Session 1 - Java Strings

Agenda

1. **Understanding Java Strings**
2. **Creating Strings**
3. **Escape Sequences**
4. **String Input**
5. **String Arrays**
6. **Strings as Method Parameters**
7. **Exception Handling**

Part 1: Understanding Java Strings

What are Java Strings?

In Java, a String is an object that represents a sequence of characters. Unlike primitive data types, Strings are reference types and are immutable (cannot be changed after creation).

```
// String literal - stored in String pool
String name = "Java Programming";
```

// Key characteristics:
1. Immutable - original string never changes
2. Reference type - stored in heap memory
3. String pool optimization for literals

Theoretical Concepts:

1. String Immutability

- **Definition**: Once a String object is created, its content cannot be modified
- **Memory Implications**: Any "modification" creates a new String object
- **Thread Safety**: Immutable objects are inherently thread-safe
- **Performance**: Enables string pooling and caching optimizations

```
String original = "Hello";
String modified = original.concat(" World"); // Creates new object
// 'original' still contains "Hello"
```

2. String Pool (String Interning)

The **String Pool** (also known as the **String Intern Pool**) is a special area in the **Java heap memory** where **String literals** are stored to improve **memory efficiency and performance**.

- **Location**: Special area in heap memory (PermGen in Java 7, Metaspace in Java 8+)
- **Purpose**: Optimize memory by storing only one copy of each string literal
- **Mechanism**: JVM automatically interns string literals
- **Benefits**: Reduces memory usage and enables fast == comparison for literals

3. String vs StringBuilder vs StringBuffer

`StringBuilder` and `StringBuffer` are classes used to create **mutable (modifiable)** sequences of characters, unlike the `String` class which is **immutable**.

Feature	String	StringBuilder	StringBuffer
Mutability	Immutable	Mutable	Mutable
Thread Safety	Yes	No	Yes
Performance	Slow for concatenation	Fast	Moderate
Memory	Creates new objects	Modifies existing buffer	Modifies existing buffer

String Literals vs String Objects

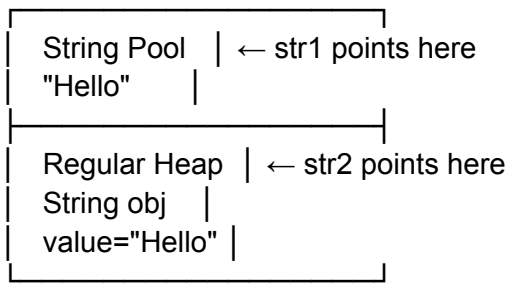
```
// String literal (recommended)
String str1 = "Hello";

// Using new keyword (creates new object)
String str2 = new String("Hello");

// Memory difference:
// str1 points to string pool
// str2 creates new object in heap
```

Memory Layout:

Heap Memory:



Part 2: Creating Strings

Theoretical Concepts:

1. String Creation Mechanisms

Java provides multiple ways to create strings, each with different memory implications and performance characteristics.

2. Compile-time vs Runtime String Creation

- **Compile-time**: String literals are resolved during compilation
- **Runtime**: Constructor calls and method returns create strings during execution
- **Optimization**: JVM can optimize literal concatenation at compile-time

3. String Constructor Overloads

The String class provides multiple constructors for different data sources:

- `String()` - Empty string
- `String(String original)` - Copy constructor
- `String(char[] value)` - From character array
- `String(char[] value, int offset, int count)` - Partial character array
- `String(byte[] bytes)` - From byte array with default charset
- `String(byte[] bytes, Charset charset)` - From byte array with specific charset

Method 1: String Literals

```
String greeting = "Welcome to Java!";
String empty = "";
```

Theory: Stored in string pool, interned automatically, memory efficient

Method 2: Using new Keyword

```
String str = new String("Hello World");
String copy = new String(greeting);
```

Theory: Forces creation of new object in heap, bypasses string pool, higher memory usage

Method 3: From Character Array

```
char[] charArray = {'J', 'a', 'v', 'a'};
String fromArray = new String(charArray);

// Partial array
String partial = new String(charArray, 1, 3); // "ava"
```

Theory: Useful for character manipulation, creates defensive copy, prevents external modification

Method 4: StringBuilder/StringBuffer

```
StringBuilder sb = new StringBuilder();
sb.append("Hello");
sb.append(" ");
sb.append("World");
String result = sb.toString();
```

Theory: Mutable string builders use resizable character arrays, optimal for multiple concatenations

Performance Comparison:

```
// Poor performance - creates multiple String objects
String result = "Hello" + " " + "World" + "!";

// Good performance - single mutable buffer
StringBuilder sb = new StringBuilder();
sb.append("Hello").append(" ").append("World").append("!");
String result = sb.toString();
```

Part 3: Escape Sequences

Theoretical Concepts:

Escape sequences in Java are **special characters preceded by a backslash (`\`)** that are used to represent characters that are otherwise difficult or impossible to include directly in a string.

They help in:

- Formatting strings.
- Representing special characters (like quotes or newlines).
- Avoiding compile-time errors

1. Character Encoding and Representation

- **ASCII**: 7-bit character encoding (0-127)
- **Unicode**: Universal character encoding standard
- **UTF-16**: Java's internal string representation
- **Code Points**: Numerical values representing characters

2. Escape Sequence Categories

1. **Control Characters**: Non-printable characters that control text formatting
2. **Special Characters**: Characters with special meaning in Java syntax
3. **Unicode Escapes**: Direct representation of Unicode code points

3. Compilation Process

- Escape sequences are processed during compilation
- Converted to actual character values in bytecode
- Runtime performance not affected by escape sequences

4. Common Escape Sequences Table

Escape	Character	ASCII	Description
`\n`	Line Feed	10	New line
`\t`	Tab	9	Horizontal tab
`\r`	Carriage Return	13	Return cursor to line start
`\"`	Quote	34	Double quote

`\'`	Apostrophe	39	Single quote
`\\`	Backslash	92	Literal backslash
`\b`	Backspace	8	Move cursor back
`\f`	Form Feed	12	Page break

Escape sequences allow you to include special characters in strings:

```
public class EscapeSequences {
    public static void main(String[] args) {
        // Common escape sequences
        String newline = "Line 1\nLine 2";
        String tab = "Column1\tColumn2";
        String quote = "He said \"Hello\"";
        String backslash = "Path: C:\\Users\\Java";
        String carriageReturn = "First\rSecond";
        String backspace = "Hello\bWorld";
        String formFeed = "Page1\fPage2";

        // Unicode escape - must be exactly 4 hex digits
        String unicode = "\u0048\u0065\u006C\u006C\u006F"; // "Hello"

        // Octal escape (legacy, not recommended)
        String octal = "\110\145\154\154\157"; // "Hello"
        System.out.println("Newline:\n" + newline);
        System.out.println("Tab:\n" + tab);
        System.out.println("Quote: " + quote);
        System.out.println("Backslash: " + backslash);
        System.out.println("Unicode: " + unicode);
    }
}
```

Platform Independence:

```
// Platform-specific line separator
String platformNewline = System.lineSeparator(); // \n on Unix, \r\n on
Windows
```

```
// File separator
String platformPath = "folder" + System.getProperty("file.separator") +
"file.txt";
```

Part 4: String Input

Theoretical Concepts:

1. Input Stream Hierarchy

InputStream (abstract)
  └── FileInputStream
  └── System.in (standard input)
    └── InputStreamReader (converts bytes to characters)
      └── BufferedReader (buffers characters for efficiency)

2. Scanner vs BufferedReader Comparison

Feature	Scanner	BufferedReader
Parsing	Built-in type parsing	Manual parsing required
Performance	Slower (regex-based)	Faster (direct reading)
Flexibility	High (multiple data types)	Limited (strings only)
Buffer Size	Small internal buffer	Large configurable buffer
Best Use	Interactive input, mixed types	Large text processing

3. Input Tokenization

- **Delimiter-based**: Scanner uses patterns to separate tokens
- **Line-based**: BufferedReader reads entire lines
- **Character-based**: Direct character reading possible

4. Memory and Performance Considerations

- **Scanner**: Uses regex engine, higher CPU usage
- **BufferedReader**: Minimal processing, better for large inputs

- **Buffer Management**: Reduces system calls

Using Scanner

```
import java.util.Scanner;

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

        // Read single word (stops at whitespace)
        System.out.print("Enter your first name: ");
        String firstName = scanner.next();

        // Read entire line (includes spaces)
        System.out.print("Enter your full name: ");
        scanner.nextLine(); // consume leftover newline character
        String fullName = scanner.nextLine();

        // Input validation example
        System.out.print("Enter your age: ");
        while (!scanner.hasNextInt()) {
            System.out.print("Please enter a valid number: ");
            scanner.next(); // consume invalid input
        }
        int age = scanner.nextInt();

        System.out.println("First name: " + firstName);
        System.out.println("Full name: " + fullName);
        System.out.println("Age: " + age);

        scanner.close(); // Important: prevents resource leak
    }
}
```

Scanner Token Parsing:

```
Scanner scanner = new Scanner("John,25,Engineer");
scanner.useDelimiter(","); // Custom delimiter
String name = scanner.next();     // "John"
```

```
int age = scanner.nextInt();      // 25
String job = scanner.next();      // "Engineer"
```

Using BufferedReader

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class BufferedReaderExample {
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));

        System.out.print("Enter a sentence: ");
        String sentence = reader.readLine();

        // Manual parsing for other data types
        System.out.print("Enter your age: ");
        String ageStr = reader.readLine();
        int age = Integer.parseInt(ageStr); // Manual conversion

        System.out.println("You entered: " + sentence);
        System.out.println("Your age: " + age);

        reader.close(); // Clean up resources
    }
}
```

Error Handling Best Practices:

```
try (Scanner scanner = new Scanner(System.in)) { // try-with-resources
    System.out.print("Enter text: ");
    String input = scanner.nextLine();

    if (input == null || input.trim().isEmpty()) {
        System.out.println("Invalid input!");
        return;
```

```
    }

    // Process valid input
    System.out.println("Processed: " + input.trim());
} // Scanner automatically closed
```

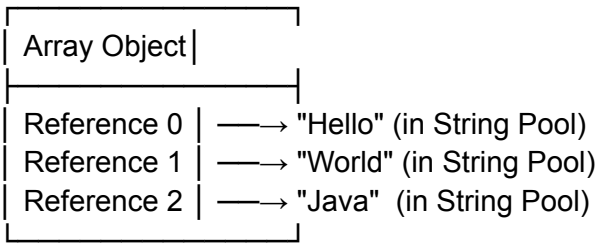
Part 5: String Arrays

Theoretical Concepts:

1. Array Memory Layout

```
String[] array = {"Hello", "World", "Java"};
```

Memory Structure:



2. Reference vs Value Semantics

- **Arrays store references**: Not the actual string data
- **Null references**: Array elements can be null
- **Shallow copying**: Array.clone() copies references, not string objects
- **Memory efficiency**: Multiple arrays can reference same strings

3. Array Initialization Patterns

```
// Static initialization - size determined by elements
String[] colors = {"Red", "Green", "Blue"};

// Dynamic initialization - specify size first
String[] fruits = new String[5];

// Anonymous array - useful for method parameters
```

```
printArray(new String[]{"A", "B", "C"});
```

4. Iteration Strategies

1. **Traditional for loop**: Index-based access, allows modification
2. **Enhanced for loop**: Read-only iteration, cleaner syntax
3. **Iterator pattern**: More flexible for different data structures

Creating and Initializing String Arrays

```
// Declaration and initialization
String[] colors = {"Red", "Green", "Blue"};

// Alternative syntax (less common but valid)
String colors2[] = {"Red", "Green", "Blue"};

// Dynamic allocation
String[] fruits = new String[3];
fruits[0] = "Apple";
fruits[1] = "Banana";
fruits[2] = "Orange";

// Array of different sizes
String[] names = new String[5]; // All elements initially null

// Multi-dimensional string arrays
String[][] matrix = {
    {"A1", "A2", "A3"},
    {"B1", "B2", "B3"}
};
```

Common String Array Operations

```
public class StringArrayDemo {
    public static void main(String[] args) {
        String[] languages = {"Java", "Python", "C++", "JavaScript"};
```

```
        // Length of array (property, not method)
        System.out.println("Array length: " + languages.length);

        // Traditional iteration with index access
        for (int i = 0; i < languages.length; i++) {
            System.out.println("Language " + (i+1) + ": " + languages[i]);
            // Can modify: languages[i] = languages[i].toUpperCase();
        }

        // Enhanced for loop (for-each) - read-only
        for (String lang : languages) {
            System.out.println("Programming language: " + lang);
            // Cannot modify array elements here
        }

        // Array bounds checking
        try {
            System.out.println(languages[10]); // Throws
ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out of bounds!");
        }
    }
}
```

Advanced Array Operations:

```
import java.util.Arrays;

// Array copying
String[] original = {"A", "B", "C"};
String[] copy = Arrays.copyOf(original, original.length);
String[] partial = Arrays.copyOfRange(original, 1, 3); // {"B", "C"}

// Array comparison
boolean equal = Arrays.equals(original, copy); // true

// Array sorting
```

```
String[] unsorted = {"Zebra", "Apple", "Banana"};
Arrays.sort(unsorted); // Modifies original array
// Result: {"Apple", "Banana", "Zebra"}

// Array searching (requires sorted array)
int index = Arrays.binarySearch(unsorted, "Banana"); // Returns index

// Convert array to string representation
String arrayString = Arrays.toString(languages);
System.out.println(arrayString); // [Java, Python, C++, JavaScript]
```

Part 6: Strings as Method Parameters

Theoretical Concepts:

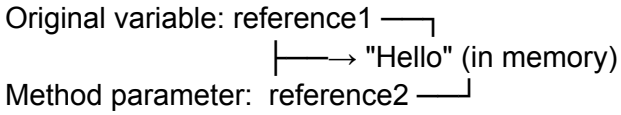
1. Pass-by-Value vs Pass-by-Reference

- **Java uses pass-by-value**: The reference value is copied, not the object
- **String immutability**: Prevents accidental modification of string objects
- **Reference copying**: Multiple variables can point to same string object

2. Method Parameter Mechanisms

```
void method(String str) {
    // 'str' is a copy of the reference
    // Points to same string object in memory
    // Cannot change what original variable points to
    // Can use the string object's methods
}
```

3. Memory Implications



4. String Pool and Method Parameters

- **Literal strings**: Passed references point to string pool

- **Constructor strings**: Passed references point to heap objects
- **Performance**: No string copying, only reference copying

```
public class StringMethods {

    // Method accepting string parameter
    public static void printGreeting(String name) {
        System.out.println("Hello, " + name + "!");
        // 'name' is local copy of reference - immutable
    }

    // Method returning modified string
    public static String formatName(String firstName, String lastName) {
        // String concatenation creates new object
        String result = lastName.toUpperCase() + ", " + firstName;
        return result; // Returns reference to new string
    }

    // Method with string array parameter
    public static void printAllNames(String[] names) {
        // 'names' is copy of array reference
        // Can modify array contents, but not original array variable
        for (int i = 0; i < names.length; i++) {
            if (names[i] != null) { // Null check important
                printGreeting(names[i]);
            }
        }
    }

    // Method demonstrating string immutability
    public static void tryToModify(String str) {
        System.out.println("Original parameter: " + str);
        str = "Modified"; // Creates new string, doesn't affect original
        System.out.println("Inside method: " + str);
        // Original variable outside method unchanged
    }

    // Method with multiple string parameters
    public static String buildAddress(String street, String city, String
state, String zip) {
```

```
        // StringBuilder more efficient for multiple concatenations
        StringBuilder address = new StringBuilder();
        address.append(street).append(", ")
               .append(city).append(", ")
               .append(state).append(" ")
               .append(zip);
        return address.toString();
    }

    // Method demonstrating null handling
    public static String safeConcat(String str1, String str2) {
        // Defensive programming - handle null inputs
        if (str1 == null) str1 = "";
        if (str2 == null) str2 = "";
        return str1 + str2;
    }

    // Varargs method - variable number of string parameters
    public static String joinStrings(String delimiter, String... strings) {
        if (strings.length == 0) return "";

        StringBuilder result = new StringBuilder(strings[0]);
        for (int i = 1; i < strings.length; i++) {
            result.append(delimiter).append(strings[i]);
        }
        return result.toString();
    }

    public static void main(String[] args) {
        // Using methods with string parameters
        printGreeting("Alice");

        String formatted = formatName("John", "Doe");
        System.out.println("Formatted: " + formatted);

        String[] team = {"Alice", "Bob", "Charlie"};
        printAllNames(team);

        // Demonstrating immutability
        String original = "Original";
```

```
        System.out.println("Before method call: " + original);
        tryToModify(original);
        System.out.println("After method call: " + original); // Still
"Original"

        // Building complex strings
        String address = buildAddress("123 Main St", "Anytown", "CA",
"12345");
        System.out.println("Address: " + address);

        // Safe concatenation
        String result = safeConcat("Hello", null); // Handles null
gracefully
        System.out.println("Safe concat: '" + result + "'");

        // Varargs example
        String joined = joinStrings(" - ", "Java", "Python", "JavaScript");
        System.out.println("Joined: " + joined);
    }
}

Method Overloading with Strings:
public class StringProcessor {
    // Method overloading - same name, different parameters

    public static String process(String input) {
        return input.trim().toLowerCase();
    }

    public static String process(String input, boolean uppercase) {
        String processed = input.trim();
        return uppercase ? processed.toUpperCase() :
processed.toLowerCase();
    }

    public static String process(String input, String prefix, String
suffix) {
        return prefix + input.trim() + suffix;
    }
}
```

Part 7: Java Exceptions

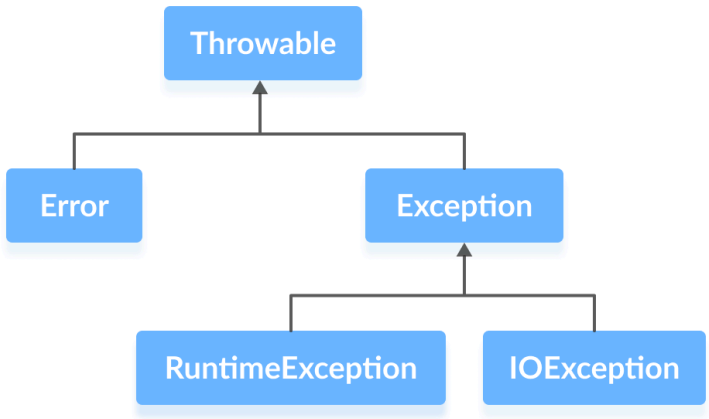
An exception is an unexpected event that occurs during program execution. It affects the flow of the program instructions which can cause the program to terminate abnormally. An exception can occur for many reasons. Some of them are:

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Accessing unavailable Memory
- Opening an unavailable file

1. **Java Exception Hierarchy -** Here is a simplified diagram of the exception hierarchy in Java. Java ***Throwable*** class is the root class in the hierarchy. It Splits into ***Error*** and ***Exception***

    a. ***java.lang.Error*** - ***Error*** represents irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.
    b. ***java.lang.Exception*** - Exceptions can be caught and handled by the program.

    I. When an exception occurs, it creates an object. This object is called the exception object.
    Ii. It contains information about the exception such as the name and description of the exception and the state of the program when the exception occurred.



2. Java Exception Types - The Exception class is further divided into ***RuntimeException*** which are Unchecked Exceptions and Checked Exceptions like ***IOException***.

a. ***java.lang.RuntimeException*****:** These exceptions are raised during runtime due to programming errors. They are not checked at compile time and hence are also called **unchecked exceptions**. Some examples are
I. Improper use of an API - ***IllegalArgumentException***
ii. Null pointer access (missing the initialization of a variable) - ***NullPointerException***
iii. Out-of-bounds array access - ***ArrayIndexOutOfBoundsException***
Iv. Dividing a number by 0 - ***ArithmeticException***

Example for unchecked exceptions. It is not compulsory to catch unchecked exceptions while writing code. Hence the following code will compile. However, when you run the code you will encounter runtime exceptions.

```
class Main {
   public static void main(String[] args) {
      // Following code generate Runtime exception but not at compile time
      int divideByZero = 5 / 0;
   }
}
```

Refactor the code to handle the RuntimeException using the try-catch block. It is a good programming practice to write the code with a try-catch. You can catch specific ***ArithmeticException*** or generic ***Exception*** or both. All will work.

```
// Program demonstrates handling of Exceptions.
class Main {
   public static void main(String[] args) {
     // Following code generate Runtime exception which is
ArithmeticException
     try {
        int divideByZero = 5 / 0;
     } catch (ArithmeticException e) {
        System.out.println("ArithmeticException => " + e.getMessage());
     } catch (Exception e) {
        System.out.println("Generic Exception => " + e.getMessage());
     }
  }
}
```

b. **Checked Exceptions:** These are exceptions that are checked by the Compiler during compile-time and need to be handled by the programmer using the try...catch block. ***java.io.IOException*** is regularly used during the handling of File Operations. Also All User-Defined Exceptions are Checked Exceptions.

   The following example is about reading the contents from a File using the File and the Scanner class. In this case, it is mandatory to handle FileNotFoundException by catching it with ***java.io.FileNotFoundException*** or with ***java.io.IOException***. Please note both FileNotFoundException and IOException cannot be used together in the catch block as anyone has already handled the file not found exception.

It's a good programming practice to catch the Checked Exception as well as generic ***Exception*** to handle other runtime exceptions.

```
// Following Program demonstrates handling FileNotFoundException
class Main {
   public static void main(String[] args) {
      try {
         java.io.File file = new java.io.File("file.txt");
         java.util.Scanner sc = new java.util.Scanner(file);
      } catch (java.io.FileNotFoundException e) {
         System.out.println("FileNotFoundException occurred");
      }  catch (Exception e) {
         System.out.println("Exception occurred");
      }
   }
}
// Following Program demonstrates handling IOException. Exception Handling is
// achieved using FileNotFoundException or IOException. Other runtime
// exceptions is handled with Exception
class Main {
   public static void main(String[] args) {
      try {
         java.io.File file = new java.io.File("file.txt");
         java.util.Scanner sc = new java.util.Scanner(file);
      } catch (java.io.IOException e) {
         System.out.println("IOException occurred");
      }  catch (Exception e) {
         System.out.println("Exception occurred");
      }
   }
}
```

Best Practices:

- Use string literals for better memory efficiency
- Remember that strings are immutable
- Use StringBuilder for multiple concatenations
- Always close Scanner/BufferedReader resources
- Validate input before processing
- Use appropriate string methods for common operations

Next Steps:

- Explore String class methods (substring, indexOf, replace, etc.)
- Learn about StringBuilder and StringBuffer in detail
- Practice with regular expressions
- Study string formatting with printf and String.format()