementations of this interface can be supplied in-line as a lambda expression, as shown below:

```
// Using a lambda expression to
define addition
Calculator addition = (a, b) -> a +
b;
int result1 = addition.calculate(5,
3);
System.out.println("Addition: " +
result1); // Output: Addition: 8

// Using a lambda expression to
define subtraction
```

```
Calculator subtraction = (a, b) -> a
- b;
int result2 = subtraction.calculate
(10, 4);
System.out.println("Subtraction: " +
result2); // Output: Subtraction: 6
```

METHOD REFERENCES

Method references provide a concise way to reference static methods, constructors, or instance methods of objects, pass them as method parameters or return them, much like anonymous functions; making your code more readable and expressive when working with functional interfaces. Let's update the calculator example to use method references:

```
public class
MethodReferenceCalculator {
    // A static method that performs
addition
    public static int add(int a, int
b) {
        return a + b;
    }

    public static void main(String[]
args) {
        // Using a method reference
to the static add method
        Calculator addition =
MethodReferenceCalculator::add;

        int result = addition
.calculate(5, 3);
        System.out.println("Addition
result: " + result); // Output:
Addition result: 8
    }
}
```

COLLECTIONS

Collections play a pivotal role in Java development. Following are some of the most commonly used collection implementation classes along with code snippets illustrating typical use cases. Depending on your specific requirements, you can choose the appropriate one to suit your data storage and retrieval needs.

ARRAYLIST

An array-based list that dynamically resizes as elements are added or removed.

```
// Create an ArrayList and add
elements.
List<String> names = new
ArrayList<>();
names.add("Alice");
names.add("Bob");

// Iterate through the ArrayList.
for (String name : names) { /*...*/
}

// Add an element by index.
names.set(i, "Charlie");

// Remove an element by index.
names.remove(0);

// Add multiple elements using
addAll.
names.addAll(Arrays.asList("David",
"Eve"));

// Remove multiple elements using
removeAll.
names.removeAll(Arrays.asList("Alice
", "Bob"));

// Retrieve an element by index.
String firstPerson = names.get(0);

// Convert ArrayList to an array.
String[] namesArray = names.toArray
(new String[0]);

// Convert an array to an ArrayList.
List<String> namesList = Arrays
.asList(namesArray);

// Sort elements in natural order.
Collections.sort(names);
```

```
// Sort elements using a custom
comparator.
Collections.sort(names, new
Comparator<String>() { public int
compare(String a, String b) { return
a.length() - b.length(); } });

// Do something with all elements in
the collection
names.forEach(System.out::println);

// Use the Stream API to filter
elements.
List<String> filteredNames = names
.stream().filter(name -> name
.startsWith("A")).collect(Collectors
.toList());

// Use the Stream API to perform
calculations on elements (e.g., sum
of lengths).
int sumOfLengths = names.stream
().mapToInt(String::length).sum();
```

LINKEDLIST

A doubly-linked list that allows efficient insertion and removal from both ends.

```
LinkedList<Integer> numbers = new
LinkedList<>();
numbers.addFirst(1);
numbers.addLast(2);
numbers.removeFirst();
```

HASHSET

An unordered collection of unique elements, implemented using a hash table.

```
// Create a HashSet and add unique
elements.
HashSet<String> uniqueWords = new
HashSet<>();
uniqueWords.add("apple");
uniqueWords.add("banana");
```

```
// Iterate through the HashSet.
for (String word : uniqueWords) {
/*...*/ }

// Remove an element from the
HashSet.
uniqueWords.remove("apple");

// Use the Stream API to check if an
element exists.
boolean containsBanana =
uniqueWords.stream().anyMatch(word
-> word.equals("banana"));
```

TREESET

A sorted set that stores elements in natural order or according to a specified comparator.

```
// Create a TreeSet and add elements
(sorted automatically).
TreeSet<Integer> sortedNumbers = new
TreeSet<>();
sortedNumbers.add(3);
sortedNumbers.add(1);
sortedNumbers.add(2);

// Iterate through the TreeSet (in
ascending order).
for (Integer number : sortedNumbers)
{ /*...*/ }

// Use the Stream API to find the
maximum element.
Optional<Integer> maxNumber =
sortedNumbers.stream().max(Integer::
compareTo);
```

HASHMAP

An unordered collection of key-value pairs, implemented using a hash table.

```
// Create a HashMap and add key-
value pairs.
HashMap<String, Integer> ageMap =
new HashMap<>();
ageMap.put("Alice", 25);
ageMap.put("Bob", 30);

// Iterate through the HashMap key-
value pairs.
for (Map.Entry<String, Integer>
entry : ageMap.entrySet()) { /*...*/
}

// Remove a key-value pair by key.
ageMap.remove("Alice");

// Use the Stream API to transform
key-value pairs.
Map<String, Integer> doubledAgeMap =
ageMap.entrySet().stream().collect(C
ollectors.toMap(Map.Entry::getKey,
entry -> entry.getValue() * 2));
```

TREEMAP

A sorted map that stores key-value pairs in natural order or according to a specified comparator.

```
// Create a TreeMap and add key-
value pairs (sorted by key).
TreeMap<String, Double> salaryMap =
new TreeMap<>();
salaryMap.put("Alice", 55000.0);
salaryMap.put("Bob", 60000.0);

// Iterate through the TreeMap keys
(in ascending order).
for (String name : salaryMap.
keySet()) { /*...*/ }

// Iterate through the TreeMap
values (in ascending order of keys).
for (Double salary : salaryMap
.values()) { /*...*/ }

// Use the Stream API to calculate
the total salary.
double totalSalary = salaryMap
.values().stream().mapToDouble(Doubl
e::doubleValue).sum();
```

LINKEDHASHMAP

A map that maintains the order of insertion while still providing hash table efficiency.

```
// Create a LinkedHashMap and add
key-value pairs (maintains insertion
order).
LinkedHashMap<String, Integer>
orderMap = new LinkedHashMap<>();
orderMap.put("First", 1);
orderMap.put("Second", 2);

// Iterate through the LinkedHashMap
key-value pairs (in insertion
order).
for (Map.Entry<String, Integer>
entry : orderMap.entrySet()) {
/*...*/ }

// Use the Stream API to extract
keys in insertion order.
List<String> keysInOrder = orderMap
.keySet().stream().collect(Collector
s.toList());
```

STACK

A data structure that follows the Last-In-First-Out (LIFO) principle.

```
// Use a Stack to push and pop
elements (LIFO).
Stack<String> stack = new Stack<>();
stack.push("Item 1");
stack.push("Item 2");
String topItem = stack.pop();
```

QUEUE

A data structure that follows the First-In-First-Out (FIFO) principle.

```
// Use a LinkedList as a Queue to
offer and poll elements (FIFO).
Queue<String> queue = new
LinkedList<>();
```

```
queue.offer("Task 1");
queue.offer("Task 2");
String nextTask = queue.poll();
```

PRIORITYQUEUE

A priority-based queue, where elements are dequeued based on their priority.

```
// Use a PriorityQueue to offer and
poll elements based on priority.
PriorityQueue<Integer> priorityQueue
= new PriorityQueue<>();
priorityQueue.offer(3);
priorityQueue.offer(1);
int highestPriority = priorityQueue
.poll();
```

CHARACTER ESCAPE SEQUENCES

Character escape sequences are used to represent special characters in Java strings. For example, `\"` is used to include a double quote within a string, and `\n` represents a newline character. The `\uXXXX` escape sequence allows you to specify a Unicode character by its hexadecimal code point. Below are some of the most common Character escape sequences in Java.

Escape Sequence	Description
`\\`	Backslash
`\'`	Single quote (apostrophe)
`\"`	Double quote
`\n`	Newline (line feed)
`\r`	Carriage return
`\t`	Tab
`\b`	Backspace
`\f`	Form feed
`\uXXXX`	Unicode character in hexadecimal format (e.g., `\u0041` for 'A')

OUTPUT FORMATTING WITH PRINTF

Here's a typical example of output formatting using the `printf` method. For a full list of format specifiers and flags visit our comprehensive guide here.

```
String name = "Alice";
int age = 30;
double salary = 55000.75;
boolean married = true;

// Format and print using printf.
The %n specifier is used to insert a
platform-specific newline character
System.out.printf("Name: %s%n",
name); // %s for String
System.out.printf("Age: %d%n", age);
// %d for int
System.out.printf("Salary: %.2f%n",
salary); // %.2f for double with 2
decimal places
System.out.printf("Married: %b%n",
married); // %b for boolean
```

REGULAR EXPRESSIONS

Regular expressions are powerful tools for pattern matching and text manipulation. Below are some typical use cases.

```
// Matching a Simple Pattern
String text = "Hello, World!";
String pattern = "Hello, .*";
boolean matches = text.matches
(pattern);

// Finding All Email Addresses in a
Text
String text = "Contact us at
john@example.com and jane@test.org
for assistance.";
String emailPattern =
"\\b\\S+@\\S+\\.[A-Za-z]+\\b";
Pattern pattern = Pattern.compile
(emailPattern);
Matcher matcher = pattern.matcher
(text);
```

```
while (matcher.find()) {
    String email = matcher.group();
    // Process or store the found
email addresses
}

// Replacing All Occurrences of a
Word
String text = "The quick brown fox
jumps over the lazy dog.";
String replacedText = text
.replaceAll("fox", "cat");

// Splitting a String into Words
String text = "Java is a versatile
programming language.";
String[] words = text.split("\\s+");

// Validating a Date Format
String date = "2023-10-05";
String datePattern = "\\d{4}-\\d{2}-
\\d{2}";
boolean isValid = date.matches
(datePattern);
```

Below are some of the most commonly used regular expression syntax elements in Java.

Syntax	Description
`.`	Matches any character except a newline.
`\t`, `\n`, `\r`, `\f`, `\a`, `\e`	The control characters tab, newline, return, form feed, alert, and escape.
`[]`	Defines a character class, matches any one character from the specified set. You can include characters, ranges or even character classes. For example, `[abc0-9\s&&[^ \t]]` matches 'a', 'b', 'c', a digit or a whitespace character that is not space or tab.
`[^]`	Defines a negated character class, matches any one character NOT in the specified set. You can include characters, ranges or even character classes. For example, `[^0-9]` matches any character that is not a digit.
`*`	Matches the preceding element zero or more times.
`+`	Matches the preceding element one or more times.
`?`	Matches the preceding element zero or one time (optional).
`{n}`	Matches the preceding element exactly `n` times.
`{n,}`	Matches the preceding element at least `n` times.
`{n,m}`	Matches the preceding element between `n` and `m` times (inclusive).
`XY`	Matches any string from X, followed by any string from Y.
`X|Y`	Matches any string from X or Y.
`()`	Groups expressions together, enabling quantifiers to apply to the entire group.
`\`	Escapes a metacharacter, allowing you to match characters like '.', '[', ']', etc., literally.
`^`	Anchors the match at the beginning of the line (or input).
`$`	Anchors the match at the end of the line (or input).

Syntax	Description
`\b`	Matches a word boundary.
`\B`	Matches a non-word boundary.
`\d`	Matches a digit (equivalent to `[0-9]`).
`\D`	Matches a non-digit (equivalent to `[^0-9]`).
`\s`	Matches a whitespace character (e.g., space, tab etc. Equivalent to `[ \t\n\r\f\x0B]`).
`\S`	Matches a non-whitespace character.
`\w`	Matches a word character (alphanumeric or underscore, equivalent to `[a-zA-Z0-9_]`).
`\W`	Matches a non-word character.

MATCHING FLAGS

Pattern matching can be adjusted with flags. You can embed them at the beginning of the pattern of use them as shown below:

```
Pattern pattern = Pattern.compile
(patternString, Pattern
.CASE_INSENSITIVE + Pattern
.UNICODE_CASE)
```

Flag	Description
`Pattern.CASE_INSENSITIVE` (`(?i)`)	Enables case-insensitive matching.
`Pattern.MULTILINE` (`(?m)`)	Enables multiline mode, where `^` and `$` match the start/end of each line.
`Pattern.DOTALL` (`(?s)`)	Enables dotall mode, allowing `.` to match any character, including newlines.
`Pattern.UNICODE_CASE` (`(?u)`)	Enables Unicode case-folding.
`Pattern.COMMENTS` (`(?x)`)	Allows whitespace and comments in the pattern for readability.
`Pattern.UNIX_LINES` (`(?d)`)	Only '\n' is recognized as a line terminator when matching ^ and $ in multiline mode.
`Pattern.CANON_EQ` (`(?c)`)	Matches canonical equivalence, treating composed and decomposed forms as equivalent.
`Pattern.LITERAL`	Treats the entire pattern as a literal string, disabling metacharacter interpretation.

LOGGING

In Java's core logging implementation (`java.util.logging`), the logging configuration is specified through a properties file (`jre/lib/logging.properties`). You can specify another file by using the system property`java.util.logging.config.file`when running your application. Below are some common properties used in the `logging.properties` file, along with their descriptions:

Property	Description
`.level`	Sets the default logging level for all loggers.
`handlers`	Specifies which handlers (output destinations) to use for log records.
`<logger>.level`	Defines the log level for a specific logger.
`java.util.logging.ConsoleHandler.level`	Sets the log level for the ConsoleHandler.
`java.util.logging.ConsoleHandler.formatter`	Specifies the formatter for log messages sent to the console.

Property	Description
`java.util.logging.FileHandler.level`	Sets the log level for the FileHandler (logging to files).
`java.util.logging.FileHandler.pattern`	Defines the pattern for log file names.
`java.util.logging.FileHandler.limit`	Specifies the maximum size (in bytes) for each log file before rolling over.
`java.util.logging.FileHandler.count`	Sets the maximum number of log files to keep.
`java.util.logging.FileHandler.formatter`	Specifies the formatter for log messages written to files.
`java.util.logging.SimpleFormatter.format`	Allows customization of log message format for SimpleFormatter.
`java.util.logging.ConsoleHandler.filter`	Specifies a filter for log records to be sent to the console.
`java.util.logging.FileHandler.filter`	Specifies a filter for log records to be written to files.
`java.util.logging.ConsoleHandler.encoding`	Specifies the character encoding for console output.
`java.util.logging.FileHandler.encoding`	Specifies the character encoding for file output.
`java.util.logging.ConsoleHandler.errorManager`	Defines the error manager for handling errors in the ConsoleHandler.
`java.util.logging.FileHandler.errorManager`	Defines the error manager for handling errors in the FileHandler.

An example logging configuration file is provided below.

```
# Set the default logging level for all loggers
.level = INFO

# Handlers specify where log records are output
handlers = java.util.logging.ConsoleHandler

# Specify the log level for a specific logger
com.example.myapp.MyClass.level = FINE

# ConsoleHandler properties
java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

# FileHandler properties
java.util.logging.FileHandler.level = ALL
java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
```

PROPERTY FILES

A properties file is a simple text file used to store configuration settings or key-value pairs. It is commonly used for configuring applications, as it provides a human-readable and editable format. Properties files are commonly used for application configuration, internationalization, and other scenarios where key-value pairs need to be stored in a structured and easily editable format.

- Properties files typically use the `.properties` file extension.
- Each line in the properties file represents a key-value pair. The key and value are separated by an equal sign (`=`) or a colon (`:`).
- You can include comments in properties files by prefixing a line with a hash (`#`) or an exclamation mark (`!`). Comments are ignored

when reading properties.

- Whitespace characters (spaces and tabs) before and after the key and value are ignored.
- You can escape special characters like spaces, colons, and equals signs using a backslash (\).

Below is an example of loading and storing data in a properties file programmatically.

```
Properties properties = new
Properties();
try {
  FileInputStream fileInputStream =
new FileInputStream
("config.properties");
  properties.load(fileInputStream);
  fileInputStream.close();

  // Access properties
  String value = properties
.getProperty("key");
  System.out.println("Value for key:
" + value);

  // Update properties
  properties.setProperty("key1",
"value1");
  properties.setProperty("key2",
"value2");
  FileOutputStream fileOutputStream
= new FileOutputStream
("config.properties");
  properties.store(fileOutputStream,
"My Configuration");
  fileOutputStream.close();

} catch (IOException e) {
  e.printStackTrace();
}
```

JAVAC COMMAND

`javac` is the Java Compiler, a command-line tool included in the Java Development Kit (JDK). Its main purpose is to compile Java source code files (files with a `.java` extension) into bytecode files (files with a `.class` extension) that can be executed by the Java Virtual Machine (JVM).

Below are some of the most commonly used options with the `javac` command. They allow you to customize the compilation process, specify source and target versions, control error handling, and manage classpaths and source file locations. You can use these options to tailor the compilation of your Java code to your project's specific requirements.

Option	Description
`-classpath` or `-cp`	Specifies the location of user-defined classes and packages.
`-d <directory>`	Specifies the destination directory for compiled class files.
`-source <release>`	Specifies the version of the source code (e.g., "1.8" for Java 8).
`-target <version>`	Generates class files compatible with the specified target version (e.g., "1.8").
`-encoding <charset>`	Sets the character encoding for source files.
`-verbose`	Enables verbose output during compilation.
`-deprecation`	Shows a warning message for the use of deprecated classes and methods.
`-nowarn`	Disables all warnings during compilation.
`-Xlint[:<key>]`	Enables or disables specific lint warnings.
`-Xlint:-<key>`	Disables specific lint warnings.
`-g`	Generates debugging information for source-level debugging.
`-g:none`	Disables generation of debugging information.
`-Werror`	Treats warnings as errors.

Option	Description
`-sourcepath`	Specifies locations to search for source files.
`-Xmaxerrs <number>`	Sets the maximum number of errors to print.
`-Xmaxwarns <number>`	Sets the maximum number of warnings to print.

JAR COMMAND

Jar (Java Archive) files are a common way to package and distribute Java applications, libraries, and resources.

- Jar files are archive files that use the ZIP format, making it easy to compress and bundle multiple files and directories into a single file.
- Jar files can contain both executable (class files with a `main` method) and non-executable (resources, libraries) components.
- A special file called `META-INF/MANIFEST.MF` can be included in a jar file. It contains metadata about the jar file, such as its main class and version information.
- Java provides command-line tools like `jar` and `jarsigner` to create, extract, and sign jar files.

Below is a table with some common `jar` command options and what they do:

Option	Description
`c`	Create a new JAR file.
`x`	Extract files from an existing JAR file.
`t`	List the contents of a JAR file.
`u`	Update an existing JAR file with new files.
`f <jarfile>`	Specifies the JAR file to be created or operated on.
`v`	Verbose mode. Display detailed output.
`C <dir>`	Change to the specified directory before performing the operation.
`M`	Do not create a manifest file for the entries.
`m <manifest>`	Include the specified manifest file.
`e`	Sets the application entry point (main class) for the JAR file's manifest.
`i <index>`	Generates or updates the index information for the specified JAR file.
`0`	Store without using ZIP compression.
`J<option>`	Passes options to the underlying Java VM when running the `jar` command.

JAVA COMMAND

The `java` command is used to run Java applications and execute bytecode files (compiled Java programs) on the Java Virtual Machine (JVM). It serves as the primary entry point for launching and executing Java applications.

Below are some of the common options used with the `java` command for controlling the behavior of the Java Virtual Machine (JVM) and Java applications. Depending on your specific use case, you may need to use some of these options to configure the JVM or your Java application.

Option	Description
`-classpath` or `-cp`	Specifies the classpath for finding user-defined classes and libraries.
`-version`	Displays the Java version information.
`-help`	Shows a summary of available command-line options.

Option	Description
`-Xmx<size>`	Sets the maximum heap size for the JVM (e.g., `-Xmx512m` sets it to 512 MB).
`-Xms<size>`	Sets the initial heap size for the JVM (e.g., `-Xms256m` sets it to 256 MB).
`-Xss<size>`	Sets the stack size for each thread (e.g., `-Xss1m` sets it to 1 MB).
`-D<property>=<value>`	Sets a system property to a specific value.
`-ea` or `-enableassertions`	Enables assertions (disabled by default).
`-da` or `-disableassertions`	Disables assertions.
`-XX:+<option>`	Enables a specific non-standard JVM option.
`-XX:-<option>`	Disables a specific non-standard JVM option.
`-XX:<option>=<value>`	Sets a specific non-standard JVM option to a value.
`-verbose:class`	Displays information about class loading.
`-verbose:gc`	Displays information about garbage collection.
`-Xrunhprof:format`	Generates heap and CPU profiling data.
`-Xdebug`	Enables debugging support.
`-Xrunjdwp:<options>`	Enables Java Debug Wire Protocol (JDWP) debugging.

POPULAR 3RD PARTY JAVA

Java has an enormous ecosystem of third-party libraries and frameworks. Below are just a few available that will greatly improve the development experience. Each library serves a specific purpose and can significantly simplify various aspects of your code, from handling data to simplifying testing and improving code quality. Depending on your project's requirements, you may find these libraries and others to be valuable additions to your Java toolkit.

Library Name	Description
Apache Commons Lang	Provides a set of utility classes for common programming tasks, such as string manipulation, object handling, and more.
Jackson (JSON Processor)	A widely used library for working with JSON data, allowing for easy serialization and deserialization between Java objects and JSON.
Log4j	A flexible and highly configurable logging framework that provides various logging options for Java applications.
Spring Framework	A comprehensive framework for building Java applications, offering modules for dependency injection, data access, web applications, and more.
Hibernate	An Object-Relational Mapping (ORM) framework that simplifies database interaction by mapping Java objects to database tables.
Apache HttpClient	A library for making HTTP requests and interacting with web services, supporting various authentication methods and request customization.

Library Name	Description
JUnit	A popular testing framework for writing and running unit tests in Java, ensuring the reliability and correctness of code.
Mockito	A mocking framework for creating mock objects during unit testing, allowing you to isolate and test specific parts of your code.
Apache Commons IO	Provides a set of utility classes for input/output operations, such as file handling, stream management, and more.
Lombok	A library that simplifies Java code by generating boilerplate code for common tasks like getter and setter methods, constructors, and logging.
JAX-RS (Java API for RESTful Web Services)	A standard API for building RESTful web services in Java, often used with frameworks like Jersey or RESTEasy.