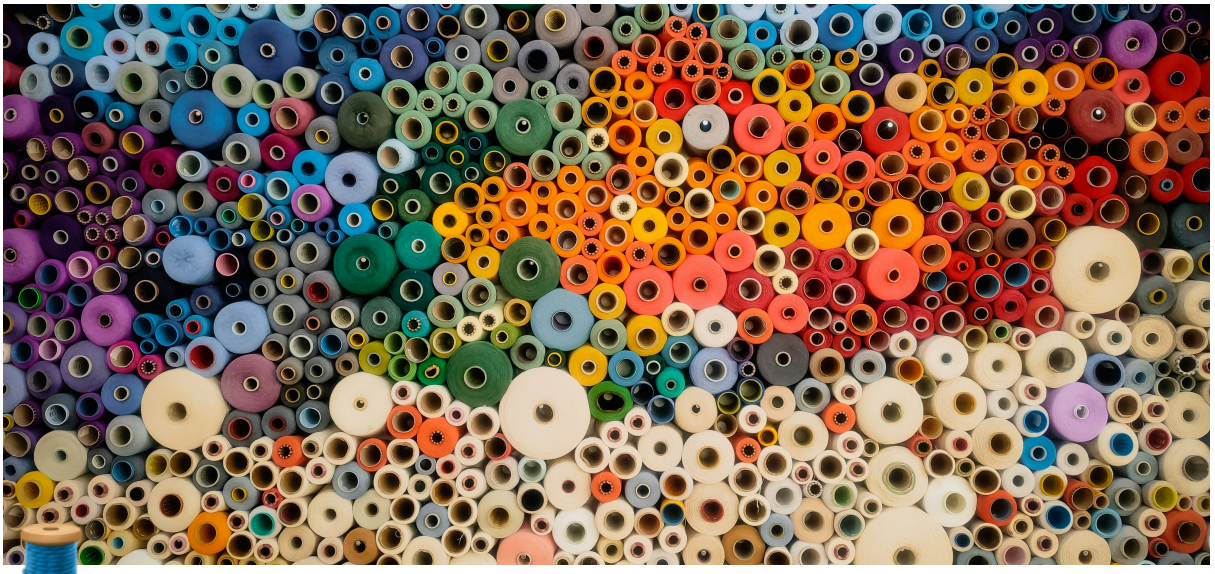

Multithreading DAY -7

Introduction to Multithreading

What is Multithreading?

- **Multithreading** is the ability of a program to execute multiple threads (smaller units of a process) simultaneously.
- It allows efficient utilization of CPU by running multiple tasks in parallel.

Why Use Multithreading?

- **Better CPU Utilization:** Runs multiple tasks at once.
- **Faster Execution:** Tasks execute independently without waiting for each other.
- **Non-blocking Operations:** Useful in GUI applications where the main thread remains responsive.
- **Concurrency Handling:** Helps manage multiple user requests (important in web applications, servers, etc.).

---

2. Understanding Threads

What is a Thread?

- A **Thread** is the smallest unit of execution in a program.
- Java provides built-in support for threads using the `Thread` class and `Runnable` interface.

Single vs Multi-threading

Single-threading	Multi-threading

One task runs at a time	Multiple tasks run simultaneously
Slower execution	Faster execution
No parallel execution	Parallel execution using multiple threads

3. Creating Threads in Java

There are **two ways** to create a thread in Java:

1. **Extending the `Thread` class**
2. **Implementing the `Runnable` interface**

4. Creating a Thread by Extending `Thread` Class

- The `Thread` class in Java provides built-in methods to create and manage threads.
- We override the `run()` method to define the task for the thread.

Example:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running: " + Thread.currentThread().get
Id());
    }

    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start(); // Starts a new thread
    }
}
```

Key Methods of `Thread` Class:

Method	Description
`start()`	Starts the thread (calls `run()` internally)
`run()`	Defines the task to be performed
`sleep(ms)`	Puts thread to sleep for a given time (milliseconds)
`join()`	Waits for a thread to finish execution
`getId()`	Returns the thread ID
`setName(name)`	Assigns a name to a thread
`getName()`	Returns the thread's name
`isAlive()`	Checks if a thread is still running

5. Creating a Thread by Implementing `Runnable` Interface

- More flexible than extending `Thread` because Java doesn't support multiple inheritance.
- Useful when a class already extends another class.

Example:

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread running: " + Thread.currentThread().getId
());
    }

    public static void main(String[] args) {
        MyRunnable runnable = new MyRunnable();
        Thread t1 = new Thread(runnable);
        t1.start();
    }
}
```

Difference Between `Thread` and `Runnable`

Feature	`Thread` Class	`Runnable` Interface
Inheritance	Extends `Thread`	Implements `Runnable`
Flexibility	Less flexible (since Java doesn't support multiple inheritance)	More flexible (can implement multiple interfaces)
Memory Usage	More memory	Less memory
Recommended For	Simpler thread creation	When class needs to extend another class

6. Thread Lifecycle (States of a Thread)

A thread goes through **five states** during execution:

1. **New (Created):** Thread is created but not started yet (`new Thread()`).
2. **Runnable (Ready to run):** Thread is ready but waiting for CPU.
3. **Running (Executing):** Thread is currently running.
4. **Blocked/Waiting (Paused):** Thread is temporarily paused (e.g., waiting for I/O).
5. **Terminated (Dead):** Thread has completed execution.

Example of Lifecycle:

```
class LifecycleExample extends Thread {
    public void run() {
        System.out.println("Thread is running...");
    }

    public static void main(String args[]) {
        LifecycleExample t1 = new LifecycleExample();
        System.out.println("Thread Created (New State)");
        t1.start();
        System.out.println("Thread Started (Runnable State)");
```

```
    }
}
```

7. Thread Priorities

Threads have priority values ranging from `1` (MIN) to `10` (MAX).

- Default priority is `5` (NORM).
- We can change priority using `setPriority(int value)`.

Example:

```
class PriorityExample extends Thread {
    public void run() {
        System.out.println("Running thread priority: " + this.getPriority());
    }

    public static void main(String[] args) {
        PriorityExample t1 = new PriorityExample();
        PriorityExample t2 = new PriorityExample();

        t1.setPriority(Thread.MIN_PRIORITY); // 1
        t2.setPriority(Thread.MAX_PRIORITY); // 10

        t1.start();
        t2.start();
    }
}
```

8. Synchronization (Thread Safety)

- When multiple threads try to access a shared resource, **race conditions** can occur.
- **Synchronization** ensures that only one thread can access the critical section at a time.

Using `synchronized` Keyword

```
class BankAccount {
    private int balance = 1000;

    public synchronized void withdraw(int amount) {
        if (balance >= amount) {
            balance -= amount;
            System.out.println("Withdrawal successful! New balance: " + balan
ce);
        } else {
            System.out.println("Insufficient funds.");
        }
    }
```

```
}

class Customer extends Thread {
    BankAccount account;

    Customer(BankAccount acc) {
        this.account = acc;
    }

    public void run() {
        account.withdraw(700);
    }

    public static void main(String args[]) {
        BankAccount account = new BankAccount();
        Customer c1 = new Customer(account);
        Customer c2 = new Customer(account);

        c1.start();
        c2.start();
    }
}
```

9. Inter-Thread Communication

- Threads can **communicate** with each other using `wait()`, `notify()`, and `notifyAll()`.
- Used in producer-consumer problems.

Example of `wait()` and `notify()`:

```
class Communication {
    synchronized void printMessage() {
        try {
            wait();  // Waits until notified
            System.out.println("Thread resumed...");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    synchronized void resumeThread() {
        notify();  // Notifies the waiting thread
    }
}

class ThreadA extends Thread {
    Communication comm;
    ThreadA(Communication c) {
```

```
        this.comm = c;
    }
    public void run() {
        comm.printMessage();
    }
}

class ThreadB extends Thread {
    Communication comm;
    ThreadB(Communication c) {
        this.comm = c;
    }
    public void run() {
        comm.resumeThread();
    }
}

public class ThreadCommExample {
    public static void main(String args[]) {
        Communication c = new Communication();
        ThreadA t1 = new ThreadA(c);
        ThreadB t2 = new ThreadB(c);

        t1.start();
        t2.start();
    }
}
```

```
public class ThreadExtends extends Thread {
    public void run() {
        Thread te = new Thread(() -> {
            try {
                Thread.sleep(1000);
                System.out.println("Child thread executing...");
            } catch (InterruptedException e) {
                System.out.println("Child thread interrupted");
            }
        });

        System.out.println("Before");
        System.out.println("State before start: " + te.getState()); // NEW
        te.start();
        System.out.println("State after start: " + te.getState()); // RUNNABLE

        try {
            Thread.sleep(500);
            System.out.println("State during sleep: " + te.getState()); // RUNN
```

```
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted");
        }

        try {
            te.join(); // Ensures this thread waits for te to complete
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("After join, state: " + te.getState()); // TERMINATE
    }

    public static void main(String[] args) {
        ThreadExtends t = new ThreadExtends();
        t.start();
    }
}
```

1️⃣ Thread Lifecycle (Thread States)

The program prints different states of a thread using `getState()`, covering:

- **NEW** → Before `start()` is called.
- **RUNNABLE** → After `start()` is called, ready to run.
- **TIMED_WAITING** → When `Thread.sleep()` is called.
- **TERMINATED** → After the thread completes execution.

💡 **Key Concept: Thread Lifecycle (`Thread.State`)**

```
System.out.println("Runnable " + te.getState());
System.out.println("Thread " + te.getState());
System.out.println("After join " + te.getState());
```

2️⃣ Thread Creation

The code creates and starts a thread using:

```
Thread te = new Thread();
te.start();
```

💡 **Key Concept: Creating Threads**

However, since `te` has no `run()` method, it starts and **immediately terminates**.

3️⃣ Thread Sleep (`sleep()`)

```
Thread.sleep(2000);
```

This makes the current thread pause for **2 seconds**, simulating a delay.

💡 **Key Concept: Pausing a Thread (`Thread.sleep()`)**

4️⃣ Thread Synchronization Using `join()`

```
try {
    te.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

- `join()` ensures that the **current thread waits** until `te` finishes execution.
- However, in your original code, `te` finishes instantly, making `join()` ineffective.

💡 **Key Concept: Thread Coordination Using `join()`**

📝 Summary of Concepts in Your Code

✅ **Thread Lifecycle (`getState()`)**

✅ **Creating Threads (`Thread.start()`)**

✅ **Pausing a Thread (`Thread.sleep()`)**

✅ **Thread Synchronization (`Thread.join()`)**

Understanding Threads in Java: `start()`, `run()`, and `join()`

Introduction to Threads in Java

- A **Thread** in Java is a lightweight process that enables parallel execution of tasks.
- Java provides **multithreading** to efficiently utilize CPU resources.
- Threads can be created in two ways:
  1. **Extending the `Thread` class**
  2. **Implementing the `Runnable` interface**

Creating and Running Threads

1. Using the `Thread` Class

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            try {
                Thread.sleep(1000); // Pause for 1 second
                System.out.println(i);
            } catch (InterruptedException e) {
                System.out.println(e);
```

```
            }
        }
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start(); // Starts a new thread
    }
}
```

2. Using the `Runnable` Interface

```
class MyRunnable implements Runnable {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            try {
                Thread.sleep(1000);
                System.out.println(i);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

public class RunnableExample {
    public static void main(String[] args) {
        MyRunnable r = new MyRunnable();
        Thread t1 = new Thread(r);
        t1.start();
    }
}
```

`start()` vs `run()`

Feature	`start()`	`run()`
Thread Creation	Starts a new thread	Runs in the current thread
Parallel Execution	Yes, executes concurrently	No, executes sequentially
Calls `run()` Method	Yes	Yes, but in the main thread

Example to Demonstrate Difference

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
```

```
            try {
                Thread.sleep(1000);
                System.out.println(i);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

public class StartVsRunExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.run(); // Runs in main thread (sequential)
        t2.run(); // Runs in main thread (sequential)

        // Uncomment below lines to see actual multithreading
        // t1.start(); // Runs in a separate thread
        // t2.start(); // Runs in a separate thread
    }
}
```

`join()` Method

Purpose of `join()`

- Makes the **current thread wait** for another thread to finish execution before continuing.
- Useful for **synchronizing** thread execution.
- Ensures a thread completes before proceeding to the next step.

Example Without `join()`

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            try {
                Thread.sleep(1000);
                System.out.println(i);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

public class WithoutJoinExample {
```

```
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.start(); // Runs concurrently
        t2.start(); // Runs concurrently

        System.out.println("Main thread continues without waiting.");
    }
}
```

Example With `join()`

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            try {
                Thread.sleep(1000);
                System.out.println(i);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

public class WithJoinExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.start();
        try {
            t1.join(); // Main thread waits for t1 to finish
        } catch (InterruptedException e) {
            System.out.println(e);
        }

        t2.start();
        try {
            t2.join(); // Main thread waits for t2 to finish
        } catch (InterruptedException e) {
            System.out.println(e);
        }

        System.out.println("Both threads finished. Main thread resumes.");
    }
}
```

Key Observations with `join()`

- **Without** `join()`, the main thread doesn't wait for `t1` and `t2` to finish.
- **With** `join()`, the main thread waits for `t1` first, then `t2`, ensuring sequential execution.

Summary

Feature	Description
`start()`	Begins execution of a new thread concurrently.
`run()`	Runs in the same thread as the caller, without starting a new thread.
`join()`	Makes the calling thread wait until the specified thread finishes execution.

Real-World Analogy

- Imagine two runners in a race 🏃‍♂️🏃‍♂️.
- Both start running at the **same time** (`start()` is called).
- If one reaches the finish line first, they have to **wait** for the second runner (`join()` is used).
- Only when both finish, they can move to the next race (execution continues).

Final Thought 💡

- **Use** `start()` **for multithreading** to run tasks in parallel.
- **Avoid calling** `run()` **directly** unless sequential execution is required.
- **Use** `join()` **when one thread depends on another** and must wait for it to complete.

Concept of Synchronization in Java

What is Synchronization?

Synchronization in Java is a technique that ensures that multiple threads do not access shared resources (like variables, files, or methods) at the same time in a way that causes inconsistencies or unexpected results.

Think of synchronization like a traffic signal at an intersection: it ensures that cars (threads) take turns, preventing accidents (race conditions).

Why is Synchronization Needed?

- When multiple threads access shared data simultaneously, it can lead to **race conditions**, where the output is unpredictable.
- Synchronization ensures **thread safety** by allowing only one thread to access the critical section (shared resource) at a time.
- Prevents **data inconsistency** and **unexpected behavior**.

Types of Synchronization in Java

1. **Method Synchronization** (Synchronized Methods)
2. **Block Synchronization** (Synchronized Blocks)

3. **Static Synchronization** (Synchronizing Static Methods)

Step-by-Step Code Implementation

1️⃣ Without Synchronization (Race Condition Example)

Let's first see what happens when multiple threads access the same resource without synchronization.

```
class SharedResource {
    void printNumbers(int n) {
        for (int i = 1; i <= 5; i++) {
            System.out.println(n * i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

class MyThread1 extends Thread {
    SharedResource obj;
    MyThread1(SharedResource obj) {
        this.obj = obj;
    }
    public void run() {
        obj.printNumbers(2);
    }
}

class MyThread2 extends Thread {
    SharedResource obj;
    MyThread2(SharedResource obj) {
        this.obj = obj;
    }
    public void run() {
        obj.printNumbers(3);
    }
}

public class WithoutSynchronization {
    public static void main(String[] args) {
        SharedResource obj = new SharedResource();
        MyThread1 t1 = new MyThread1(obj);
        MyThread2 t2 = new MyThread2(obj);

        t1.start();
```

```
        t2.start();
    }
}
```

Output (May vary due to race condition):

```
2
3
4
6
6
8
10
12
15
```

💡 **Problem**: The outputs of both threads mix up because both access `printNumbers()` simultaneously.

2️⃣ Using Synchronized Method (Thread Safety)

We solve the issue by synchronizing the method so only one thread can execute it at a time.

```
class SharedResource {
    synchronized void printNumbers(int n) {  // Synchronized method
        for (int i = 1; i <= 5; i++) {
            System.out.println(n * i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

class MyThread1 extends Thread {
    SharedResource obj;
    MyThread1(SharedResource obj) {
        this.obj = obj;
    }
    public void run() {
        obj.printNumbers(2);
    }
}

class MyThread2 extends Thread {
    SharedResource obj;
    MyThread2(SharedResource obj) {
        this.obj = obj;
    }
```

```
    public void run() {
        obj.printNumbers(3);
    }
}

public class SynchronizedMethod {
    public static void main(String[] args) {
        SharedResource obj = new SharedResource();
        MyThread1 t1 = new MyThread1(obj);
        MyThread2 t2 = new MyThread2(obj);

        t1.start();
        t2.start();
    }
}
```

Output (Thread-safe execution):

```
2
4
6
8
10
3
6
9
12
15
```

🔷 **Now, one thread completes execution before another starts.**

3️⃣ Using Synchronized Block

Instead of synchronizing the whole method, we can synchronize only the critical section.

```
class SharedResource {
    void printNumbers(int n) {
        synchronized (this) {  // Synchronized block
            for (int i = 1; i <= 5; i++) {
                System.out.println(n * i);
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    System.out.println(e);
                }
            }
        }
    }
}
```

```
class MyThread1 extends Thread {
    SharedResource obj;
    MyThread1(SharedResource obj) {
        this.obj = obj;
    }
    public void run() {
        obj.printNumbers(2);
    }
}

class MyThread2 extends Thread {
    SharedResource obj;
    MyThread2(SharedResource obj) {
        this.obj = obj;
    }
    public void run() {
        obj.printNumbers(3);
    }
}

public class SynchronizedBlock {
    public static void main(String[] args) {
        SharedResource obj = new SharedResource();
        MyThread1 t1 = new MyThread1(obj);
        MyThread2 t2 = new MyThread2(obj);

        t1.start();
        t2.start();
    }
}
```

Same Output as the synchronized method:

🔷 Synchronizing only the required part increases performance.

4️⃣ Static Synchronization (For Static Methods)

If multiple threads access a static method of a class, we can synchronize the static method.

```
class SharedResource {
    synchronized static void printNumbers(int n) {
        for (int i = 1; i <= 5; i++) {
            System.out.println(n * i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
```

```
}

class MyThread1 extends Thread {
    public void run() {
        SharedResource.printNumbers(2);
    }
}

class MyThread2 extends Thread {
    public void run() {
        SharedResource.printNumbers(3);
    }
}

public class StaticSynchronization {
    public static void main(String[] args) {
        MyThread1 t1 = new MyThread1();
        MyThread2 t2 = new MyThread2();

        t1.start();
        t2.start();
    }
}
```

🔷 **Ensures thread safety at a class level** (All instances use the same lock).

Key Takeaways

Feature	Without Synchronization	Synchronized Method	Synchronized Block	Static Synchronization
Thread Safety	❌ Not Safe	✅ Safe	✅ Safe	✅ Safe
Performance	⚡ Fast but risky	🐢 Slow	⚡ Faster than method	🐢 Slow
Lock Scope	No lock	Entire method	Block of code	Entire class

💡 **Final Summary:**

- Synchronization ensures **thread safety** when multiple threads access shared resources.
- Use **synchronized methods** when you need to lock an entire function.
- Use **synchronized blocks** to improve performance by locking only necessary code.
- Use **static synchronization** for static methods shared by multiple threads.

Synchronization in Java with `wait()` & `notify()`

📌 Concept

Synchronization ensures **only one thread can access a shared resource at a time** to prevent data inconsistency.

- `wait()` → Makes a thread **pause** until another thread notifies it.
- `notify()` → Wakes up a **waiting thread** when a condition is met.

🔷 Simple Example

This example simulates a **bank account** where:

- One thread **withdraws money** (if not enough balance, it waits).
- Another thread **deposits money** and then **notifies** the waiting thread.

```
class BankAccount {
    private int balance = 1000; // Initial balance

    // Withdraw method (only one thread can use at a time)
    synchronized void withdraw(int amount) {
        System.out.println("Trying to withdraw: " + amount);
        while (balance < amount) {  // Use while to prevent spurious wakeups
            System.out.println("Not enough balance, waiting for deposit...");
            try {
                wait();  // Wait until deposit happens
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
        balance -= amount;
        System.out.println("Withdrawal successful! New balance: " + balance);
    }

    // Deposit method
    synchronized void deposit(int amount) {
        System.out.println("Depositing: " + amount);
        balance += amount;
        System.out.println("Deposit successful! New balance: " + balance);
        notify(); // Notify waiting thread (withdraw)
    }
}

public class SyncNotifyExample {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();

        // Withdraw thread
        new Thread(() -> account.withdraw(2000)).start();

        // Deposit thread (runs after a delay)
        new Thread(() -> {
            try { Thread.sleep(2000); } catch (Exception e) {}
            account.deposit(3000);
        }).start();
```

```
    }
}
```

🔷 How `wait()` & `notify()` Work Here?

Step	Explanation
1️⃣ **Withdraw starts**	Tries to withdraw **₹2000** but balance is **₹1000**.
2️⃣ `wait()`	Thread **waits** for money to be deposited.
3️⃣ **Deposit starts**	Deposits **₹3000**, balance becomes **₹4000**.
4️⃣ `notify()`	Wakes up the **waiting thread** (withdraw).
5️⃣ **Withdraw completes**	Now balance is **₹2000** after withdrawal.

🔷 Expected Output

```
Trying to withdraw: 2000
Not enough balance, waiting for deposit...
Depositing: 3000
Deposit successful! New balance: 4000
Withdrawal successful! New balance: 2000
```

🔷 Key Takeaways

✅ `synchronized` → Ensures only **one thread** can access the method at a time.

✅ `wait()` → Makes the thread **pause** until it gets notified.

✅ `notify()` → **Wakes up the waiting thread** after deposit is done.

✅ `while (condition)` → Prevents **spurious wakeups** (unexpected notifications).

🔷 Summary

🔷 Synchronization **prevents race conditions** and ensures **data consistency**.

🔷 `wait()` makes a thread pause, while `notify()` allows it to continue.

Wrapper Classes

1. Primitive Data Types in Java

These are the basic data types in Java that store simple values. They are **not objects**, and they directly hold the value.

- **Examples**: `int`, `char`, `double`, `boolean`, etc.

2. Wrapper Classes

Wrapper classes in Java are used to wrap or "box" primitive types into objects. Java provides a wrapper class for each primitive type, allowing you to treat primitive values as objects.

- **Why use wrapper classes?**
  - You need objects, not just primitive values (e.g., when working with collections like `ArrayList`).

- Provides utility methods for converting strings, parsing, and manipulating values.

- **Common Wrapper Classes**:
  - `Integer` for `int`
  - `Character` for `char`
  - `Double` for `double`
  - `Boolean` for `boolean`

Example:

```
int primitiveInt = 10;   // primitive
Integer wrappedInteger = Integer.valueOf(primitiveInt);  // wrapped as Intege
r object
```

3. Autoboxing and Unboxing

Java automatically converts between primitive types and their corresponding wrapper class objects, which is known as **autoboxing** and **unboxing**.

- **Autoboxing**: Converting a primitive to a wrapper class object automatically.

```
Integer num = 10; // Autoboxing (int to Integer)
```

- **Unboxing**: Converting a wrapper class object back to a primitive type automatically.

```
int num = numWrapper; // Unboxing (Integer to int)
```

4. Converting String to Wrapper Classes (Integer)

In real-world applications, you often need to convert `String` data (like user input or data from a file) into numerical values.

- `Integer.valueOf(String s)`: Converts a `String` to an `Integer` object.

```
String str = "123";
Integer integerValue = Integer.valueOf(str); // "123" is converted to Inte
ger object
```

- `Integer.parseInt(String s)`: Converts a `String` to a primitive `int`.

```
String str = "123";
int intValue = Integer.parseInt(str); // "123" is converted to primitive i
nt
```

5. Hexadecimal Conversion

Hexadecimal is a base-16 number system, commonly used in computing for compactly representing binary data. Java allows you to easily convert hexadecimal strings to integers.

- **Hexadecimal conversion using** `Integer.valueOf(String s, 16)`:

```
String hexStr = "1A";  // Hexadecimal String
Integer hexValue = Integer.valueOf(hexStr, 16);  // Converts "1A" to 26 (d
ecimal)
System.out.println(hexValue);  // Output: 26
```

- Here, `"1A"` is a hexadecimal number, which represents the decimal value `26`.

6. Why Use Wrapper Classes?

- **Collections**: You can only store objects in collections like `ArrayList`, so you need to use wrapper classes for primitive types.

```
ArrayList<Integer> list = new ArrayList<>();
list.add(5); // Autoboxing: 5 is converted from int to Integer
```

- **Nullability**: Wrapper classes can be set to `null`, while primitives cannot.

```
Integer nullWrapper = null;  // This is valid
int primitive = null;       // This will give a compilation error
```

7. Parsing and Formatting

You often need to convert `String` values to other data types, especially when dealing with input or text-based data.

- **Parsing**: Converting `String` to primitive types or wrapper class objects.

```
String str = "200";
int num = Integer.parseInt(str);  // Parsing String to int
```

- **Formatting**: Converting numerical values to `String` for display purposes.

```
int num = 123;
String str = Integer.toString(num);  // Converts int to String
```

8. Working with Strings in Java

Strings are an important part of Java, used to represent text. They are **objects** and come with a variety of methods for manipulation.

- **String Concatenation**:

```
String str1 = "Hello";
String str2 = "World";
String result = str1 + " " + str2;  // Concatenation
System.out.println(result);  // Output: Hello World
```

- **String to Wrapper Conversion**:
  When working with strings that represent numeric values, you can convert them into their respective wrapper class objects for further manipulation.

9. Important Concepts Recap

- **Primitive Types**: Store simple values (e.g., `int`, `char`, etc.).
- **Wrapper Classes**: Box primitive types into objects (e.g., `Integer`, `Character`, etc.).
- **Autoboxing & Unboxing**: Automatic conversion between primitives and wrapper classes.
- **String and Wrapper Class Conversion**: Convert strings to primitive types or wrapper objects (`Integer.valueOf()`, `Integer.parseInt()`).
- **Hexadecimal Conversion**: Convert hexadecimal strings to integers using `Integer.valueOf()` with base `16`.

10. Real-World Example:

Imagine you have a program where you need to read user input, process some mathematical calculations, and print the result back to the user. You may need to convert the user's input (a string) into an integer to perform arithmetic and then convert it back to a string to display the result.

```
import java.util.Scanner;

public class WrapperExample {
    public static void main(String[] args) {
        // Create scanner object to read user input
        Scanner scanner = new Scanner(System.in);

        // Get user input as string
        System.out.print("Enter a hexadecimal number: ");
        String input = scanner.nextLine();  // Reading input

        // Convert the string input (hexadecimal) to integer
        Integer decimalValue = Integer.valueOf(input, 16);

        // Print the result
        System.out.println("Decimal value: " + decimalValue);  // Prints the
converted decimal value
    }
}
```

Summary:

- **Wrapper classes** are essential for object representation of primitive types in Java, especially when working with collections.
- You can convert `String` values to `Integer` or other wrapper classes using methods like `Integer.valueOf()` or `Integer.parseInt()`.
- **Autoboxing and unboxing** allow you to seamlessly use primitives and objects without worrying about explicit conversion.
- **Hexadecimal conversion** is useful in many applications, such as in low-level programming or working with binary data.