Intro to Multithreading

Concurrency

Concurrency refers to breaking up a task or piece of computation into different parts that can be executed independently, out of order, or in partial order without affecting the final outcome. One way - but not the only way - of achieving concurrency is by using multiple threads in the same program.

Operating systems use concurrency to manage the many different programs that run on them. The GUI - graphical user interface - for example, is run at the same time as other processes. Without this, any process that took too long in the background, like reading / writing to files or making an HTTP request, would block the GUI and prevent any other user input.

Multi-core Processing

Most computers these days have multiple cores or CPUs, which means that calculations at the hardware level can be done in parallel. Without multiple cores, operating systems can still achieve concurrency with a process called **time splicing** - this means running one process for a short time, then switching to another, and back very rapidly. This ensures that no process or application is completely blocked.

On multi-core systems, different processes can be run on different CPUs entirely. This enables true parallelization and is a key benefit of writing multithreaded programs.

Introduction to Threads

A thread is a subset of a process that is also an independent sequence of execution, but threads of the main process run in the same memory space, managed independently by a scheduler. So, we can think of a thread as a "path of execution", but they can access the same objects in memory.

Every thread that is created in a program is given its own call stack, where it stores local variables references. However, all threads share the same heap, where the objects live in memory. Thus, two threads could have separate variable references on two different stacks that still point to the same object in the heap.

Multithreading

Multithreading extends the idea of multitasking into applications where you can subdivide operations in a single application into individual, parallel threads. Each thread can have its own task that it performs. The OS divides processing time not just with applications, but between threads. Multi-core processors can actually run multiple different processes and threads concurrently, enabling true parallelization.

In Java, multithreading is achieved via the `Thread` class and/or the `Runnable` interface.

A Note on Best Practices

In general, it is best to avoid implementing multithreading yourself if possible. The benefit of multithreaded applications is better performance due to non-blocking execution. However, you should always measure or attempt to estimate the performance benefit you will get by using threads versus the tradeoff in complexity and subtle bugs that might be generated. Usually there are frameworks, tools, or libraries that have implemented the problem you are trying to solve, and you can leverage those instead of trying to build your own solution. For example, web servers like Apache Tomcat have multithreading built-in and provide APIs for dealing with network requests without having to worry about threads.

Thread Class

In Java, multithreading is achieved via the `Thread` class and/or the `Runnable` interface.

Thread methods

A few important methods in the `Thread` class include:

- getters and setters for id, name, and priority
- `interrupt()` to explicitly interrupt the thread
- `isAlive()`, `isInterrupted()` and `isDaemon()` to test the state of the thread
- `join()` to wait for the thread to finish execution
- `start()` to actually begin thread execution after instantiation

A few important `static` methods are also defined:

- `Thread.currentThread()` which returns the thread that is currently executing
- `Thread.sleep(long millis)` which causes the currently executing thread to temporarily stop for a specified number of milliseconds

Thread Priorities

Priorities signify which order threads are to be run. The Thread class contains a few static variables for priority:

- MIN_PRIORITY = 1
- NORM_PRIORITY = 5, default
- MAX_PRIORITY = 10

Creating Threads using `Thread` class

- Create a class that extends `Thread`
    - implement the `run()` method

- instantiate your class
- call the `start()` method

```
public class MyThread extends Thread {
        @Override
        public void run() {
                System.out.println("Inside the MyThread class");
        }
}
public class ThreadDemo {
        public static void main(String[] args) {
                Thread myRunnable = new Thread(new MyRunnable());
                Thread myThread = new MyThread();
                myRunnable.start();
                myThread.start();
        }
}
```

Runnable Interface

In Java, multithreading is achieved via the `Thread` class and/or the `Runnable` interface.

`java.lang.Runnable` is an interface that is to be implemented by a class whose instances are intended to be executed by a thread.

Creating Threads using `Runnable` Interface

- Create a class that implements the `Runnable` functional interface
  - implement the `run()` method
  - pass an instance of your class to a `Thread` constructor
  - call the `start()` method on the thread

```
public class MyRunnable implements Runnable {
        @Override
        public void run() {
                System.out.println("Inside the MyRunnable class");
        }
```

```
        }
```

Runnable and Lambda Expressions

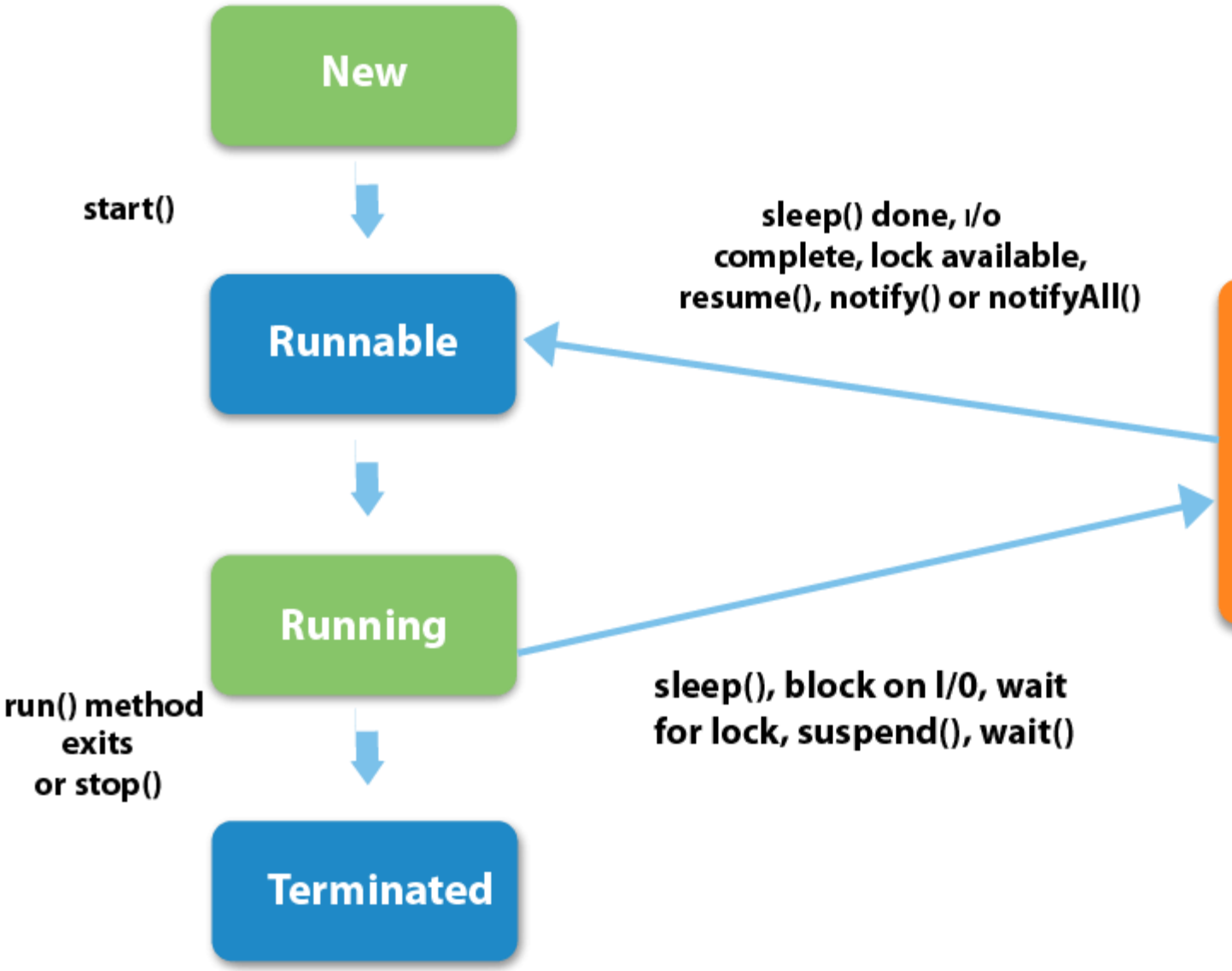
Because `Runnable` is a *functional* interface, we can use a lambda expression to define thread behavior inline instead of implementing the interface in a separate class. We pass a lambda expression as the `Runnable` type required in the `Thread` constructor. For example:

```
public class ThreadLambda {
  public static main(String[] args) {
    Thread willRun = new Thread(() -> {
          System.out.println("Running!");
        });
        willRun.start();
  }
}
```

States of a Thread

At any given time, a thread can be in one of these states:

1. **New**: newly created thread that has not started executing
2. **Runnable**: either running or ready for execution but waiting for its resource allocation
3. **Blocked**: waiting to acquire a monitor lock to enter or re-enter a synchronized block/method
4. **Waiting**: waiting for some other thread to perform an action without any time limit
5. **Timed_Waiting**: waiting for some other thread to perform a specific action for a specified time period
6. **Terminated**: has completed its execution



Synchronization

Synchronization is the capability to control the access of multiple threads to any shared resource.

Synchronized keyword

In a multithreaded environment, a race condition occurs when 2 or more threads attempt to access the same resource. Using the `synchronized` keyword on a piece of logic enforces

that only one thread can access the resource at any given time. `synchronized` blocks or methods can be created using the keyword. Also, one way a class can be "thread-safe" is if all of its methods are `synchronized`.

```
synchronized(objectidentifier) {
   // Access shared variables and other shared resources
}
```

Producer-Consumer Problem

The Producer-Consumer problem is a classic example of a multi-process synchronization problem. Here, we have a *fixed-size buffer* and two classes of threads - *producers* and *consumers*. Producers produces the data to the queue and Consumers consume the data from the queue. Both producer and consumer shares the same fixed-size buffer as a queue.

Problem - The producer should produce data only when the queue is not full. If the queue is full, then the producer shouldn't be allowed to put any data into the queue. The consumer should consume data only when the queue is not empty. If the queue is empty, then the consumer shouldn't be allowed to take any data from the queue.

We can solve the Producer-Consumer problem by using `wait()` & `notify()`methods to communicate between producer and consumer threads. The `wait()` method to pause the producer or consumer thread depending on the queue size. The `notify()` method sends a notification to the waiting thread.

Producer thread will keep on producing data for Consumer to consume. It will use `wait()` method when Queue is full and use `notify()` method to send notification to Consumer thread once data is added to the queue.

```
public synchronized void produce() {
        while (queue.size() == MAX_SIZE) {
                //Queue is full, Producer thread waiting for consumer to take data f
rom the queue
                wait();
        }
        /* When queue has space, Producer produces the data and adds them into the q
ueue.
        *  After that, Producer sends the notification to the Consumer.
        */
        //producing data
        queue.add(data);
        notify();
```

```
}
```

Consumer thread will consume the data form the queue. It will also use `wait()` method to wait if queue is empty. It will also use `notify()` method to send notification to producer thread after consuming data from the queue.

```
public synchronized String consume() {
        while (messages.isEmpty()) {
                //Queue is empty, Consumer thread waiting for producer to put data t
o the queue
                wait();
        }
                /* When queue has data, Consumer consumes the data and removes it fr
om the queue.
                *  After that, Consumer sends the notification to the Producer.
                */
                //consuming data
                queue.remove(data);
                notify()
}
```

Lambda Expressions

Lambda expressions are one of the biggest new features of Java 8, and they introduce some important aspects of **functional programming** to Java. The most basic syntax of a lambda expression is:

```
parameter(s) -> expression
```

For example, we can use the `.forEach` method of the `Iterable` interface, which accepts a lambda expression as its argument:

```
List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");
names.add("Charlie");
names.forEach(str -> System.out.println(str));
```

This will print out the names just as if we had used a `for` loop. The lambda syntax could also be done with an explicit type declaration for the parameter, but the compiler can infer

the type from the value used. For multiple parameters, parentheses are required around them. Also, curly braces are optional for single statements but required for multiple. Finally, the `return` keyword is also optional for a single expression because the value will be returned by default.

Functional Interfaces

Functional interfaces are interfaces that **have only one abstract method**. This method is what lambdas are implementing when they are declared - the parameter types and return types of the lambda must match the functional interface method declaration. The Java 8 JDK comes with many built-in functional interfaces, listed in the Javadocs link above.

We can also use functional interfaces as types to which we can assign lambda functions, like so:

```
interface MyFunctionalInt {
  int doMath(int number);
}


public class Execute {
  public static void main(String[] args) {
    MyFunctionalInt doubleIt = n -> n * 2;
        MyFunctionalInt subtractIt = n -> n - 2;
        int result1 = doubleIt.doMath(2);
        int result2 = subtractIt.doMath(8);
        System.out.println(result1); // 4
        System.out.println(result2); // 6
  }
}
```

Stream API

The Java 8 Stream API is a functional-style way of defining operations on a stream of elements. Streams are an abstraction which allow defining operations which do not modify the source data and are lazily executed. Streams **do not store data**, they simply define operations like filtering, mapping, or reducing, and can be combined with other operations and then executed. Some built-in `Stream`s are located in the `java.util.stream` package.

Streams are divided into *intermediate* and *terminal* operations. Intermediate streams return a new stream and are always lazy - they don't actually execute until a terminal operation is

called. Terminal operations trigger the execution of the stream pipeline, which allows efficiency by performing all operations in a single pass over the data.

Finally, reduction operations take a sequence of elements and combine them into a single result. Stream classes have the `reduce()` and `collect()` methods for this purpose, with many built-in operations defined in the `Collectors` class.

```
List<Student> students = new ArrayList<>();
// add students...
List<Double> grades = students.stream()
                          .filter(s -> s.isAttending())
                                                  .mapToDouble(s -> s.getGrade)
                                                  .collect(Collectors.toList());
```

Serialization

Serialization is the process of writing the state of an object to a byte stream; the reverse is called deserialization. In order for an object to be serialized, it must implement the `Serializable` interface.

Marker Interfaces

`Serializable` is a **marker interface**, which is an interface with no methods. The point of such an interface is to *provide metadata* to the compiler - in this case, it tells the compiler that this class can be serialized.

Serializing an object

To serialize an Object, you need a `FileOutputStream` instance inside the constructor of an `ObjectOutputStream`, passing in the file path of where you want the Object to be serialized

- Call the `ObjectOutputStream.writeObject(yourObject)` method

Deserializing an object

To deserialize an Object, you need a `FileInputStream` instance inside the constructor of an `ObjectInputStream`, passing in the file path of where the serialized object is

- Call the `ObjectInputStream.readObject()` method, casting it to a bean of your type

File I/O: FileInputStream/FileReader/FileWriter

File I/O

"I/O" refers to the nature of how data is accessed, either by reading it from a resource (input) or writing it to a resource (output). In Java, File I/O objects fall into one of two categories:

- `Streams` are for reading or writing **bytes**
- `Reader/Writer` are for reading or writing **characters**

Some Common File I/O classes are

- `FileInputStream` - reads raw bytes from a file
- `FileOutputStream` - writes raw bytes to a file
- `FileReader` - reads characters from a file
- `FileWriter` - writes characters to a file
- `BufferedReader` - reads a file line by line, needs an instance of a `FileReader` with a path to the resource to be read in the constructor
- `BufferedWriter` - writes to a file line by line, needs an instance of a `FileWriter` with a path to the destination file in the constructor
- `Scanner` - can read from an `InputStream`, useful methods for character reading

Character files are read line by line, either until a carriage return (`\r`) or a newline (`\n`), depending on your operating system. When using I/O classes to read and write, you should **always close your resources** with the `.close()` method. This prevents exceptions from being thrown later, memory leaks, and system overutilization of unused resources.

Reading User Input from Console

The `Scanner` class can be used to read user input from the command line:

```
Scanner sc = new Scanner(System.in);
while (true) {
  String input = sc.readLine();
  System.out.println("Your input: " + input);
}
```

When the code above is run, the program acts to "echo" back any input given from `stdin`.