**StringBuffer in Java**

**Introduction**

In Java, **StringBuffer** is a class that provides a mutable (modifiable) sequence of characters. Unlike the **String** class, which creates immutable objects, **StringBuffer** allows modifications without creating new objects, making it more memory-efficient for operations involving frequent string manipulation.

**Key Features of StringBuffer**

1. **Mutable**: Allows modification of the string without creating a new object.
2. **Thread-Safe**: Synchronized, making it safe for multi-threaded environments.
3. **Performance**: Faster than String when multiple modifications are performed.

**Creating a StringBuffer Object**

**Syntax:**

StringBuffer sb = new StringBuffer("Hello");

**Example:**

public class StringBufferExample {

public static void main(String[] args) {

StringBuffer sb = new StringBuffer("Hello");

System.out.println("Original StringBuffer: " + sb);

}

}

**Output:**

Original StringBuffer: Hello

**Important Methods of StringBuffer**

**1. append() - Adds text at the end**

public class AppendExample {

public static void main(String[] args) {

StringBuffer sb = new StringBuffer("Hello");

sb.append(" World"); // Adds " World" to the existing string

System.out.println(sb);

}

}

**Output:**

Hello World

**2. insert() - Inserts text at a specific index**

public class InsertExample {

public static void main(String[] args) {

StringBuffer sb = new StringBuffer("Java");

sb.insert(2, "Programming"); // Inserts "Programming" at index 2

System.out.println(sb);

}

}

**Output:**

JaProgrammingva

**3. replace() - Replaces a part of the string**

public class ReplaceExample {

public static void main(String[] args) {

StringBuffer sb = new StringBuffer("Hello Java");

sb.replace(6, 10, "World"); // Replaces "Java" with "World"

System.out.println(sb);

}

}

**Output:**

Hello World

**4. delete() - Deletes characters from a specific range**

public class DeleteExample {

public static void main(String[] args) {

StringBuffer sb = new StringBuffer("Hello World");

sb.delete(5, 11); // Removes " World"

System.out.println(sb);

}

}

**Output:**

Hello

**5. reverse() - Reverses the string**

public class ReverseExample {

public static void main(String[] args) {

StringBuffer sb = new StringBuffer("Java");

sb.reverse();

System.out.println(sb);

}

}

**Output:**

avaJ

**6. capacity() - Returns the current capacity of the buffer**

public class CapacityExample {

public static void main(String[] args) {

StringBuffer sb = new StringBuffer();

System.out.println("Initial Capacity: " + sb.capacity()); // Default capacity is 16

sb.append("Hello World");

System.out.println("Capacity after append: " + sb.capacity());

sb.append("Java is great!");

System.out.println("New Capacity: " + sb.capacity()); // Grows dynamically

}

}

**Output:**

Initial Capacity: 16

Capacity after append: 16

New Capacity: 34

**7. ensureCapacity() - Ensures a minimum capacity**

public class EnsureCapacityExample {

public static void main(String[] args) {

StringBuffer sb = new StringBuffer();

System.out.println("Initial Capacity: " + sb.capacity());

sb.ensureCapacity(50); // Ensures capacity is at least 50

System.out.println("New Capacity: " + sb.capacity());

}

}

**Output:**

Initial Capacity: 16

New Capacity: 50

**Difference Between String and StringBuffer**

| **Feature** | **String** | **StringBuffer** |
| --- | --- | --- |
| **Mutability** | Immutable (creates a new object on modification) | Mutable (modifies the same object) |
| **Performance** | Slower for multiple modifications | Faster for multiple modifications |
| **Thread-Safety** | Thread-safe but creates a lot of objects | Thread-safe (synchronized methods) |
| **Memory Usage** | Higher (due to new object creation) | Lower (modifies existing object) |

**Conclusion**

* StringBuffer is useful when frequent modifications are needed.
* It provides better performance than String in such cases.
* Use String when immutability is required and StringBuffer when efficient modifications are needed.

**UNIT -5**

**Networking in Java**

**1. Networking Basics**

Networking allows Java applications to communicate over the internet or a local network. Java provides built-in support for networking through the **java.net** package.

Key networking concepts:

* **IP Address**: A unique identifier for a device on a network.
* **Port Number**: Identifies a specific process running on a machine.
* **Protocol**: Rules that define how data is transmitted (e.g., HTTP, FTP, TCP, UDP).
* **Socket**: Endpoints for sending and receiving data.

**2. Java and the Internet Address**

Java provides the **InetAddress** class to handle IP addresses.

**Example:**

import java.net.\*;

public class IPAddressExample {

public static void main(String[] args) {

try {

InetAddress ip = InetAddress.getByName("www.google.com");

System.out.println("IP Address: " + ip.getHostAddress());

System.out.println("Host Name: " + ip.getHostName());

} catch (UnknownHostException e) {

System.out.println("Host not found: " + e.getMessage());

}

}

}

**3. TCP/IP Client Sockets**

A **TCP/IP socket** allows two applications to communicate using the **Transmission Control Protocol (TCP)**.

**Client Example:**

import java.io.\*;

import java.net.\*;

public class ClientSocketExample {

public static void main(String[] args) {

try {

Socket socket = new Socket("localhost", 5000);

DataOutputStream dout = new DataOutputStream(socket.getOutputStream());

dout.writeUTF("Hello Server!");

dout.flush();

dout.close();

socket.close();

} catch (IOException e) {

System.out.println(e);

}

}

}

**4. TCP/IP Server Sockets**

The **ServerSocket** class listens for client requests and establishes a connection.

**Server Example:**

import java.io.\*;

import java.net.\*;

public class ServerSocketExample {

public static void main(String[] args) {

try {

ServerSocket server = new ServerSocket(5000);

System.out.println("Server is waiting for a client...");

Socket socket = server.accept();

DataInputStream din = new DataInputStream(socket.getInputStream());

String message = din.readUTF();

System.out.println("Client says: " + message);

din.close();

socket.close();

server.close();

} catch (IOException e) {

System.out.println(e);

}

}

}

**5. URL and URLConnection in Java**

The **URL** class represents a web address, and **URLConnection** allows Java to interact with web resources.

**Example: Fetching Data from a URL**

import java.net.\*;

import java.io.\*;

public class URLExample {

public static void main(String[] args) {

try {

URL url = new URL("https://www.example.com");

BufferedReader br = new BufferedReader(new InputStreamReader(url.openStream()));

String line;

while ((line = br.readLine()) != null) {

System.out.println(line);

}

br.close();

} catch (Exception e) {

System.out.println(e);

}

}

}

**The Applet Class in Java**

**1. Introduction to Applets**

Applets are small Java programs that run inside a web browser. Java Applets extend the Applet class from java.applet package.

**2. Applet Architecture**

An Applet typically has the following lifecycle methods:

1. **init()** – Initializes the applet.
2. **start()** – Starts the applet.
3. **paint(Graphics g)** – Used to draw on the applet.
4. **stop()** – Stops the applet.
5. **destroy()** – Cleans up before applet is removed.

**3. Display Methods in Applets**

The paint() method is used to draw on the applet.

**Example Applet:**

import java.applet.\*;

import java.awt.\*;

public class SimpleApplet extends Applet {

public void paint(Graphics g) {

g.drawString("Hello, this is a Java Applet!", 20, 20);

}

}

**HTML File:**

<html>

<body>

<applet code="SimpleApplet.class" width="300" height="200"></applet>

</body>

</html>

**4. Passing Parameters to Applet**

You can pass parameters from the HTML file to the Applet using <param> tags.

**Example Applet with Parameters:**

import java.applet.\*;

import java.awt.\*;

public class ParamApplet extends Applet {

String message;

public void init() {

message = getParameter("msg");

if (message == null)

message = "Default Message";

}

public void paint(Graphics g) {

g.drawString(message, 20, 20);

}

}

**HTML File:**

<html>

<body>

<applet code="ParamApplet.class" width="300" height="200">

<param name="msg" value="Hello from HTML!">

</applet>

</body>

</html>

**5. getDocumentBase() and getCodeBase() Methods**

* **getDocumentBase()**: Returns the URL of the document containing the Applet.
* **getCodeBase()**: Returns the URL of the Applet’s code.

**Example:**

import java.applet.\*;

import java.awt.\*;

public class BaseExample extends Applet {

public void paint(Graphics g) {

g.drawString("Document Base: " + getDocumentBase(), 20, 20);

g.drawString("Code Base: " + getCodeBase(), 20, 40);

}

}

**6. AppletContext and showDocument()**

The **AppletContext** interface allows applets to interact with the browser, such as opening a new webpage.

**Example: Opening a URL from an Applet**

import java.applet.\*;

import java.awt.\*;

import java.net.\*;

public class OpenURLApplet extends Applet {

public void start() {

try {

AppletContext context = getAppletContext();

URL url = new URL("https://www.google.com");

context.showDocument(url, "\_blank"); // Opens in a new tab

} catch (Exception e) {

System.out.println(e);

}

}

}