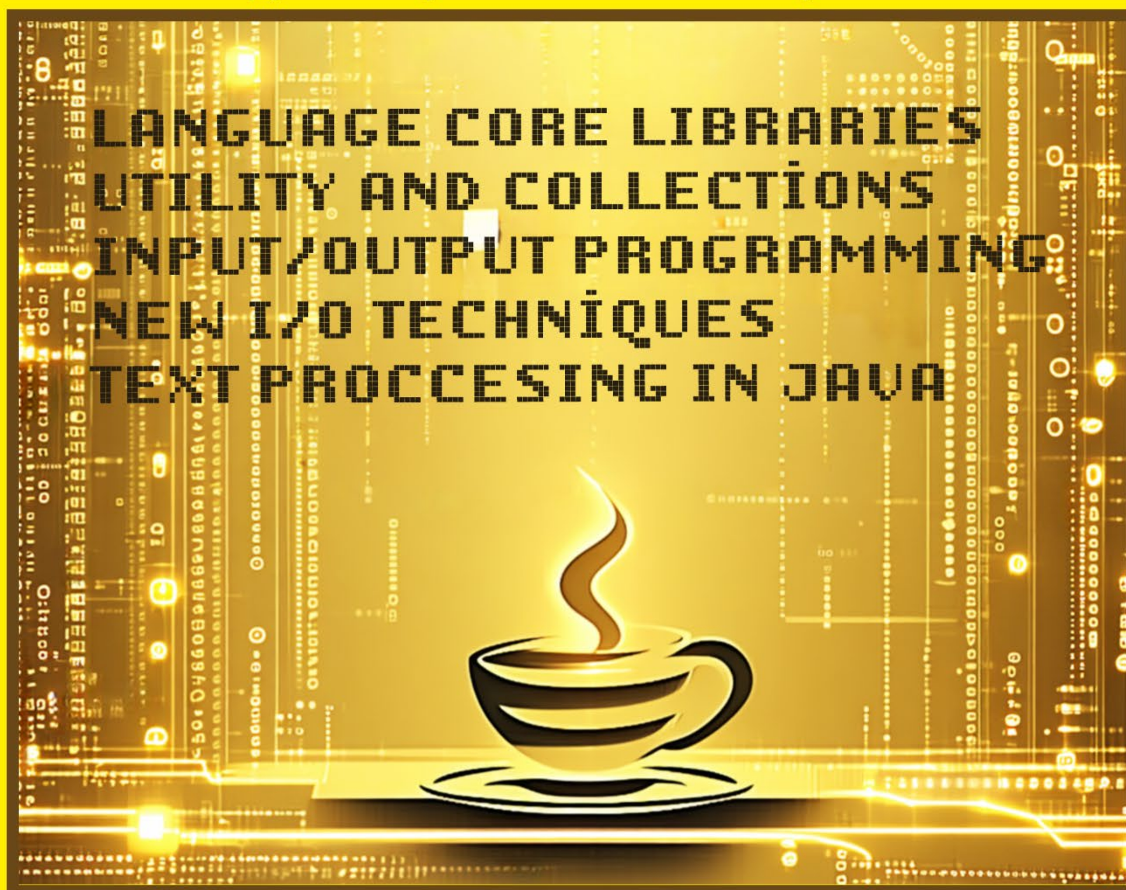
EXPLORING JAVA LIBRARIES

A DEVELOPER'S GUIDE

I

Supported by More Than 300 Projects



SOLİS DYNAMİCS

Preface

Dear Software Developers and Architects of the Future,

It is with great excitement and pride that I share with you the book titled "*Exploring Java Libraries: A Developer's Guide.*" This work is designed for developers who wish to explore and effectively utilize the system libraries of Java, one of the most powerful and widely used programming languages in the software world. My goal is to help you gain a broad perspective by mastering Java libraries and to reinforce this knowledge through practical projects.

The book is structured around Java's core packages. These packages are the fundamental building blocks that harness the power and flexibility of Java. Each one enables expertise in different areas crucial to the software development process. With Java's modular architecture, these packages interact seamlessly to provide an efficient, organized, and sustainable development environment. Understanding and effectively using these packages according to user needs is critical for successful software projects.

I compiled this book from my own study notes, and my purpose in sharing it is to gain experience and pass that experience on to you. While developing the projects, I utilized the AI assistant Sider.ai. This serves as an example of how artificial intelligence can be integrated into modern software development processes. Additionally, I left certain solutions in the projects for developers to resolve. This allows readers to conduct their own research and improve themselves further.

The structure of the book delves into Java packages and their categories, such as interfaces, classes, and exceptions, in great detail. As a learning method, alongside the definition of each class, the definitions of its methods, sample codes, and their outputs are provided. By developing the projects in the book using different approaches, I aim to encourage your mastery of Java libraries. Therefore, I recommend typing out the codes in the book rather than relying on copy-paste, as I believe this method of learning is insufficient.

This book is the first in the series and was written using Java 22 on the Windows platform with the Eclipse IDE. I encourage my readers to provide feedback; your insights will contribute to the book's development and serve as an invaluable guide for me.

I hope this journey will take you to new horizons in the world of Java libraries and inspire you in your software development endeavors. Remember, libraries are not just code—they are also worlds of thought and creativity. This book is a starting point to unlock that potential.

I hope this book proves useful in your journey to explore Java libraries and opens new vistas in software development. Wishing you an enjoyable and fruitful reading experience!

Sincerely,
Solis Dynamics

solisdynamicscontact@gmail.com

Table of Contents

1. Lang Package ... 1
A. Classes ... 3
- Core Structural Classes ... 3
- Numeric Wrapper Classes ... 9
- String and Text Processing ... 15
- System and Process Management ... 26
- Reflection and Class Management ... 37
- Mathematical Classes ... 41
- Security and System Control Classes ... 48
- Auxiliary Classes ... 49

B Interfaces ... 52
- Core Functional Interfaces ... 52
- Data Processing Interfaces ... 56
- Special Purpose Interfaces ... 59

C. Enum Classes ... 61
- Most Critical Enum Classes ... 61

D. Exceptions ... 66
- Fundamental and Most Critical Exceptions ... 66
- Frequently Encountered Critical Exceptions ... 70
- System and Security Exceptions ... 77
- Special Condition Exceptions ... 82
- Index and Dimension Exceptions ... 84
- Type and Conversion Exceptions ... 86
- Other Specific Exceptions ... 88

E. Errors ... 91
- Fundamental and Most Critical Erors ... 91
- System and Loading Erors ... 101
- Runtime Erros ... 108
- Acces and Initialization Errors ... 112
- Special Condition Errors ... 115

F. Annotation Interfaces ... 118
- Most Critical Annotations ... 118

2. Util Package ... 127
A. Interfaces ... 131
- Collection Hierarchy Interfaces ... 131
- Sorting and Navigation Interfaces ... 145
- Iteration and Navigation Interfaces ... 157
- Primitive Iteration Interfaces ... 167

◆ Map Related Interfaces 171
◆ Specialized Purpose Interfaces 180
B. Classes 185
◆ Basic Collection Classes 185
◆ Collection Infrastructure Classes 211
◆ Utility Classes 224
◆ Resource and Localization Classes 240
◆ Date and Time Classes 246
◆ Formatting and Conversion Classes 257
◆ Event and Listener Classes 264
C. Enum Classes 271
◆ Localization-Related Enum Classes 271
◆ Formatting Operations Related Enum Classes 277
D. Errors 278
◆ Critical Collection and Structural Exceptions 278
◆ Formatting and Conversion Exceptions 281
◆ Localization and Resource Exceptions 285
◆ Input and System Exceptions 288
3. IO Package **291**
A. Interfaces 293
◆ Data Stream Interfaces 293
◆ Serialization Interfaces 300
◆ Filtering Interfaces 305
◆ Resource Management Interfaces 307
B. Classes 309
◆ Basic Byte Stream 309
◆ File Byte Stream 316
◆ Filter Byte Stream 319
◆ Array Byte Stream 323
◆ Piped Byte Stream 327
◆ Basic Character Stream 330
◆ File Character Stream 341
◆ Filter Character Stream 346
◆ Array Character Stream 350
◆ Conversion Stream Classes 354
◆ Object Serialization Classes 360
◆ Utility Classes 365
◆ String-Based Stream Classes 382
C. Enum Classes 386

D. Exception Classes ... 388
◆ Basic I/O Exceptions ... 388
◆ File and Access Exceptions ... 392
◆ Serialization Exceptions ... 395
◆ Data Format Exceptions ... 403
E. Annotation Interfaces ... 407
4. NIO Package ... 410
A. Classes ... 414
-NIO.File Package ... 439
A. Interfaces ... 439
◆ File System Core Interfaces ... 439
◆ File Operation Interfaces ... 443
◆ Event Handling Interfaces ... 451
◆ Watch Service Interfaces ... 455
B. Classes ... 460
◆ File System Classes ... 460
◆ File Operation Classes ... 468
◆ File Traversal Classes ... 478
◆ Watching Events Classes ... 480
C. Enum Classes ... 482
◆ Primary File Operation Enums ... 482
◆ Acces and Event Enums ... 489
D. Exception Classes ... 492
◆ File System Operation Exceptions ... 492
◆ Path and Directory Exceptions ... 503
◆ Closed Resource Exceptions ... 509
◆ Provider and Specific Exceptions ... 512
-NIO.File.SPI Package ... 519
A. Class ... 519
-NIO.Channels Package ... 524
A. Interfaces ... 524
◆ Main Classes ... 524
◆ Asynchronous Communication Interfaces ... 533
◆ Completing Action Interfaces ... 535
B. Class ... 537
◆ Main Channel Classes ... 537
◆ Asynchronous Channel Classes ... 551
◆ Pipe Classes ... 559
◆ Selection and Key Classes ... 565

◆ Other Classes ... 572

C. Exception Classes ... 578

-NIO.Charset Package ... **594**

A. Classes ... 594

B. Exception Classes ... 611

-NIO.Charset.SPI Package ... **614**

-NIO.File.Attribute Package ... **616**

A. Interfaceses ... 616

B. Classes ... 635

C. Enum Classes ... 644

D. Exception Classes ... 649

5. Text Package ... **651**

A. Interfaceses ... 654

B. Classes ... 660

◆ Annotation Classes ... 660

◆ Format Classes ... 664

◆ Collection/Iteration Classes ... 679

◆ Symbol and Format Helper Classes ... 685

◆ Comparison and Sorting Classes ... 693

◆ Unicode and Text Processing Classes ... 700

C. Enum Classes ... 707

D. Exception Classes ... 712

1-java.lang

Java is a package that contains the fundamental classes of the Java programming language and hosts the most commonly used classes in Java applications. This package forms the core building blocks of Java and is automatically available in every Java program. The classes that constitute Java's core features cover basic data types, object processing methods, mathematical calculations, and tools for interacting with the system. The java.lang package offers various classes and interfaces related to the language features and fundamentals of Java. This package is automatically included in every Java program, so there is no need to specify the package name before using class names.

GENERAL USAGE

⇒ **Error Management**: Exception classes are used to manage errors that occur in the application. Errors can be caught and processed using try-catch blocks.

⇒ **Working with Strings**: The String and StringBuilder classes are commonly used for operations on text data.

⇒ **Mathematical Calculations**: The Math class is used for various mathematical formulas and calculations.

⇒ **Multithreading**: The Thread class is used to allow multiple tasks to be executed simultaneously in your application.

EXAMPLE USAGE

```
package a1_java.lang;
import java.lang.reflect.Method;
public class a1_javaLangExample {
    public static void main(String[] args) {
        // Example of Object class
        System.out.println("============Object Class Example============");
        MyObject obj1 = new MyObject("Sample Object 1");
        MyObject obj2 = new MyObject("Sample Object 1");
        // Checking equality of objects using equals method
        System.out.println("Are obj1 and obj2 equal? " + obj1.equals(obj2)); // true
        System.out.println("Hash code of obj1: " + obj1.hashCode());
        System.out.println("Hash code of obj2: " + obj2.hashCode());
        System.out.println("String representation of obj1: " + obj1.toString());
        // Example of Class class
        System.out.println("============Class Example============");
        try {
            // Represents a "wildcard" type indicating that the type of the class is unspecified.
              Class<?> clazz = Class.forName("a1_java.lang.MyObject");  // Called with package name
            System.out.println("Class name: " + clazz.getName());
            System.out.println("Superclass: " + clazz.getSuperclass().getName());
            System.out.println("Declared methods: ");
            for (Method method : clazz.getDeclaredMethods()) {
                // var is a feature used to automatically determine the type of declared variables.
                System.out.println(" - " + method.getName());    }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();  }
        // Example of System class
        System.out.println("============System Class Example============");
        long currentTime = System.currentTimeMillis();
        System.out.println("Current time (milliseconds): " + currentTime);
        System.out.println("Java version: " + System.getProperty("java.version"));
        // Example of String class
        System.out.println("============String Class Example============");
        String str = "Hello World";
        System.out.println("String name: " + str);
        System.out.println("String length: " + str.length());
        System.out.println("Substring: " + str.substring(0, 7));
        System.out.println("4th character of the string: " + str.charAt(3));
        System.out.println("Uppercase: " + str.toUpperCase());
        System.out.println("Lowercase: " + str.toLowerCase());
        // Example of Math class
        System.out.println("============Math Class Example============");
        double number = -25;
        System.out.println("Number: " + number);
        System.out.println("Absolute value: " + Math.abs(number));
        System.out.println("Square root: " + Math.sqrt(25));
        System.out.println("2 to the power of 3: " + Math.pow(2, 3));
        System.out.println("Random number: " + Math.random());
        // Example of Exception class
        System.out.println("============Exception Class Example============");
        try {
            int[] arr = new int[5];
```

```
            System.out.println(arr[10]); // This line will throw ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: " + e.getMessage());
            e.printStackTrace();}  // Print the stack trace of the error
        // Example of Thread class
        System.out.println("=============Thread Class Example=============");
        Thread thread = new Thread() { // Thread is a class used to create multiple threads in Java.
            public void run() {
                for (int i = 0; i < 5; i++) {
                    System.out.println("Thread is running: " + i);
                    try {
                        Thread.sleep(500); // Wait for 0.5 seconds
                    } catch (InterruptedException e) {
                        e.printStackTrace(); } } } };
        thread.start(); // Start the thread
        // Wait for the main thread to complete
        try {
            thread.join();  // join() provides synchronization between threads
        } catch (InterruptedException e) {
            e.printStackTrace(); }
        System.out.println("Main thread completed."); } }
// A class that inherits from Object class
class MyObject {
    private String name;
    public MyObject(String name) {
        this.name = name; }
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true; // Reference equality
        if (!(obj instanceof MyObject)) return false; // Type check
        MyObject other = (MyObject) obj; // Type casting
        return name != null ? name.equals(other.name) : other.name == null; }
        // Check for equality of names
    @Override
    public int hashCode() {
        return name != null ? name.hashCode() : 0;}  // Return hash code
    @Override
    public String toString() {
        // String representation of the object
        return "MyObject{name='" + name + "'}"; } }
```

OUTPUT

```
=============Object Class Example=============
Are obj1 and obj2 equal? true
Hash code of obj1: 1710299462
Hash code of obj2: 1710299462
String representation of obj1: MyObject{name='Sample Object 1'}
=============Class Example=============
Class name: a1_java.lang.MyObject
Superclass: java.lang.Object
Declared methods:
 - equals
 - toString
 - hashCode
=============System Class Example=============
Current time (milliseconds): 1740526904415
Java version: 22.0.2
=============String Class Example=============
String name: Hello World
String length: 11
Substring: Hello W
4th character of the string: l
Uppercase: HELLO WORLD
Lowercase: hello world
=============Math Class Example=============
Number: -25.0
Absolute value: 25.0
Square root: 5.0
2 to the power of 3: 8.0
Random number: 0.07434692122063724
=============Exception Class Example=============
Error: Index 10 out of bounds for length 5
java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 5
	at a1_java.lang.a1_javaLangExample.main(a1_javaLangExample.java:53)
```

```
=============Thread Class Example=============
Thread is running: 0
Thread is running: 1
Thread is running: 2
Thread is running: 3
Thread is running: 4
Main thread completed.
```

A. Classes
◈ Core Structural Classe
1-Object

The Object class is the superclass of all Java classes. Every class is derived from the Object class. Each Java class is either directly or indirectly derived from the Object class. This class provides the fundamental functionality for objects and contains many important methods.

1) **equals(Object obj):** Checks whether two objects are equal. By default, this method compares object references. If a class overrides this method, it can be used to compare the contents of the objects.
2) **hashCode():** Returns the hash code of the object. It is important when used in conjunction with the equals() method, as equal objects must have the same hash code. By default, this method returns a hash code based on the object's memory address.
3) **toString():** Returns the string representation of the object. By default, it returns the class name and hash code of the object. This method can be overridden to provide a more meaningful representation of the object.
4) **getClass():** Returns a Class object that represents the class of the object. This method can be used to determine the dynamic type of the object.
5) **clone():** Returns a copy of the object. This method must be used with the Cloneable interface. By default, it creates a shallow copy of the object. If a deep copy is desired, this method should be overridden.
6) **finalize():** Called during the garbage collection process. This method can be used to perform cleanup operations before the object is destroyed. However, its use is discouraged and it has been deprecated since Java 9.
7) **notify():** Wakes up a waiting thread with wait(). Synchronization and multi-threading threads are used.
8) **notifyAll():** Wakes up all threads that are waiting on the object. This method is used for synchronization operations.
9) **wait():** Suspends a thread that is waiting on the object. This method is used for synchronization operations and allows the thread to wait for a specified period.
10) **wait(long timeout):** Suspends a thread that is waiting on the object and wakes it up after the specified time.
11) **wait(long timeout, int nanos):** Suspends a thread that is waiting on the object and wakes it up after the specified time and nanoseconds.

EXAMPLE USAGE 1

```
package a1_java.lang;
public class a1_ObjectExample implements Cloneable {
    private String name;
    public a1_ObjectExample(String name) {
        this.name = name;  }
    // 1) equals(Object obj): Checks object equality
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof a1_ObjectExample)) return false;
        a1_ObjectExample other = (a1_ObjectExample) obj;
        return name != null ? name.equals(other.name) : other.name == null; }
    // 2) hashCode(): Returns object's hash code
    @Override
    public int hashCode() {
        return name != null ? name.hashCode() : 0; }
    // 3) toString(): Returns string representation of object
    @Override
    public String toString() {
        return "Object{name='" + name + "'}"; }
    // 5) clone(): Creates a copy of the object
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone(); }
    // 6) finalize(): Called by garbage collector (deprecated since Java 9)
   @SuppressWarnings("removal")
 @Override
```

```java
    protected void finalize() throws Throwable {
        System.out.println("Finalize called: " + name);
        super.finalize(); }
    // 7) notify(): Wakes up a waiting thread
    public synchronized void notifyExample() {
        notify();
        System.out.println("Notify called."); }
    // 8) notifyAll(): Wakes up all waiting threads
    public synchronized void notifyAllExample() {
        notifyAll();
        System.out.println("NotifyAll called."); }
    // 9) wait(): Suspends the thread
    public synchronized void waitExample() throws InterruptedException {
        wait();
        System.out.println("Thread continued."); }
    // 10) wait(long timeout): Suspends for specified time
    public synchronized void waitWithTimeout() throws InterruptedException {
        wait(1000);
        System.out.println("Continued after timeout."); }
    // 11) wait(long timeout, int nanos): Suspends with milliseconds and nanoseconds
    public synchronized void waitWithTimeoutNanos() throws InterruptedException {
        wait(1000, 500000);
        System.out.println("Continued after nanosecond wait."); }
    public static void main(String[] args) {
        // 1) equals() method example
        a1_ObjectExample obj1 = new a1_ObjectExample("Alice");
        a1_ObjectExample obj2 = new a1_ObjectExample("Alice");
        a1_ObjectExample obj3 = new a1_ObjectExample("Bob");
        System.out.println("Are obj1 and obj2 equal? " + obj1.equals(obj2));
        System.out.println("Are obj1 and obj3 equal? " + obj1.equals(obj3));
        // 2) hashCode() method example
        System.out.println("obj1 Hash Code: " + obj1.hashCode());
        System.out.println("obj2 Hash Code: " + obj2.hashCode());
        // 3) toString() method example
        System.out.println("obj1 String Representation: " + obj1);
        // 4) getClass() method example (cannot be overridden)
        System.out.println("Class Name: " + obj1.getClass().getName());
        // 5) clone() method example
        try {
            a1_ObjectExample clonedObj = (a1_ObjectExample) obj1.clone();
            System.out.println("Cloned Object: " + clonedObj);
        } catch (CloneNotSupportedException e) {
            e.printStackTrace(); }
        // 6) finalize() method example
        obj1 = null;
        System.gc(); } }
```

OUTPUT

```
Are obj1 and obj2 equal? true
Are obj1 and obj3 equal? false
obj1 Hash Code: 63350368
obj2 Hash Code: 63350368
obj1 String Representation: Object{name='Alice'}
Class Name: a1_java.lang.a1_ObjectExample
Cloned Object: Object{name='Alice'}
Finalize called: Alice
```

2-Class

The Class class is a reflection class used to obtain information about classes and interfaces at runtime. It provides a comprehensive mechanism for introspecting and manipulating class metadata dynamically during program execution.

Purpose

- Retrieve runtime information about classes and interfaces
- Provide detailed metadata about class structures
- Enable dynamic class manipulation
- Support reflection-based programming techniques

Methods

A. Class Information Retrieval Methods:

1. **getName():** Returns the fully qualified class name as a String.
2. **getSimpleName():** Returns the simple name of the class without package.

3. **getCanonicalName():** Returns the canonical name of the class.
4. **getTypeName():** Returns a type name for the class.
5. **toString():** Returns a string representation of the class.

B. Object Creation Methods:

6. **newInstance():** Creates a new instance of the class.
7. **forName(String className):** Loads a class dynamically by its name.
8. **getConstructor(Class<?>... parameterTypes):** Returns a specific public constructor.
9. **getDeclaredConstructor(Class<?>... parameterTypes):** Returns a constructor regardless of access modifier.

C. Member Information Retrieval Methods:

10. **getMethods():** Returns all public methods of the class.
11. **getDeclaredMethods():** Returns all declared methods, including private.
12. **getFields():** Returns all public fields.
13. **getDeclaredFields():** Returns all fields, including private.
14. **getConstructors():** Returns all public constructors.
15. **getDeclaredConstructors():** Returns all constructors, including private.

D. Class Characteristic Check Methods:

16. **isInterface():** Checks if the class is an interface.
17. **isEnum():** Checks if the class is an enum.
18. **isArray():** Checks if the class represents an array.
19. **isPrimitive():** Checks if the class is a primitive type.
20. **isAnnotation():** Checks if the class is an annotation type.
21. **isAnonymousClass():** Checks if the class is an anonymous class.
22. **isMemberClass():** Checks if the class is a member class.
23. **isLocalClass():** Checks if the class is a local class.

E. Inheritance and Package Information Methods:

24. **getSuperclass():** Returns the superclass of the current class.
25. **getPackage():** Returns the package of the class.
26. **getInterfaces():** Returns implemented interfaces.
27. **getGenericInterfaces():** Returns generic interfaces.

F. Access and Security Methods:

28. **getModifiers():** Returns the Java language modifiers.
29. **getProtectionDomain():** Returns the protection domain of the class.
30. **getClassLoader():** Returns the class loader for the class.

EXAMPLE USAGE 2

```java
package a1_java.lang;
import java.lang.reflect.*;
import java.security.ProtectionDomain;
import java.io.Serializable;
public class a2_ClassExample {
    // Custom class for demonstration
    public static class ExampleClass implements Serializable {
        private static final long serialVersionUID = 1L;
        private int privateField;
        public String publicField;
        // Parameterless Constructor
        public ExampleClass() {
            this.privateField = 0; }
        // Public constructor
        public ExampleClass(int value) {
            this.privateField = value; }
        // Public method
        public void publicMethod() {
            System.out.println("Public method called"); }
        // Getter for privateField to use the field
        public int getPrivateField() {
            return privateField; } }
    public static void main(String[] args) {
        try {
            // Method A: Class Information Retrieval Methods
            Class<?> exampleClass = ExampleClass.class;
            // A1. getName(): Fully qualified class name
            System.out.println("1. Full Class Name: " + exampleClass.getName());
```

```
            // A2. getSimpleName(): Simple class name without package
            System.out.println("2. Simple Class Name: " + exampleClass.getSimpleName());
            // A3. getCanonicalName(): Canonical class name
            System.out.println("3. Canonical Name: " + exampleClass.getCanonicalName());
            // A4. getTypeName(): Type name of the class
            System.out.println("4. Type Name: " + exampleClass.getTypeName());
            // A5. toString(): String representation of the class
            System.out.println("5. Class toString(): " + exampleClass.toString());
            // Method B: Object Creation Methods
            // B6. Create new instance and demonstrate its use
            ExampleClass instance = (ExampleClass) exampleClass
                .getConstructor(int.class)
                .newInstance(42);
            System.out.println("B6. Created Instance Private Field Value: " +
instance.getPrivateField());
            // B7. forName(): Dynamic class loading and demonstrate its use
            Class<?> dynamicClass = Class.forName("a1_java.lang.a2_ClassExample$ExampleClass");
            System.out.println("B7. Dynamic Class Name: " + dynamicClass.getSimpleName());
            // B8. getConstructor(): Get public constructor and demonstrate its use
            Constructor<?> publicConstructor = exampleClass.getConstructor(int.class);
            System.out.println("B8. Public Constructor: " + publicConstructor);
            // B9. getDeclaredConstructor(): Get any constructor
            Constructor<?> privateConstructor = exampleClass.getDeclaredConstructor();
            privateConstructor.setAccessible(true);
            Object privateInstance = privateConstructor.newInstance();
            System.out.println("B9. Private Constructor Instance: " + privateInstance);
            // Method C: Member Information Retrieval Methods
            // C10. getMethods(): Get public methods
            System.out.println("\nPublic Methods:");
            for (Method method : exampleClass.getMethods()) {
                System.out.println("C10. Public Method: " + method.getName()); }
            // C11. getDeclaredMethods(): Get all methods
            System.out.println("\nAll Declared Methods:");
            for (Method method : exampleClass.getDeclaredMethods()) {
                System.out.println("C11. Declared Method: " + method.getName()); }
            // Method D: Class Characteristic Check Methods
            // D16. isInterface()
            System.out.println("D16. Is Interface: " + exampleClass.isInterface());
            // D17. isEnum()
            System.out.println("D17. Is Enum: " + exampleClass.isEnum());
            // D18. isArray()
            System.out.println("D18. Is Array: " + exampleClass.isArray());
            // Method E: Inheritance and Package Information Methods
            // E24. getSuperclass()
            System.out.println("E24. Superclass: " + exampleClass.getSuperclass());
            // E25. getPackage()
            System.out.println("E25. Package: " + exampleClass.getPackage());
            // Method F: Access and Security Methods
            // F28. getModifiers()
            int modifiers = exampleClass.getModifiers();
            System.out.println("F28. Modifiers: " + Modifier.toString(modifiers));
            // F29. getProtectionDomain()
            ProtectionDomain protectionDomain = exampleClass.getProtectionDomain();
            System.out.println("F29. Protection Domain: " + protectionDomain);
            // F30. getClassLoader()
            ClassLoader classLoader = exampleClass.getClassLoader();
            System.out.println("F30. ClassLoader: " + classLoader);
        } catch (Exception e) {
            e.printStackTrace(); } } }
```

OUTPUT

```
1. Full Class Name: a1_java.lang.a2_ClassExample$ExampleClass
2. Simple Class Name: ExampleClass
3. Canonical Name: a1_java.lang.a2_ClassExample.ExampleClass
4. Type Name: a1_java.lang.a2_ClassExample$ExampleClass
5. Class toString(): class a1_java.lang.a2_ClassExample$ExampleClass
B6. Created Instance Private Field Value: 42
B7. Dynamic Class Name: ExampleClass
B8. Public Constructor: public a1_java.lang.a2_ClassExample$ExampleClass(int)
B9. Private Constructor Instance: a1_java.lang.a2_ClassExample$ExampleClass@1b28cdfa

Public Methods:
C10. Public Method: publicMethod
C10. Public Method: getPrivateField
```

```
C10. Public Method: equals
C10. Public Method: toString
C10. Public Method: hashCode
C10. Public Method: getClass
C10. Public Method: notify
C10. Public Method: notifyAll
C10. Public Method: wait
C10. Public Method: wait
C10. Public Method: wait

All Declared Methods:
C11. Declared Method: publicMethod
C11. Declared Method: getPrivateField
D16. Is Interface: false
D17. Is Enum: false
D18. Is Array: false
E24. Superclass: class java.lang.Object
E25. Package: package a1_java.lang
F28. Modifiers: public static
F29. Protection Domain: ProtectionDomain  (file:/C:/Users/Pc/OneDrive/Masa%c3%bcst%c3%bc/Eclipse-Workspace/JavaSystemLibraries/target/classes/ <no signer certificates>)
 jdk.internal.loader.ClassLoaders$AppClassLoader@55f96302
 <no principals>
 java.security.Permissions@3feba861 (
 ("java.lang.RuntimePermission" "exitVM")
 ("java.io.FilePermission" "C:\Users\Pc\OneDrive\Masaüstü\Eclipse-Workspace\JavaSystemLibraries\target\classes\-" "read")
)


F30. ClassLoader: jdk.internal.loader.ClassLoaders$AppClassLoader@55f96302
```

3-Number

Number is the fundamental abstract superclass for numeric values in the Java programming language. It provides a common interface for conversions among different numeric data types and serves as the parent class for all numeric wrapper classes such as Byte, Short, Integer, Long, Float, and Double. Additionally, classes like BigInteger and BigDecimal (in the java.math package), as well as AtomicInteger and AtomicLong (in the java.util.concurrent.-atomic package), also extend Number.

Main Features

- It is an abstract class and cannot be instantiated directly.
- It implements the Serializable interface.
- It declares abstract methods for numeric conversions.
- It enables conversion across different numeric types.

Constructor

protected Number(): Since Number is abstract, it cannot be instantiated on its own. Subclasses (e.g., Byte, Integer, etc.) call this constructor.

Methods

1. **byteValue():** Converts the numeric value to a byte. Larger or negative values may cause data loss.
2. **shortValue():** Converts the numeric value to a short. Values outside the valid range for short will be truncated.
3. **intValue():** Converts the numeric value to an int. Fractional parts are discarded, and out-of-range values can lead to loss of information.
4. **longValue():** Converts the numeric value to a long. Fractional parts are discarded. Larger values may suffer truncation.
5. **floatValue():** Converts the numeric value to a float. Extremely large or small values may be approximated, resulting in potential precision loss.
6. **doubleValue():** Converts the numeric value to a double. Although it has one of the broadest ranges, very large or precise values can still lose accuracy.

Inheritance Classes

- Integer
- Swim without sinking
- Double
- Long
- Short
- Byte
- BigDecimal
- BigInteger

EXAMPLE USAGE 3

```
package a1_java.lang;
public class a3_NumberExample {
    public static void main(String[] args) {
```

```java
        // Create a Double object with a sample numeric value
        // Double is a subclass of Number and can be instantiated
        Number myNumber = Double.valueOf(45.67);
        // 1. byteValue() Conversion
        // Converts the numeric value to a byte
        // Warning: Larger or negative values may cause data loss
        byte byteVal = myNumber.byteValue();
        System.out.println("1. byteValue() result: " + byteVal);
        // 2. shortValue() Conversion
        // Converts the numeric value to a short
        // Warning: Values outside the valid short range will be truncated
        short shortVal = myNumber.shortValue();
        System.out.println("2. shortValue() result: " + shortVal);
        // 3. intValue() Conversion
        // Converts the numeric value to an integer
        // Fractional parts are discarded, potential information loss
        int intVal = myNumber.intValue();
        System.out.println("3. intValue() result: " + intVal);
        // 4. longValue() Conversion
        // Converts the numeric value to a long
        // Fractional parts are discarded, potential truncation
        long longVal = myNumber.longValue();
        System.out.println("4. longValue() result: " + longVal);
        // 5. floatValue() Conversion
        // Converts the numeric value to a float
        // Warning: Potential precision loss for very large/small values
        float floatVal = myNumber.floatValue();
        System.out.println("5. floatValue() result: " + floatVal);
        // 6. doubleValue() Conversion
        // Converts the numeric value to a double
        // Broadest range, but still potential accuracy issues
        double doubleVal = myNumber.doubleValue();
        System.out.println("6. doubleValue() result: " + doubleVal);
        // Additional demonstration with different initial values
        demonstrateConversionVariations(); }
    private static void demonstrateConversionVariations() {
        System.out.println("\n--- Conversion Variations ---");
        // Large value to demonstrate potential truncation
        Number largeNumber = Double.valueOf(123456.789);
        System.out.println("Large Number: " + largeNumber);
        // Negative value to show conversion behavior
        Number negativeNumber = Double.valueOf(-45.67);
        System.out.println("Negative Number: " + negativeNumber);
        // Conversions for large and negative numbers
        System.out.println("Large Number Conversions:");
        System.out.println("byteValue(): " + largeNumber.byteValue());
        System.out.println("shortValue(): " + largeNumber.shortValue());
        System.out.println("intValue(): " + largeNumber.intValue());
        System.out.println("\nNegative Number Conversions:");
        System.out.println("byteValue(): " + negativeNumber.byteValue());
        System.out.println("shortValue(): " + negativeNumber.shortValue());
        System.out.println("intValue(): " + negativeNumber.intValue()); } }
```

OUTPUT

```
1. byteValue() result: 45
2. shortValue() result: 45
3. intValue() result: 45
4. longValue() result: 45
5. floatValue() result: 45.67
6. doubleValue() result: 45.67

--- Conversion Variations ---
Large Number: 123456.789
Negative Number: -45.67
Large Number Conversions:
byteValue(): 64
shortValue(): -7616
intValue(): 123456

Negative Number Conversions:
byteValue(): -45
shortValue(): -45
intValue(): -45
```

🔷 Numeric Wrapper Classes

4-Integer

The Integer class is a wrapper class in Java that represents the primitive int type as an object. This class enables us to use int values as objects and provides various useful methods.

Key Class Characteristics

- Defined as final, meaning it cannot be inherited.
- Extends the Number class.
- Implements the Comparable interface.
- An immutable class.

Constructors Explanation

1. **`Integer(int value)`:** Converts a primitive int value to an Integer object. This constructor stores the given integer value in the object's internal field. Purpose:
   - ✓ Converting primitive int to object type.
   - ✓ Object-based numerical operations.
2. **`Integer(String s)`:** ConstructorConverts a String expression to an Integer object. The String must be in a valid integer format. Characteristics:
   - ✓ Performs conversion in base 10 by default.
   - ✓ Throws NumberFormatException for invalid number formats.

Additional Notes

- Constructors and methods can contain null values.
- `valueOf()` methods are recommended for object creation performance.
- Number range checks are performed during conversion.

Constant Values

⇒ ***MIN_VALUE***: -2^31 (-2,147,483,648).
⇒ ***MAX_VALUE***: 2^31 - 1 (2,147,483,647).
⇒ ***SIZE***: 32 bits.
⇒ ***BYTES***: 4 bytes.
⇒ ***TYPE***: int.class.

Methods Categories

A. Conversion Methods

1. **`byteValue()`:** Converts int to byte.
2. **`doubleValue()`:** Converts int to double.
3. **`floatValue()`:** Converts int to float.
4. **`intValue()`:** Returns int value.
5. **`longValue()`:** Converts int to long.
6. **`shortValue()`:** Converts int to short.
7. **`toString()`:** Converts int to string.

B. Static Conversion Methods

8. **`parseInt(String s)`:** Converts String to int.
9. **`parseInt(String s, int radix)`:** Conversion in different bases.
10. **`valueOf(String s)`:** Creates Integer object from String.
11. **`valueOf(int i)`:** Creates Integer object from int.
12. **`toString(int i)`:** Converts int to string.
13. **`toHexString(int i)`:** Hexadecimal representation.
14. **`toOctalString(int i)`:** Octal representation.
15. **`toBinaryString(int i)`:** Binary representation.

C. Comparison Methods

16. **`compareTo(Integer anotherInteger)`:** Integer comparison.
17. **`compare(int x, int y)`:** Static comparison.
18. **`equals(Object obj)`:** Equality check.
19. **`hashCode()`:** Generates hash code.

D. Bit Manipulation Methods

20. **`bitCount(int i)`:** Counts 1 bits.
21. **`highestOneBit(int i)`:** Finds highest 1 bit
22. **`lowestOneBit(int i)`:** Finds lowest 1 bit.
23. **`numberOfLeadingZeros(int i)`:** Counts leading zeros.
24. **`numberOfTrailingZeros(int i)`:** Counts trailing zeros.
25. **`reverse(int i)`:** Reverses bits
26. **`reverseBytes(int i)`:** Reverses byte order.

E. Utility Methods

27.signum(int i): Sign check.

28.rotateLeft(int i, int distance): Left rotation.

29.rotateRight(int i, int distance): Right rotation.

EXAMPLE USAGE 4

```
package a1_java.lang;
public class a4_IntegerExample {
    public static void main(String[] args) {
        // 1. Constructor Methods Demonstration
        // Method 1: Integer(int value) - Creates Integer object from primitive int
        @SuppressWarnings("removal")
              Integer numberFromPrimitive = new Integer(100);
        System.out.println("Integer from Primitive: " + numberFromPrimitive);
        // Method 2: Integer(String s) - Creates Integer object from String
        @SuppressWarnings("removal")
              Integer numberFromString = new Integer("250");
        System.out.println("Integer from String: " + numberFromString);
        // 2. Constant Values Demonstration
        System.out.println("Minimum Integer Value: " + Integer.MIN_VALUE);
        System.out.println("Maximum Integer Value: " + Integer.MAX_VALUE);
        System.out.println("Integer Size in Bits: " + Integer.SIZE);
        System.out.println("Integer Bytes: " + Integer.BYTES);
        System.out.println("Integer Type: " + Integer.TYPE);
        // 3. Conversion Methods (A)
        int originalNumber = 500;
        if (originalNumber > 0) {
            System.out.println("Original Number is positive: " + originalNumber); }
        // Method 1-7: Various Conversion Methods
        byte byteValue = numberFromPrimitive.byteValue();
        double doubleValue = numberFromPrimitive.doubleValue();
        float floatValue = numberFromPrimitive.floatValue();
        int intValue = numberFromPrimitive.intValue();
        long longValue = numberFromPrimitive.longValue();
        short shortValue = numberFromPrimitive.shortValue();
        String stringValue = numberFromPrimitive.toString();
        System.out.println("Byte Value: " + byteValue);
        System.out.println("Double Value: " + doubleValue);
        System.out.println("Float Value: " + floatValue);
        System.out.println("Int Value: " + intValue);
        System.out.println("Long Value: " + longValue);
        System.out.println("Short Value: " + shortValue);
        System.out.println("String Value: " + stringValue);
        // 4. Static Conversion Methods (B)
        // Method 8-9: Parse Methods
        int parsedNumber = Integer.parseInt("1000");
        int parsedNumberWithRadix = Integer.parseInt("1010", 2); // Binary to decimal
        // Method 10-11: Value Of Methods
        Integer valueFromString = Integer.valueOf("750");
        Integer valueFromInt = Integer.valueOf(750);
        if (valueFromString.intValue() > 0 && valueFromInt.intValue() > 0) {
            System.out.println("Value From String: " + valueFromString);
            System.out.println("Value From Int: " + valueFromInt); }
        // Method 12-15: String Representation Methods
        String decimalString = Integer.toString(1000);
        String hexString = Integer.toHexString(255);
        String octalString = Integer.toOctalString(64);
        String binaryString = Integer.toBinaryString(10);
        System.out.println("Parsed Number: " + parsedNumber);
        System.out.println("Parsed Binary Number: " + parsedNumberWithRadix);
        System.out.println("Decimal String: " + decimalString);
        System.out.println("Hex String: " + hexString);
        System.out.println("Octal String: " + octalString);
        System.out.println("Binary String: " + binaryString);
        // 5. Comparison Methods (C)
        // Method 16-19: Comparison Methods
        Integer num1 = 100;
        Integer num2 = 200;
        int comparisonResult = num1.compareTo(num2);
        int staticComparisonResult = Integer.compare(100, 200);
        boolean equalityCheck = num1.equals(100);
        int hashCodeValue = num1.hashCode();
        System.out.println("Comparison Result: " + comparisonResult);
```

```
        System.out.println("Static Comparison Result: " + staticComparisonResult);
        System.out.println("Equality Check: " + equalityCheck);
        System.out.println("Hash Code: " + hashCodeValue);
        // 6. Bit Manipulation Methods (D)
        // Method 20-26: Bit Manipulation
        int manipulationNumber = 15; // Binary: 1111
        int bitCount = Integer.bitCount(manipulationNumber);
        int highestOneBit = Integer.highestOneBit(manipulationNumber);
        int lowestOneBit = Integer.lowestOneBit(manipulationNumber);
        int leadingZeros = Integer.numberOfLeadingZeros(manipulationNumber);
        int trailingZeros = Integer.numberOfTrailingZeros(manipulationNumber);
        int reversedBits = Integer.reverse(manipulationNumber);
        int reversedBytes = Integer.reverseBytes(manipulationNumber);
        System.out.println("Bit Count: " + bitCount);
        System.out.println("Highest One Bit: " + highestOneBit);
        System.out.println("Lowest One Bit: " + lowestOneBit);
        System.out.println("Leading Zeros: " + leadingZeros);
        System.out.println("Trailing Zeros: " + trailingZeros);
        System.out.println("Reversed Bits: " + reversedBits);
        System.out.println("Reversed Bytes: " + reversedBytes);
        // 7. Utility Methods (E)
        // Method 27-29: Utility Methods
        int signumPositive = Integer.signum(250);
        int rotatedLeft = Integer.rotateLeft(10, 2);
        int rotatedRight = Integer.rotateRight(10, 2);
        System.out.println("Signum (Positive): " + signumPositive);
        System.out.println("Rotated Left: " + rotatedLeft);
        System.out.println("Rotated Right: " + rotatedRight); } }
```

OUTPUT

```
Integer from Primitive: 100
Integer from String: 250
Minimum Integer Value: -2147483648
Maximum Integer Value: 2147483647
Integer Size in Bits: 32
Integer Bytes: 4
Integer Type: int
Original Number is positive: 500
Byte Value: 100
Double Value: 100.0
Float Value: 100.0
Int Value: 100
Long Value: 100
Short Value: 100
String Value: 100
Value From String: 750
Value From Int: 750
Parsed Number: 1000
Parsed Binary Number: 10
Decimal String: 1000
Hex String: ff
Octal String: 100
Binary String: 1010
Comparison Result: -1
Static Comparison Result: -1
Equality Check: true
Hash Code: 100
Bit Count: 4
Highest One Bit: 8
Lowest One Bit: 1
Leading Zeros: 28
Trailing Zeros: 0
Reversed Bits: -268435456
Reversed Bytes: 251658240
Signum (Positive): 1
Rotated Left: 40
Rotated Right: -2147483646
```

5-Boolean

The Boolean class is a wrapper class for the primitive boolean data type. Located in the java.lang package, it serves to represent boolean values as objects.

Class Characteristics

- Defined as final.
- Inherits from the Object class.
- Implements Serializable and Comparable interfaces.
- An immutable class.

Constructor Methods

1. **`Boolean(boolean value)`:** Creates a Boolean object from a primitive boolean value.
2. **`Boolean(String s)`:** Creates a Boolean object from a string expression.

Static Constant Values

TRUE: Boolean.TRUE constant object.
FALSE: Boolean.FALSE constant object.
TYPE: boolean.class.

Methods

A. Conversion Methods

1. **`booleanValue()`:** Retrieves the primitive boolean value from a Boolean object.
2. **`toString()`:** Converts Boolean value to String.
3. **`static toString(boolean b)`:** Static string conversion method.
4. **`static parseBoolean(String s)`:** Converts String to boolean.

B. Comparison Methods

5. **`equals(Object obj)`:** Object equality check.
6. **`compareTo(Boolean b)`:** Compares Boolean values.
7. **`static compare(boolean x, boolean y)`:** Static boolean comparison.
8. **`hashCode()`:** Generates hash code.

C. Static Utility Methods

9. **`static valueOf(boolean b)`:** Creates a Boolean object.
10. **`static valueOf(String s)`:** Creates a Boolean object from a String.
11. **`static logicalAnd(boolean a, boolean b)`:** Logical AND operation.
12. **`static logicalOr(boolean a, boolean b)`:** Logical OR operation.
13. **`static logicalXor(boolean a, boolean b)`:** Logical XOR operation.

EXAMPLE USAGE 5

```
package a1_java.lang;
public class a5_BooleanExample {
    public static void main(String[] args) {
        // 1. Constructor Methods Demonstration
        // Method 1: Boolean(boolean value) - Creates Boolean object from primitive boolean
        @SuppressWarnings("removal")
              Boolean boolFromPrimitive = new Boolean(true);
        System.out.println("Boolean from Primitive: " + boolFromPrimitive);
        // Method 2: Boolean(String s) - Creates Boolean object from String
        @SuppressWarnings("removal")
              Boolean boolFromString = new Boolean("false");
        System.out.println("Boolean from String: " + boolFromString);
        // 2. Static Constant Values Demonstration
        System.out.println("Boolean TRUE Constant: " + Boolean.TRUE);
        System.out.println("Boolean FALSE Constant: " + Boolean.FALSE);
        System.out.println("Boolean Type: " + Boolean.TYPE);
        // 3. Conversion Methods (A)
        // Method 1: booleanValue() - Retrieves primitive boolean value
        boolean primitiveValue = boolFromPrimitive.booleanValue();
        System.out.println("Primitive Value: " + primitiveValue);
        // Method 2: toString() - Converts Boolean to String
        String boolToString = boolFromPrimitive.toString();
        System.out.println("Boolean to String: " + boolToString);
        // Method 3: static toString(boolean b) - Static string conversion
        String staticToString = Boolean.toString(true);
        System.out.println("Static ToString: " + staticToString);
        // Method 4: static parseBoolean(String s) - Converts String to boolean
        boolean parsedBool = Boolean.parseBoolean("true");
        System.out.println("Parsed Boolean: " + parsedBool);
        // 4. Comparison Methods (B)
        // Method 5: equals(Object obj) - Object equality check
        boolean equalityCheck = boolFromPrimitive.equals(true);
        System.out.println("Equality Check: " + equalityCheck);
```

```java
        // Method 6: compareTo(Boolean b) - Compares Boolean values
        int comparisonResult = boolFromPrimitive.compareTo(boolFromString);
        System.out.println("Comparison Result: " + comparisonResult);
        // Method 7: static compare(boolean x, boolean y) - Static boolean comparison
        int staticCompareResult = Boolean.compare(true, false);
        System.out.println("Static Comparison Result: " + staticCompareResult);
        // Method 8: hashCode() - Generates hash code
        int hashCodeValue = boolFromPrimitive.hashCode();
        System.out.println("Hash Code: " + hashCodeValue);
        // 5. Static Utility Methods (C)
        // Method 9: static valueOf(boolean b) - Creates Boolean object
        Boolean valueOfBoolean = Boolean.valueOf(true);
        System.out.println("ValueOf Boolean: " + valueOfBoolean);
        // Method 10: static valueOf(String s) - Creates Boolean object from String
        Boolean valueOfString = Boolean.valueOf("false");
        System.out.println("ValueOf String: " + valueOfString);
        // Method 11: static logicalAnd(boolean a, boolean b) - Logical AND operation
        boolean andResult = Boolean.logicalAnd(true, false);
        System.out.println("Logical AND Result: " + andResult);
        // Method 12: static logicalOr(boolean a, boolean b) - Logical OR operation
        boolean orResult = Boolean.logicalOr(true, false);
        System.out.println("Logical OR Result: " + orResult);
        // Method 13: static logicalXor(boolean a, boolean b) - Logical XOR operation
        boolean xorResult = Boolean.logicalXor(true, false);
        System.out.println("Logical XOR Result: " + xorResult); } }
```

OUTPUT

```
Boolean from Primitive: true
Boolean from String: false
Boolean TRUE Constant: true
Boolean FALSE Constant: false
Boolean Type: boolean
Primitive Value: true
Boolean to String: true
Static ToString: true
Parsed Boolean: true
Equality Check: true
Comparison Result: 1
Static Comparison Result: 1
Hash Code: 1231
ValueOf Boolean: true
ValueOf String: false
Logical AND Result: false
Logical OR Result: true
Logical XOR Result: true
```

6-Void

The Void class is a special wrapper class representing methods that do not return any value. Located in the java.lang package, it is typically used as a return type for methods. The Void class represents the empty return type of methods in Java. With no methods of its own, it is primarily used for type safety and reflection mechanisms.

Class Characteristics

- Defined as final.
- Inherits from the Object class.
- No instances can be created.
- Follows the Singleton design pattern.

Static Constant Values

TYPE: void.class.

Method Categories

Static Utility Methods: The Void class has no methods.

Usage Purposes

- Specifying empty return types for methods.
- Used as a type parameter in generic programming.
- Used as a return type in Reflection API.

Inherited Methods

Standard methods from the Object class:

1. **equals()**
2. **hashCode()**
3. **toString()**

Void Usage Scenarios

1. Method Return Type

```java
public void processData() {
// Method that returns nothing
}
```

2. Generic Programming

```java
public class Result<T, E extends Void> {
       private T successResult;
       // Indicates no error of Void type
}
```

3. Reflection API Usage

```
Method method = SomeClass.class.getMethod("voidMethod");
if (method.getReturnType() == Void.TYPE) {
       System.out.println("Method returns nothing"); }
```

EXAMPLE USAGE 6

```
package a1_java.lang;
import java.lang.reflect.Method;
import java.util.function.Supplier;
public class a6_VoidExample {
    // 1. Basic Void Method (Method Return Type Scenario)
    // Explanation: Void methods (void keyword) do not return any value, used for performing actions
    public static void processData() {
        System.out.println("Processing data with void method"); }
    // 2. Generic Programming Demonstration
    // Explanation: Incorrect use of Void in generic type parameters (not recommended)
    public static class Result<T> {
        private T successResult;
        public Result(T result) {
            this.successResult = result; }
        public T getSuccessResult() {
            return successResult; } }
    // 3. Reflection API Usage (Void Method Detection)
    // Explanation: Demonstrating how to detect void methods using Reflection API
    public static void voidMethodForReflection() {
        System.out.println("Void method for reflection"); }
    // 4. Void in Functional Interface
    // Explanation: Using Supplier<Void> to represent an action without a meaningful return
    public static Supplier<Void> createVoidAction() {
        return () -> {
            System.out.println("Performing action via Void supplier");
            // Explicit null return for Void
            return null; }; }
    // 5. Void as Explicit Return Type
    // Explanation: Rare scenario of explicitly returning null for Void type
    public Void performComplexOperation() {
        System.out.println("Complex operation performed");
        // Explicitly returning null for Void
        return null; }
    public static void main(String[] args) {
        // Void Demonstration Scenarios
        // 1. Basic Void Method Invocation
        // Demonstrates a method that performs an action without returning a value
        processData();
        // 2. Result Demonstration (Generic Usage)
        Result<String> result = new Result<>("Success");
        System.out.println("Generic Result: " + result.getSuccessResult());
        // 3. Reflection API Void Method Analysis
        try {
            Method method = a6_VoidExample.class.getMethod("voidMethodForReflection");
            // Check if method returns void
            if (method.getReturnType() == void.class) {
                System.out.println("Method is a void method"); }
            // 4. Functional Interface with Void
            Supplier<Void> voidAction = createVoidAction();
            voidAction.get(); // Executes the action
            // 5. Void Type Characteristics
            Class<?> voidType = Void.TYPE;
            System.out.println("Void Type toString(): " + voidType.toString());
            System.out.println("Void Type hashCode: " + voidType.hashCode());
        } catch (NoSuchMethodException e) {
            e.printStackTrace(); }
        // 6. Void Type Comparison
        // Explanation: Analyzing Void type characteristics
        Class<?> voidType = Void.TYPE;
        System.out.println("Is Void a primitive type? " + voidType.isPrimitive());
        System.out.println("Void Type Name: " + voidType.getName()); } }
```

OUTPUT

```
Processing data with void method
Generic Result: Success
Method is a void method
Performing action via Void supplier
Void Type toString(): void
Void Type hashCode: 821270929
Is Void a primitive type? true
Void Type Name: void
```

◈ String and Text Processing
7-String

The String class is an immutable class representing text sequences. Located in the java.lang package, it provides comprehensive methods for processing character arrays.

Class Characteristics

- Immutable class.
- Implements Serializable and Comparable interfaces.
- Directly inherits from the Object class.
- Supports Unicode character strings.

Constructors

1. **`String()`:** Creates an empty string.
2. **`String(String original)`:** Creates a copy of an existing string.
3. **`String(char[] value)`:** Creates a string from a character array.
4. **`String(byte[] bytes)`:** Creates a string from a byte array.
5. **`String(StringBuilder builder)`:** Creates a string from a StringBuilder.
6. **`String(StringBuffer buffer)`:** Creates a string from a StringBuffer.

Methods

A. Conversion Methods

1. **`toString()`:** Returns the string itself.
2. **`toCharArray()`:** Converts string to a character array.
3. **`getBytes()`:** Converts string to a byte array.
4. **`valueOf()`:** Converts different types to string.

B. Comparison Methods

5. **`equals()`:** Checks content equality.
6. **`equalsIgnoreCase()`:** Case-insensitive equality.
7. **`compareTo()`:** Lexicographical ordering comparison.
8. **`compareToIgnoreCase()`:** Case-insensitive comparison.

C. Search Methods

9. **`contains()`:** Checks for substring presence.
10. **`startsWith()`:** Checks if string starts with a specific prefix.
11. **`endsWith()`:** Checks if string ends with a specific suffix.
12. **`indexOf()`:** Finds substring location
13. **`lastIndexOf()`:** Finds last matching location.

D. Modification Methods

14. **`substring()`:** Extracts a portion of the string.
15. **`trim()`:** Removes leading and trailing whitespaces.
16. **`replace()`:** Replaces characters.
17. **`replaceAll()`:** Replaces using regular expressions.
18. **`toLowerCase()`:** Converts all characters to lowercase.
19. **`toUpperCase()`:** Converts all characters to uppercase.

E. Analysis Methods

20. **`length()`:** Returns string length.
21. **`isEmpty()`:** Checks if string is empty.
22. **`charAt()`:** Returns character at specific position.

F. Splitting and Concatenation Methods

23. **`split()`:** Splits string into substrings.
24. **`join()`:** Joins strings.
25. **`concat()`:** Concatenates strings sequentially.

G. Static Utility Methods

26. **`format()`:** Creates formatted string.
27. **`copyValueOf()`:** Creates string from character array.

Key Design Principles

- Immutability ensures thread safety.
- Optimized for string operations.
- Extensive method set for text manipulation.
- Supports internationalization through Unicode.

Performance Considerations

- Use StringBuilder for multiple string modifications.
- Consider *intern()* method for constant strings.
- Be mindful of performance in large text operations.

EXAMPLE USAGE 7

```
package a1_java.lang;
public class a7_StringExample {
    public static void main(String[] args) {
        // 1. Constructor Demonstrations
        // Method 1: Empty String Constructor
        String emptyString = new String();
        System.out.println("Empty String: " + emptyString);
        // Method 2: Copy Constructor
        String originalString = "Hello, World!";
        String copiedString = new String(originalString);
        System.out.println("Copied String: " + copiedString);
        // Method 3: Character Array Constructor
        char[] charArray = {'J', 'a', 'v', 'a'};
        String charArrayString = new String(charArray);
        System.out.println("Char Array String: " + charArrayString);
        // Method 4: Byte Array Constructor
        byte[] byteArray = {74, 97, 118, 97};
        String byteArrayString = new String(byteArray);
        System.out.println("Byte Array String: " + byteArrayString);
        // Method 5: StringBuilder Constructor
        StringBuilder builder = new StringBuilder("StringBuilder Example");
        String builderString = new String(builder);
        System.out.println("StringBuilder String: " + builderString);
        // Method 6: StringBuffer Constructor
        StringBuffer buffer = new StringBuffer("StringBuffer Example");
        String bufferString = new String(buffer);
        System.out.println("StringBuffer String: " + bufferString);
        // A. Conversion Methods
        // Method 1: toString()
        String conversionString = "Conversion Method";
        System.out.println("ToString: " + conversionString.toString());
        // Method 2: toCharArray()
        char[] convertedCharArray = conversionString.toCharArray();
        System.out.println("Char Array: " + java.util.Arrays.toString(convertedCharArray));
        // Method 3: getBytes()
        byte[] convertedByteArray = conversionString.getBytes();
        System.out.println("Byte Array: " + java.util.Arrays.toString(convertedByteArray));
        // Method 4: valueOf()
        String intString = String.valueOf(42);
        String doubleString = String.valueOf(3.14);
        System.out.println("Int to String: " + intString);
        System.out.println("Double to String: " + doubleString);
        // B. Comparison Methods
        String str1 = "Hello";
        String str2 = "hello";
        // Method 5: equals()
        boolean isEqual = str1.equals(str2);
        System.out.println("Equals: " + isEqual);
        // Method 6: equalsIgnoreCase()
        boolean isEqualIgnoreCase = str1.equalsIgnoreCase(str2);
        System.out.println("Equals Ignore Case: " + isEqualIgnoreCase);
        // Method 7: compareTo()
        int compareResult = str1.compareTo(str2);
        System.out.println("Compare To: " + compareResult);
        // Method 8: compareToIgnoreCase()
        int compareIgnoreCaseResult = str1.compareToIgnoreCase(str2);
        System.out.println("Compare To Ignore Case: " + compareIgnoreCaseResult);
        // C. Search Methods
        String searchString = "Hello, World! Welcome to Java Programming";
```

```java
// Method 9: contains()
boolean containsWorld = searchString.contains("World");
System.out.println("Contains 'World': " + containsWorld);
// Method 10: startsWith()
boolean startsWithHello = searchString.startsWith("Hello");
System.out.println("Starts with 'Hello': " + startsWithHello);
// Method 11: endsWith()
boolean endsWithProgramming = searchString.endsWith("Programming");
System.out.println("Ends with 'Programming': " + endsWithProgramming);
// Method 12: indexOf()
int worldIndex = searchString.indexOf("World");
System.out.println("Index of 'World': " + worldIndex);
// Method 13: lastIndexOf()
int lastOIndex = searchString.lastIndexOf("o");
System.out.println("Last Index of 'o': " + lastOIndex);
// D. Modification Methods
String modificationString = "  Java Programming  ";
// Method 14: substring()
String subStr = modificationString.substring(2, 10);
System.out.println("Substring: " + subStr);
// Method 15: trim()
String trimmedStr = modificationString.trim();
System.out.println("Trimmed String: " + trimmedStr);
// Method 16: replace()
String replacedStr = modificationString.replace('a', 'X');
System.out.println("Replaced String: " + replacedStr);
// Method 17: replaceAll()
String regexReplacedStr = modificationString.replaceAll("\\s", "_");
System.out.println("Regex Replaced String: " + regexReplacedStr);
// Method 18: toLowerCase()
String lowerCaseStr = modificationString.toLowerCase();
System.out.println("Lowercase: " + lowerCaseStr);
// Method 19: toUpperCase()
String upperCaseStr = modificationString.toUpperCase();
System.out.println("Uppercase: " + upperCaseStr);
// E. Analysis Methods
// Method 20: length()
int stringLength = modificationString.length();
System.out.println("String Length: " + stringLength);
// Method 21: isEmpty()
boolean isEmptyCheck = modificationString.isEmpty();
System.out.println("Is Empty: " + isEmptyCheck);
// Method 22: charAt()
char specificChar = modificationString.charAt(5);
System.out.println("Character at index 5: " + specificChar);
// F. Splitting and Concatenation Methods
String csvString = "Apple,Banana,Cherry,Date";
// Method 23: split()
String[] fruits = csvString.split(",");
System.out.println("Split Fruits: " + java.util.Arrays.toString(fruits));
// Method 24: join()
String joinedFruits = String.join(" | ", fruits);
System.out.println("Joined Fruits: " + joinedFruits);
// Method 25: concat()
String concatString = "Hello, ".concat("World!");
System.out.println("Concatenated String: " + concatString);
// G. Static Utility Methods
// Method 26: format()
String formattedString = String.format("Name: %s, Age: %d", "John", 30);
System.out.println("Formatted String: " + formattedString);
// Method 27: copyValueOf()
String copyValueString = String.copyValueOf(charArray);
System.out.println("Copy Value String: " + copyValueString); } }
```

OUTPUT

```
Empty String:
Copied String: Hello, World!
```

```
Char Array String: Java
Byte Array String: Java
StringBuilder String: StringBuilder Example
StringBuffer String: StringBuffer Example
ToString: Conversion Method
Char Array: [C, o, n, v, e, r, s, i, o, n,  , M, e, t, h, o, d]
Byte Array: [67, 111, 110, 118, 101, 114, 115, 105, 111, 110, 32, 77, 101, 116, 104, 111,
100]
Int to String: 42
Double to String: 3.14
Equals: false
Equals Ignore Case: true
Compare To: -32
Compare To Ignore Case: 0
Contains 'World': true
Starts with 'Hello': true
Ends with 'Programming': true
Index of 'World': 7
Last Index of 'o': 32
Substring: Java Pro
Trimmed String: Java Programming
Replaced String:   JXvX ProgrXmming
Regex Replaced String: __Java_Programming__
Lowercase:   java programming
Uppercase:   JAVA PROGRAMMİNG
String Length: 20
Is Empty: false
Character at index 5: a
Split Fruits: [Apple, Banana, Cherry, Date]
Joined Fruits: Apple | Banana | Cherry | Date
Concatenated String: Hello, World!
Formatted String: Name: John, Age: 30
Copy Value String: Java
```

8-StringBuilder

StringBuilder is a mutable class in the java.lang package used for efficiently and dynamically managing textual data (character sequences) in Java. Unlike the String class, it has "mutable" properties, which provide significant advantages in terms of memory and performance, especially in situations where frequent changes to the text are made (such as repeated concatenation, deletion, and insertion operations).

Key Features

⇒ **Mutable**: Allows direct modification of the character sequence after creation.

⇒ **Not Thread-Safe**: May require external synchronization if accessed by multiple threads simultaneously.

⇒ **Dynamic Memory Management**: The capacity of the character sequence automatically expands as needed.

⇒ **High Performance**: Particularly fast and efficient in scenarios with numerous concatenation operations compared to the String class.

Constructors

1. **`StringBuilder()`:** The default constructor. Creates an empty StringBuilder object with a default capacity of 16 characters.
2. **`StringBuilder(int capacity)`:** Creates a StringBuilder with the specified initial capacity. Pre-defining the initial capacity can improve performance in scenarios requiring frequent growth.
3. **`StringBuilder(String str)`:** Initializes the content with the specified String value. The initial capacity is calculated by adding 16 to the length of the entered string.
4. **`StringBuilder(CharSequence seq)`:** Initializes with any CharSequence (such as String, StringBuffer, StringBuilder, CharBuffer). The capacity is set by adding 16 to the length of `seq`.

Methods

A. Append Methods

1. **append(boolean b):** Appends the string representation of the boolean value.
2. **append(char c):** Appends the string representation of the character.
3. **append(char[] str):** Appends the string representation of the character array.
4. **append(char[] str, int offset, int len):** Appends a portion of the character array.
5. **append(CharSequence s):** Appends the string representation of the character sequence.
6. **append(CharSequence s, int start, int end):** Appends a portion of the character sequence.
7. **append(double d):** Appends the string representation of the double value.
8. **append(float f):** Appends the string representation of the float value.
9. **append(int i):** Appends the string representation of the integer value.
10. **append(long lng):** Appends the string representation of the long value.
11. **append(Object obj):** Appends the string representation of the object.
12. **append(String str):** Appends the specified string value.
13. **append(StringBuffer sb):** Appends the string representation of the StringBuffer object.
14. **append(StringBuilder sb):** Appends the string representation of the StringBuilder object.
15. **appendCodePoint(int codePoint):** Appends the specified character (Unicode code point) to the sequence.

B. Insert Methods

16. **insert(int offset, boolean b):** Inserts the string representation of the boolean value at the specified offset.
17. **insert(int offset, char c):** Inserts the character at the specified offset.
18. **insert(int offset, char[] str):** Inserts the character array starting from the specified offset.
19. **insert(int offset, char[] str, int strOffset, int strLen):** Inserts a specified portion of a character array.
20. **insert(int offset, CharSequence s):** Inserts the character sequence starting from the specified offset.
21. **insert(int offset, CharSequence s, int start, int end):** Inserts a specified range of the character sequence.
22. **insert(int offset, double d):** Inserts the string representation of the double value at the specified offset.
23. **insert(int offset, float f):** Inserts the string representation of the float value at the specified offset.
24. **insert(int offset, int i):** Inserts the string representation of the integer value at the specified offset.
25. **insert(int offset, long l):** Inserts the string representation of the long value at the specified offset.
26. **insert(int offset, Object obj):** Inserts the string representation of the object at the specified offset.
27. **insert(int offset, String str):** Inserts the specified string at the specified offset.

C. Deletion Methods

28. **delete(int start, int end):** Deletes the characters in the specified range (from start to end).
29. **deleteCharAt(int index):** Deletes the character at the specified index.

D. Conversion Methods

30. **toString():** Returns the current character sequence as a String.
31. **substring(int start):** Creates a substring from the specified start position to the end.
32. **substring(int start, int end):** Creates a substring from the specified range (from start to end).
33. **subSequence(int start, int end):** Returns a subsequence of characters, as specified by the CharSequence interface.

E. Size & Capacity Methods

34. **length():** Returns the number of characters currently present.

35. **`capacity()`**: Returns the current capacity of the StringBuilder.
36. **`ensureCapacity(int minimumCapacity)`**: Ensures that the capacity is at least equal to the specified minimum capacity (increases if necessary).
37. **`trimToSize()`**: Reduces the capacity to the current number of characters.
38. **`setLength(int newLength)`**: Directly changes the current character length. If the length is reduced, excess characters are deleted; if increased, `\u0000` (null) is added.

F. Modification Methods

39. **`replace(int start, int end, String str)`**: Replaces the characters in the specified range with the specified string.
40. **`reverse()`**: Reverses the current character sequence.
41. **`setCharAt(int index, char ch)`**: Modifies the character at the specified index directly.

G. Search Methods

42. **`indexOf(String str)`**: Returns the index of the first occurrence of the specified substring.
43. **`indexOf(String str, int fromIndex)`**: Starts searching for the specified substring from the specified index.
44. **`lastIndexOf(String str)`**: Returns the index of the last occurrence of the specified substring.
45. **`lastIndexOf(String str, int fromIndex)`**: Limits the search to the specified fromIndex.

H. Character Retrieval Methods

46. **`charAt(int index)`**: Returns the character at the specified index.
47. **`getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`**: Copies characters from the specified range to the destination array.
48. **`codePointAt(int index)`**: Returns the Unicode code point at the specified position. (Inherits from AbstractStringBuilder)
49. **`codePointBefore(int index)`**: Returns the Unicode code point just before the specified index.
50. **`codePointCount(int beginIndex, int endIndex)`**: Returns the number of Unicode code points between two indexes.
51. **`offsetByCodePoints(int index, int codePointOffset)`**: Calculates the new index after moving forward/backward by the specified number of Unicode code points.

I. Other Important Methods

52. **`equals(Object obj)`**: Typically checks for reference equality for the StringBuilder (does not check for content equality).
53. **`toString()`**: Returns the contents of the StringBuilder as a new String object (important and frequently used).

EXAMPLE USAGE 8

```
package a1_java.lang;
public class a8_StringBuilderExample {
    public static void main(String[] args) {
        // Demonstrating different constructors
        // 1. Default constructor (Constructor #1)
        StringBuilder sb1 = new StringBuilder("Unused Default Constructor");
        System.out.println("Default Constructor Demo: " + sb1);
        // 2. Constructor with specific initial capacity (Constructor #2)
        StringBuilder sb2 = new StringBuilder(50);
        sb2.append("Capacity Initialized StringBuilder");
        System.out.println("Capacity Initialized Demo: " + sb2);
        System.out.println("Capacity of sb2: " + sb2.capacity()); // Use capacity
        // 3. Constructor with initial string (Constructor #3)
        StringBuilder sb3 = new StringBuilder("Initial Content");
        System.out.println("Initial Content Demo: " + sb3);
        // 4. Constructor with CharSequence (Constructor #4)
        CharSequence charSeq = "Hello, CharSequence!";
        StringBuilder sb4 = new StringBuilder(charSeq);
        System.out.println("CharSequence Constructor Demo: " + sb4);
        // Append Methods Demonstration
        StringBuilder appendDemo = new StringBuilder();
        // Appending various types of data (Methods #1-14)
```

```
        appendDemo.append(true); // boolean
        appendDemo.append('A'); // char
        char[] charArray = {'J', 'A', 'V', 'A'};
        appendDemo.append(charArray); // char array
        appendDemo.append(charArray, 1, 2); // partial char array
        appendDemo.append("Programming"); // string
        appendDemo.append(2023); // int
        appendDemo.append(3.14); // double
        appendDemo.append(2.5f); // float
        appendDemo.append(1000L); // long
        appendDemo.append(new Object()); // object
        // Append Unicode code point (Method #15)
        appendDemo.appendCodePoint(65); // Appends 'A'
        // Insert Methods Demonstration
        StringBuilder insertDemo = new StringBuilder("Hello World");
        // Inserting various types at specific offsets (Methods #16-27)
        insertDemo.insert(5, " Beautiful"); // string
        insertDemo.insert(0, true); // boolean
        insertDemo.insert(1, 'X'); // char
        insertDemo.insert(2, charArray); // char array
        insertDemo.insert(3, 2023); // int
        insertDemo.insert(4, 3.14); // double
        // Deletion Methods (Methods #28-29)
        StringBuilder deleteDemo = new StringBuilder("Delete Example");
        deleteDemo.delete(6, 13); // Delete a range
        deleteDemo.deleteCharAt(5); // Delete single character
        // Conversion Methods (Methods #30-33)
        StringBuilder conversionDemo = new StringBuilder("Conversion Demo");
        String fullString = conversionDemo.toString(); // Convert to String
        String subString1 = conversionDemo.substring(0, 10); // Substring with end
        String subString2 = conversionDemo.substring(5); // Substring from start
        System.out.println("Substring 1: " + subString1);
        System.out.println("Substring 2: " + subString2);
        CharSequence subSeq = conversionDemo.subSequence(0, 5);
        System.out.println("SubSequence: " + subSeq);
        // Size & Capacity Methods (Methods #34-38)
        StringBuilder capacityDemo = new StringBuilder("Capacity Demonstration");
        int length = capacityDemo.length(); // Get current length
        int capacity = capacityDemo.capacity(); // Get current capacity
        System.out.println("Capacity of capacityDemo: " + capacity); // Use capacity
        capacityDemo.ensureCapacity(100); // Ensure minimum capacity
        capacityDemo.trimToSize(); // Trim to actual length
        capacityDemo.setLength(10); // Set specific length
        // Modification Methods (Methods #39-41)
        StringBuilder modificationDemo = new StringBuilder("Modification Example");
        modificationDemo.replace(0, 12, "Modified"); // Replace range
        modificationDemo.reverse(); // Reverse entire string
        modificationDemo.setCharAt(5, 'X'); // Set character at specific index
        // Search Methods (Methods #42-45)
        StringBuilder searchDemo = new StringBuilder("Search in this string");
        int firstIndex = searchDemo.indexOf("in"); // First occurrence
        int lastIndex = searchDemo.lastIndexOf("in"); // Last occurrence
        System.out.println("Last Index of 'in': " + lastIndex); // Use lastIndex
        int indexFromPosition = searchDemo.indexOf("in", 10); // Index from specific
position
        System.out.println("Index from position 10: " + indexFromPosition); // Use
indexFromPosition
        // Character Retrieval Methods (Methods #46-51)
        StringBuilder retrievalDemo = new StringBuilder("Character Retrieval");
        char specificChar = retrievalDemo.charAt(5); // Get character at index
        // Copy characters to an array
        char[] destArray = new char[10];
        retrievalDemo.getChars(0, 5, destArray, 0);
```