50 Keywords of Java

Keywords are reserved words in Java that have specific meanings and cannot be used for any other purpose, such as variable names or method names. Java has a total of 50 keywords, which are used to define the syntax and structure of Java programming language.

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Rules to follow for keywords:

Keywords cannot be used as an identifier for class, subclass, variables, and methods.
Keywords are case-sensitive.

Some significant points about Java keywords:

const and goto are reserved words but not used.
True, false, and null are literals, not keywords.
All keywords are in lower-case.

Here's a list of Java keywords:

abstract: used to declare a class or method as abstract. An abstract class is a class that cannot be instantiated, and an abstract method is a method without a body that must be implemented in a subclass.

assert: used to perform assertion testing in Java. An assertion is a statement that should always be true, and if it is false, then an AssertionError is thrown.

strictfp: used to enforce strict floating-point precision in Java.

byte: used to declare a byte variable, which is a data type that can store values from -128 to 127.

instanceof: used to check if an object is an instance of a particular class or interface.
enum: used to declare an enumeration, which is a type that consists of a set of named constants.

native: used to declare a method as native, which means that its implementation is provided by the underlying platform, rather than in Java code.

volatile: The volatile keyword in Java is used to indicate that a variable's value may be modified by multiple threads simultaneously. It ensures that the variable is always read from and written to the main memory, rather than from thread-specific caches, ensuring visibility across threads.

transient: used to declare a variable as transient, which means that it will not be serialized when the object is written to a file or transmitted over a network.

synchronized: used to ensure that only one thread can access a block of code or object at a time in Java.

final: used to declare a variable or method as final, which means that its value or implementation cannot be changed.

static: used to declare a variable or method as static, which means that it belongs to the class rather than to individual objects of the class.

super: used to call a method or constructor in the superclass.

this: used to refer to the current object in Java

class: used to declare a class in Java.

extends: used to extend a class in Java.

interface: used to declare an interface in Java.

implements: used to implement an interface in Java.

package: used to define a package in Java.

import: used to import a package or class into a Java program.

private: used to declare a variable or method as private, which means that it can only be accessed within the same class.

protected: used to declare a variable or method as protected, which means that it can be accessed within the same class or any subclass.

public: used to declare a variable or method as public, which means that it can be accessed from anywhere in the Java program.

break: used to break out of a loop or switch statement.

continue: used to skip the current iteration of a loop and continue to the next iteration.

switch: used to start a switch statement in Java.

case: used in a switch statement to define a case label.

default: used in a switch statement to define a default case.

try: used to start a try-catch block in Java.

catch: used to catch and handle exceptions in Java.

finally: used in a try-catch block to define a block of code that will always be executed, regardless of whether an exception is thrown or not.

throw: used to throw an exception in Java.

throws: used to declare that a method may throw an exception in Java.

new: used to create a new object in Java.

return: used to return a value from a method or exit a method without returning a value.

BIG DECIMAL

BigDecimal class is a useful tool for performing mathematical operations with a high degree of precision. Unlike the primitive double and float data types, BigDecimal can accurately represent decimal numbers and avoid common rounding errors.

```
import java.math.BigDecimal;
public class Main{
    public static void main(String[] args) {
        double x = 1.05;
        double y = 2.55;
        System.out.println(x+y);//3.5999999999999996
        BigDecimal num1 = new BigDecimal("1.05");
        BigDecimal num2 = new BigDecimal("2.55");
        BigDecimal add = num1.add(num2);//3.60
        BigDecimal sub = num2.subtract(num1);//1.50
        BigDecimal devide = num2.divide(num1,2,BigDecimal.ROUND_HALF_UP);//2.43
        BigDecimal num3 = new BigDecimal("1.05");
        boolean result1 = num1.equals(num3);//true
        BigDecimal maxNum = num1.max(num2);//2.55
        BigDecimal minNum = num1.min(num2);//1.05
    }
}
```

Note : num2.divide(num1,2,BigDecimal.ROUND_HALF_UP);

- 2 is used as the scale to round the result to two decimal places
- BigDecimal.ROUND_HALF_UP is used as the rounding mode, which rounds towards the nearest neighbor and , in case of a tie, rounds away from zero.
- New BigDecimal(String argument here for accurate data);

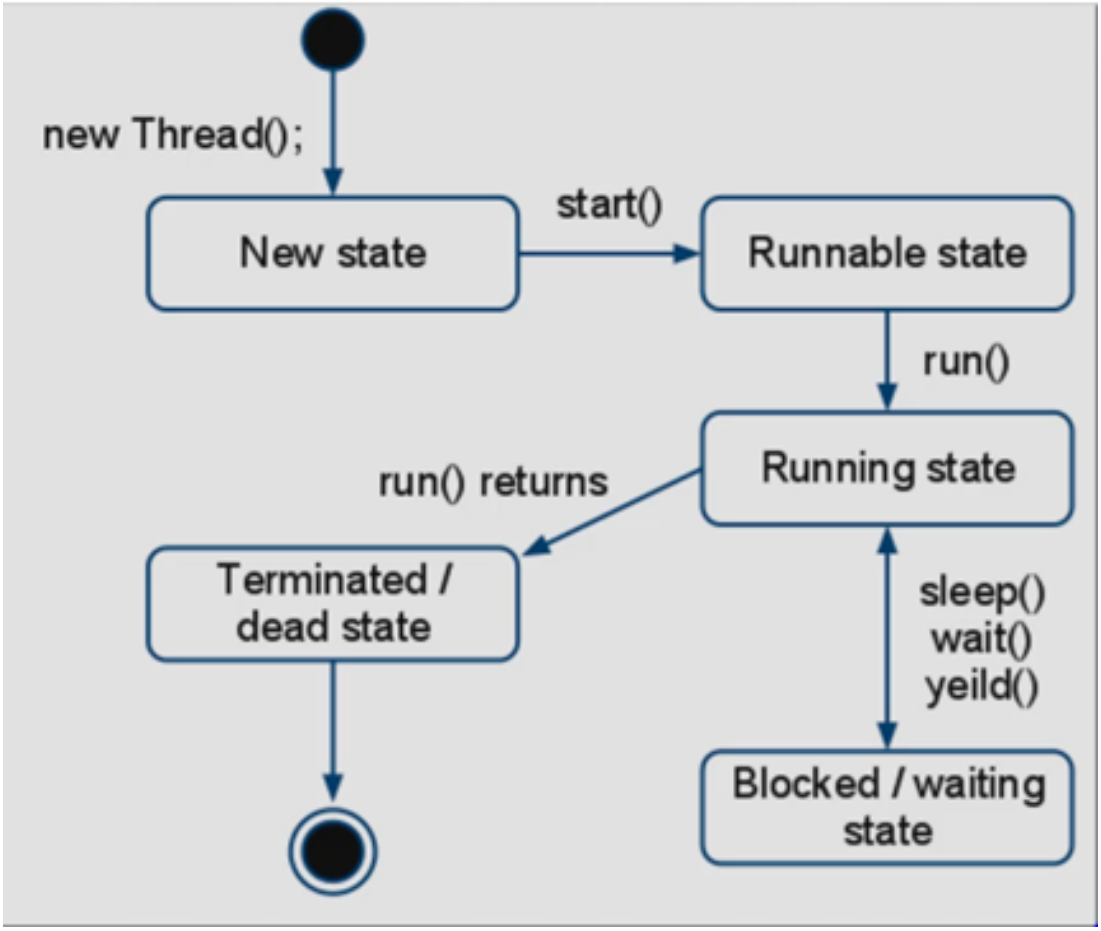
literals

Literals are nothing but the constant we assign to variable.

```
public static void main(String[] args) {
    int a = 100000;//100000
    int b = 1_00_000;//100000
    int binary = 0b101;//5
    int hexa = 0x5E;//94
    int octal = 023;//19
    double d = 12e5;//
    char c = 'a';
    c++;
    System.out.println(c);//b
}
```

Multithreading

A process of executing multiple threads simultaneously is known as Multithreading. However, we use multithreading rather than multiprocessing because threads use a shared memory area.



- **New:** when we create an instance of the Thread class, a thread is in a new state.
- **Runnable State:** Runnable status is a state where the thread is ready to run.
- **Running state:** A thread is executing where multi-threaded programming is derived by the hardware.
- **Blocked/ waiting state:** A Java thread can be blocked when a resource is expected.
- **Terminated/ dead state:** A thread can be terminated, which stops its execution immediately at any time. Once a thread is finished, it cannot be resumed.

Creating thread by extending Thread class

```
class MyCounter extends Thread{
    private int threadNo;
    public MyCounter(int threadNo) {
        this.threadNo = threadNo;
    }
    @Override
    public void run() {
        countMe();
    }
    public void countMe(){
        for(int i=1; i<=9;i++){
            try {
                sleep(500);
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
            System.out.println("The value if i is: "+i+" and the thread
number is: "+threadNo);
        }
    }
}
public class App {
    public static void main(String[] args) throws InterruptedException {
        MyCounter counter1 = new MyCounter(1);
        MyCounter counter2 = new MyCounter(2);
        long startTime = System.currentTimeMillis();
        counter1.run(); //Run should be called by JVM
        System.out.println("*************************");
        counter2.start(); //so replace run by start
        long endTime = System.currentTimeMillis();
        System.out.println("Total time required to process: "+(endTime-
startTime));
    }
}
```

Creating Thread by implementing Runnable Interface

```
import java.util.Random;

class MyCounter implements Runnable{

    private int threadNo;

    public MyCounter(int threadNo) {

        this.threadNo = threadNo;

    }


    @Override

    public void run() {

        Random random = new Random();

        for(int i = 0;i<=9;i++){

            try {

                Thread.sleep(random.nextInt(500));

            } catch (InterruptedException e) {

                // TODO Auto-generated catch block

                e.printStackTrace();

            }

            System.out.println("The value if i is: "+i+" and the thread number is:
"+threadNo);

        }

    }

}


public class App {

    public static void main(String[] args) throws InterruptedException {

        Thread thread1 = new Thread(new MyCounter(1));

        Thread thread2 = new Thread(new MyCounter(2));

        thread1.start();

        thread2.start();

    }

}
```

In the following code, we create an anonymous object and start the thread immediately.

```
public class App {
    public static void main(String[] args) throws InterruptedException {
        new Thread(new Runnable() {
            @Override
            public void run() {
                for(int i=1;i<10;i++){
                    try {
                        Thread.sleep(500);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    };
                    System.out.println(i);
                }
            }
        }).start();
        new Runnable(){}.start();//2nd thread and so on as required
    }
}
```

Synchronization

Synchronization in Java is a technique used to control access to shared resources in a multi-threaded environment.
It ensures that only one thread can access a shared resource at a time, preventing data corruption and ensuring consistency.
There are two main mechanisms for synchronization:

I. `synchronized` methods or blocks,
II. and the `java.util.concurrent` package.

1. `synchronized` keyword:

a. Synchronized Methods:

You can use the `synchronized` keyword with a method to ensure that only one thread can execute the method at a time.

```
public synchronized void synchronizedMethod() {
   // Synchronized code
}
```

b. Synchronized Blocks:

You can use synchronized blocks to control access to a specific section of code. This allows for more fine-grained control compared to synchronized methods.

```
public void someMethod() {
   // Non-critical section
   synchronized (lockObject) {
      // Critical section (synchronized code)
   } // Non-critical section}
```

It's common to use a dedicated object (e.g., `lockObject` in the example) as a lock to avoid potential deadlocks.

2. `java.util.concurrent` package:

Java provides more advanced synchronization utilities in the `java.util.concurrent` package, offering higher-level abstractions.

a. `Lock` Interface:

The `Lock` interface provides a more flexible alternative to the `synchronized` keyword, allowing for more sophisticated locking mechanisms.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class Example {
    private final Lock lock = new ReentrantLock();
    public void someMethod() {
        lock.lock();
        try {
            // Synchronized code
        } finally {
            lock.unlock();
        }
    }
}
```

b. `Semaphore` and `CountDownLatch`:

These classes are used for more complex synchronization scenarios. `Semaphore` can be used to control access to a resource, and `CountDownLatch` can be used to synchronize multiple threads.

Best Practices for Synchronization:

1. Minimize Synchronized Code:

Only synchronize the critical sections of your code to minimize the impact on performance.

2. Use `java.util.concurrent` Utilities:

When applicable, prefer the higher-level abstractions provided by the `java.util.concurrent` package.

3. Avoid Deadlocks:

Be cautious about nested locks and ensure a consistent locking order to avoid potential deadlocks.

4. Use `volatile` Modifier:

The `volatile` modifier can be used to ensure visibility of changes across threads.

5. Consider Thread Safety:

Make sure that shared data structures are designed to be thread-safe, or use synchronization mechanisms to protect them.

wait() and its overloads, notify(), notifyAll(), interupt()

```
public class App {
    static public int balance = 0;
    public void withdraw(int amount) {
        synchronized (this) {
            if (balance <= 0) {
                try {
                    System.out.println("Waiting for balance updation");
                    wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
        balance = balance - amount;
        System.out.println("Withdrawal successful and the current balance is: " +
balance);
    }
    public void deposit(int amount) {
        System.out.println("We are depositing the amount");
        balance = balance + amount;
    }
    public static void main(String[] args) {
        App app = new App();
        Thread thread1 = new Thread(new Runnable() {
            @Override
            public void run() {
                app.withdraw(1000);
            }
        });
        thread1.setName("Thread1");
        thread1.start();
        Thread thread2 = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                app.deposit(2000);
            }
        });
        thread2.setName("Thread");
        thread2.start();
    }
}
```

When using `wait()`, `notify()`, and `notifyAll()`, it's crucial to follow best practices to avoid issues like missed signals and deadlocks. Always use these methods within a synchronized block and carefully design your synchronization logic to ensure correct behavior in a multi-threaded environment.

the `wait()`, `notify()`, and `notifyAll()` methods are used in conjunction with the `synchronized` keyword to implement inter-thread communication and synchronization.

These methods are part of the `Object` class and are commonly used in scenarios where one thread needs to wait for a certain condition to be met before proceeding, and another thread needs to signal that the condition has been satisfied.

1. wait():
- The wait() method is used to make a thread wait until another thread invokes the notify() or notifyAll() method for the same object.
- It must be called from within a synchronized block.
- It releases the lock on the object, allowing other threads to enter the synchronized block.

```
synchronized (sharedObject) {
   while (conditionNotMet) {
   sharedObject.wait();
   }
   // Code to execute after condition is met
}
```

2. notify():
- The notify() method is used to wake up one of the threads that are currently waiting on the same object.
- It must be called from within a synchronized block.
- It does not release the lock immediately; the thread calling notify() must exit the synchronized block to release the lock.

```
synchronized (sharedObject) {
   // Code to change the condition
   sharedObject.notify();
}
```

3. notifyAll():
- The notifyAll() method is similar to notify() but wakes up all threads that are currently waiting on the same object.
- It must be called from within a synchronized block.
- It is often used to avoid the issue of missed signals when multiple threads are waiting.

```
synchronized (sharedObject) {
   // Code to change the condition
   sharedObject.notifyAll();
}
```

4. interrupt():
- The interrupt() method is not directly related to `wait()` and `notify()`, but it is used for interrupting a thread that is currently in a blocked state.
- If a thread is waiting, sleeping, or in some other blocking operation, calling `interrupt()` on that thread will cause it to throw an `InterruptedException`.
- Handling interruptions is the responsibility of the developer and can be used to gracefully terminate a thread or take other appropriate action.

5. join():

The joining method is an optimal solution to the problem of inconsistency that occurs due to threads not having a proper order. The join method is used to notify that the thread will wait until the execution of the given thread before execution.

```
  public class App {
    public int counter = 0;
    public static void main(String[] args) {
        App app = new App();
        Thread thread1 = new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 1000; i++) {
                    app.counter++;
                }
            }
        });
        Thread thread2 = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    thread1.join(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                for (int i = 0; i < 1000; i++) {
                    app.counter++;
                }
            }
        });
        thread1.start();
        thread2.start();
        try {
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("The value of counter: " + app.counter);
    }
}
```

Concurrency Control

Concurrency control in Java involves managing the execution of multiple threads to ensure correctness, consistency, and avoid potential issues such as race conditions, deadlocks, and data corruption. Java provides several mechanisms for concurrency control:

1. Synchronization (using `synchronized` keyword):

As discussed earlier, synchronization is a fundamental mechanism in Java for controlling access to shared resources. It can be applied to methods or blocks of code using the `synchronized` keyword. Synchronization ensures that only one thread can access the synchronized code at a time, preventing data corruption.

```
public synchronized void synchronizedMethod() {
    // Synchronized code
}
```

2. Volatile Keyword:

The `volatile` keyword is used to indicate that a variable's value may be changed by multiple threads simultaneously. It ensures that reads and writes to the variable are atomic and that changes are visible to all threads.

```
private volatile int sharedVariable;
```

3. Locks (using `Lock` interface):

The `Lock` interface provides a more flexible alternative to the `synchronized` keyword. It allows for explicit control over locks, enabling features like reentrant locking, timed waiting, and interruption handling.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Example {
    private final Lock lock = new ReentrantLock();

    public void someMethod() {
        lock.lock();
        try {
            // Synchronized code
        } finally {
            lock.unlock();
        }
    }
}
```

4. Atomic Classes (e.g., `AtomicInteger`, `AtomicReference`):

The `java.util.concurrent.atomic` package provides atomic classes that support atomic operations on single variables. These classes are useful for scenarios where simple operations need to be performed without explicit synchronization.

```
import java.util.concurrent.atomic.AtomicInteger;
public class Example {
    private AtomicInteger atomicCounter = new AtomicInteger(0);

    public void incrementCounter() {
        atomicCounter.incrementAndGet();
    }
}
```

5. Thread Pools (using `ExecutorService`):

Java's `ExecutorService` and related interfaces provide a higher-level abstraction for managing and controlling thread execution. Thread pools can be used to limit the number of concurrent threads, manage thread lifecycle, and improve performance.We need ExecutorService as there is some limit to hardware we cannot run lakhs of threads simultaneously , so this service will manage it well.

```
class SomeThread extends Thread{
    private String name;
    public SomeThread(String name) {
        this.name = name;
    }
    @Override
    public void run() {
        System.out.println("Starting thread: "+name);
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println("Thread ended: "+name);
    }
}
public class App {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(4);
        //at max 4 threads will be running in parallel

        SomeThread thread1 = new  SomeThread("Thread 1");
        SomeThread thread2 = new  SomeThread("Thread 2");
        SomeThread thread3 = new  SomeThread("Thread 3");
        SomeThread thread4 = new  SomeThread("Thread 4");
        SomeThread thread5 = new  SomeThread("Thread 5");
        SomeThread thread6 = new  SomeThread("Thread 6");
        SomeThread thread7 = new  SomeThread("Thread 7");
        SomeThread thread8 = new  SomeThread("Thread 8");
        SomeThread thread9 = new  SomeThread("Thread 9");
        SomeThread thread10 = new  SomeThread("Thread 10");
        SomeThread thread11 = new  SomeThread("Thread 11");
        SomeThread thread12 = new  SomeThread("Thread 12");

        executorService.execute(thread1);
        executorService.execute(thread2);
        executorService.execute(thread3);
        executorService.execute(thread4);
        executorService.execute(thread5);
        executorService.execute(thread6);
        executorService.execute(thread7);
        executorService.execute(thread8);
        executorService.execute(thread9);
        executorService.execute(thread10);
        executorService.execute(thread11);
        executorService.execute(thread12);

        executorService.shutdown();//manual shutdown is required
    }
}
```

6. Read-Write Locks (using `ReentrantReadWriteLock`):

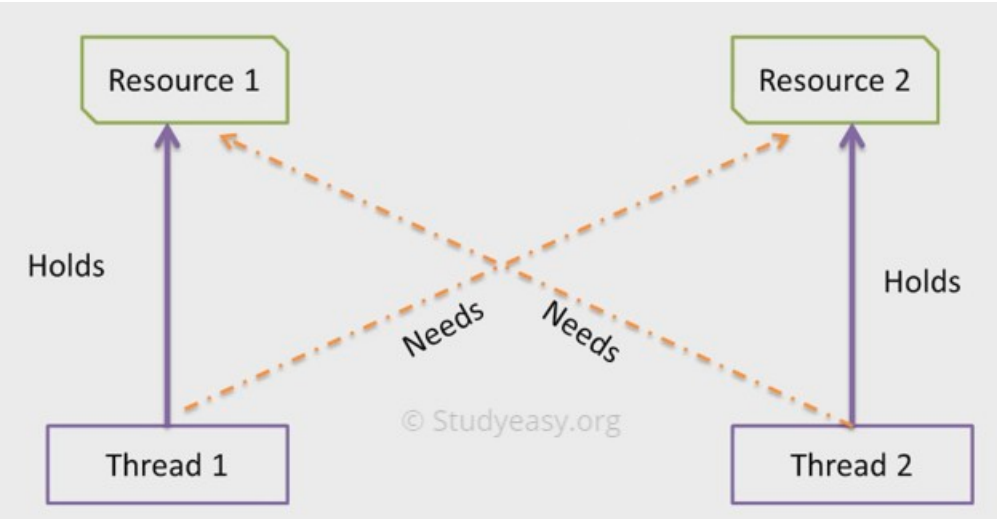
The `ReentrantReadWriteLock` provides a mechanism for controlling access to shared resources with multiple threads. It allows multiple threads to read a resource concurrently but ensures exclusive access for writing.

```
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class Example {
    private final ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
    public void readData() {
        readWriteLock.readLock().lock();
        try {
            // Read data
        } finally {
            readWriteLock.readLock().unlock();
        }
    }
    public void writeData() {
        readWriteLock.writeLock().lock();
        try {
            // Write data
        } finally {
            readWriteLock.writeLock().unlock();
        }
    }
}
```

DeadLock :

A deadlock is a situation in concurrent programming where two or more threads are blocked indefinitely, each waiting for the other to release a resource. Deadlocks occur when there is a circular wait among two or more threads, and each thread holds a resource that another thread needs.



Solution : do not use nested locks , and rather than lock() or manual object lock use reentrant tryLock() method.

CountDownLatch : The basic use of this is that we can place the thread that we want to execute after a certain number of threads. This can be done by setting the countdown number to the number of threads after which we want the specific thread to be executed. In other words , allows one or more threads to wait for a set of operations to be completed by other threads before proceeding.

```
class SomeThread extends Thread{
    private CountDownLatch latch;
    public SomeThread(CountDownLatch latch) {
        this.latch = latch;
    }
    @Override
    public void run() {
        System.out.println("Thread running with thread name
            "+Thread.currentThread().getName());
        System.out.println("Thread execution completed");
        System.out.println("************************");
        latch.countDown();//counting threads that are executed
    }
}
public class App {
    public static void main(String[] args) {
        CountDownLatch latch = new CountDownLatch(4);
        /*
        The argument specifies threads after latch.await() will be executed
         after atleast 4 threads.

        if we start
        * 3 threads -> program will never terminate
        * 4 and more threads -> program will terminate after all threads execution
        */
        SomeThread thread1 = new SomeThread(latch);
        SomeThread thread2 = new SomeThread(latch);
        SomeThread thread3 = new SomeThread(latch);
        SomeThread thread4 = new SomeThread(latch);
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();
        try {
            latch.await();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println("I am in main thread");
    }
}
```

BlockingQueue :

1. It is a queue that supports blocking operations when certain conditions are not met.
2. It is part of the `java.util.concurrent` package
3. It provides an efficient and thread-safe way for multiple threads to communicate and share data.
4. The primary use case for a `BlockingQueue` is to implement producer-consumer scenarios, where one or more threads produce items, and one or more threads consume them.

Some key methods and types associated with `BlockingQueue` include:

1. `put(E element)` and `take()`:

I. **`put(E element)`** is used to add an element to the queue. If the queue is full, it will block until space is available.

II. **`take()`** is used to retrieve and remove an element from the queue. If the queue is empty, it will block until an element is available.

```
BlockingQueue<String> blockingQueue = new LinkedBlockingQueue<>();
// Producer thread
blockingQueue.put("Item 1");
// Consumer thread
String item = blockingQueue.take();
```

2. `offer(E element, long timeout, TimeUnit unit)` and `poll(long timeout, TimeUnit unit)`:

- **`offer(E element, long timeout, TimeUnit unit)`** adds an element to the queue if space is available, blocking for the specified time if the queue is full.
- **`poll(long timeout, TimeUnit unit)`** retrieves and removes an element from the queue, blocking for the specified time if the queue is empty.

```
BlockingQueue<Integer> blockingQueue = new LinkedBlockingQueue<>(5);
// Producer thread
boolean added = blockingQueue.offer(42, 1, TimeUnit.SECONDS);
// Consumer thread
Integer item = blockingQueue.poll(1, TimeUnit.SECONDS);
```

3. Types of `BlockingQueue`:

- There are different implementations of `BlockingQueue`, each with its characteristics. Common types include
  1. **`LinkedBlockingQueue`,**
  2. **`ArrayBlockingQueue`,**
  3. **`PriorityBlockingQueue`,**
  4. **`DelayQueue`.**

```
BlockingQueue<String> linkedQueue = new LinkedBlockingQueue<>();
BlockingQueue<Integer> arrayQueue = new ArrayBlockingQueue<>(10);
BlockingQueue<Runnable> priorityQueue = new PriorityBlockingQueue<>();
```

4. `BlockingQueue` in the Executor Framework:

- `BlockingQueue` is often used in conjunction with the Executor framework for managing a pool of worker threads. The `ThreadPoolExecutor` class, for example, uses a `BlockingQueue` to hold tasks.

```
BlockingQueue<Runnable> taskQueue = new LinkedBlockingQueue<>();
Executor executor = new ThreadPoolExecutor(5, 10, 1, TimeUnit.MINUTES, taskQueue);
```

Using a `BlockingQueue` simplifies coordination between producer and consumer threads, providing a clean and efficient way to implement concurrent systems. It helps avoid issues such as busy-waiting and explicit locking, which can be error-prone and inefficient in a multi-threaded environment.

```
class Producer implements Runnable{
    private ArrayBlockingQueue<Integer> queue;
    public Producer(ArrayBlockingQueue<Integer> queue) {
        this.queue = queue;
    }
    @Override
    public void run() {
        while(true){
            try {
                Thread.sleep(1000);
                queue.put(App.counter++);
                System.out.println("Value added in the queue: "+App.counter);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
class Consumer implements Runnable{
    private ArrayBlockingQueue<Integer> queue;
    public Consumer(ArrayBlockingQueue<Integer> queue) {
        this.queue = queue;
    }
    @Override
    public void run() {
        while(true){
            try {
                Thread.sleep(5000);
                queue.take();
                App.counter--;
                System.out.println("Value removed in the queue: "+App.counter);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public class App {
    static int counter = 0;
    public static void main(String[] args) {
        ArrayBlockingQueue<Integer> queue = new ArrayBlockingQueue<>(10);
        Producer produce = new Producer(queue);
        Thread producerThread = new Thread(produce);
        producerThread.start();
        Consumer consume = new Consumer(queue);
        Thread consumerThread = new Thread(consume);
        consumerThread.start();
    }
}
```

Semaphore :

1. it is a synchronization primitive in Java that controls access to a shared resource by maintaining a set number of permits.
2. Threads can acquire permits from the `Semaphore` before accessing the resource, and they must release the permits when done.
3. It is often used to control access to a resource that has a limited capacity.
4. The `Semaphore` class is part of the `java.util.concurrent` package and provides a flexible mechanism for managing concurrent access to resources.

```
import java.util.concurrent.Semaphore;
 public class SemaphoreExample {
     private static final int MAX_AVAILABLE_PERMITS = 3;
     private Semaphore semaphore = new Semaphore(MAX_AVAILABLE_PERMITS);
     public static void main(String[] args) {
             for (int i = 1; i <= 5; i++) {
                     Thread thread = new Thread(new Worker(i));
                     thread.start();
             }
     }
  }
  class Worker implements Runnable {
    private final int id;
    public Worker(int id) {
       this.id = id;
    }
    @Override
    public void run() {
       try {
          System.out.println("Thread " + id + " is trying to acquire a permit.");
          semaphore.acquire(); // Acquire a permit
          System.out.println("Thread " + id + " has acquired a permit and is performing some work.");
          // Simulate some work
          Thread.sleep(2000);
          System.out.println("Thread " + id + " is releasing the permit.");
       } catch (InterruptedException e) {
          e.printStackTrace();
       } finally {
          semaphore.release(); // Release the permit
       }
    }
 }
}
```

`Semaphore` is useful in scenarios where you want to limit the number of concurrent threads accessing a particular resource or section of code. It provides more flexibility than other synchronization constructs like `synchronized` blocks or methods.

Additionally, `Semaphore` supports the concept of fairness, where the first thread that requests a permit is the first one to acquire it. This behavior can be specified when creating a `Semaphore` by passing `true` as the second argument to the constructor:

```
Semaphore semaphore = new Semaphore(MAX_AVAILABLE_PERMITS, true);
```

This is useful in scenarios where fairness is desired in resource allocation.