

## Agenda

- Stream Programming
- Multi-Threading

## Stream creation

- Collection interface: stream() or parallelStream()
- Arrays class: Arrays.stream()
- Stream interface: static of() method
- Stream interface: static generate() method
- Stream interface: static iterate() method
- Stream interface: static empty() method
- nio Files class: `static Stream<String> lines(filePath)` method

## Stream creation

- Collection interface: stream() or parallelStream()

```
List<String> list = new ArrayList<>();  
// ...  
Stream<String> strm = list.stream();
```

- Arrays class: Arrays.stream()

```
Double arr[] = {1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9};  
Stream<Double> strm = Arrays.stream(arr);
```

- Stream interface: static of() method

```
Stream<Integer> strm = Stream.of(arr);
```

- Stream interface: static generate() method
  - generate() internally calls given Supplier in an infinite loop to produce infinite stream of elements.

```
Stream<Double> strm = Stream.generate(() -> Math.random()).limit(25);
```

```
Random r = new Random();  
Stream<Integer> strm = Stream.generate(() -> r.nextInt(1000)).limit(10);
```

- Stream interface: static iterate() method
  - iterate() start the stream from given (arg1) "seed" and calls the given UnaryOperator in infinite loop to produce infinite stream of elements.

```
Stream<Integer> strm = Stream.iterate(1, i -> i + 1).limit(10);
```

- Stream interface: static empty() method
- nio Files class: static Stream lines(filePath) method

## Stream operations

- Source of elements

```
String[] names = {"Smita", "Rahul", "Rachana", "Amit", "Shraddha", "Nilesh",  
"Rohan", "Pradnya", "Rohan", "Pooja", "Lalita"};
```

- Create Stream and display all names

```
Stream.of(names)  
    .forEach(s -> System.out.println(s));
```

- filter() -- Get all names ending with "a"
  - `Predicate<T>: (T) -> boolean`

```
Stream.of(names)  
    .filter(s -> s.endsWith("a"))  
    .forEach(s -> System.out.println(s));
```

- map() -- Convert all names into upper case
  - `Function<T,R>: (T) -> R`

```
Stream.of(names)  
    .map(s -> s.toUpperCase())  
    .forEach(s -> System.out.println(s));
```

- sorted() -- sort all names in ascending order
  - String class natural ordering is ascending order.
  - sorted() is a stateful operation (i.e. needs all element to sort).

```
Stream.of(names)
    .sorted()
    .forEach(s -> System.out.println(s));
```

- sorted() -- sort all names in descending order
  - `Comparator<T>: (T,T) -> int`

```
Stream.of(names)
    .sorted((x,y) -> y.compareTo(x))
    .forEach(s -> System.out.println(s));
```

- skip() & limit() -- leave first 2 names and print next 4 names

```
Stream.of(names)
    .skip(2)
    .limit(4)
    .forEach(s -> System.out.println(s));
```

- distinct() -- remove duplicate names
  - duplicates are removed according to equals().

```
Stream.of(names)
    .distinct()
    .forEach(s -> System.out.println(s));
```

- count() -- count number of names
  - terminal operation: returns long.

```
long cnt = Stream.of(names)
    .count();
System.out.println(cnt);
```

- collect() -- collects all stream elements into an collection (list, set, or map)

```
List<String> list = Stream.of(names)
    .collect(Collectors.toList());
// Collectors.toList() returns a Collector that can collect all stream
elements into a list
```

```
Set<String> set = Stream.of(names)
    .collect(Collectors.toSet());
// Collectors.toSet() returns a Collector that can collect all stream
elements into a set
```

- `reduce()` -- addition of 1 to 5 numbers

```
int result = Stream
    .iterate(1, i -> i+1)
    .limit(5)
    .reduce(0, (x,y) -> x + y);
```

- `max()` -- find the max string
  - terminal operation
  - See examples.

## Collect Stream result

- Collecting stream result is terminal operation.
- `Object[] toArray()`
- `R collect(Collector)`
  - `Collectors.toList()`, `Collectors.toSet()`, `Collectors.toCollection()`, `Collectors.joining()`
  - `Collectors.toMap(key, value)`

## Stream of primitive types

- Efficient in terms of storage and processing. No auto-boxing and unboxing is done.
- `IntStream` class
  - `IntStream.of()` or `IntStream.range()` or `IntStream.rangeClosed()` or `Random.ints()`
  - `sum()`, `min()`, `max()`, `average()`, `summaryStatistics()`,
  - `OptionalInt reduce()`.

## Platform Independence

- Java is architecture neutral i.e. can work on various CPU architectures like x86, ARM, SPARC, PPC, etc (if JVM is available on those architectures).
- Java is NOT fully platform independent. It can work on various platforms like Windows, Linux, Mac, UNIX, etc (if JVM is available on those platforms).
- Few features of Java remains platform dependent.
  - Multi-threading (Scheduling, Priority)
  - File IO (Performance, File types, Paths)
  - AWT GUI (Look & Feel)
  - Networking (Socket connection)

## Program

- Program is set of instructions given to the computer.
- Executable file is a program.
- Executable file contains text, data, rodata, symbol table, exe header.

## Process

- Process is program in execution.
- Program (executable file) is loaded in RAM (from disk) for execution. Also OS keep information required for execution of the program in a struct called PCB (Process Control Block).
- Process contains text, data, rodata, stack, and heap section.

## Thread

- Threads are used to do multiple tasks concurrently within a single process.
- Thread is a lightweight process.
- When a new thread is created, a new TCB is created along with a new stack. Remaining sections are shared with parent process.

## Process vs Thread

- Process is a container that holds resources required for execution and thread is unit of execution/scheduling.
- Each process have one thread created by default -- called as main thread.

## Process creation (Java)

- In Java, process can be created using Runtime object.
- Runtime object holds information of current runtime environment that includes number of processors, JVM memory usage, etc.
- Current runtime can be accessed using static `getRuntime()` method.

```
Runtime rt = Runtime.getRuntime();
```

- The process is created using `exec()` method, which returns the Process object. This object represents the OS process and its `waitFor()` method wait for the process termination (and returns exit status).

```
String[] args = { "/path/of/executable", "cmd-line arg1", ... };  
Process p = rt.exec(args);  
int exitStatus = p.waitFor();
```

## Multi-threading (Java)

- Java applications are always multi-threaded.
- When any java application is executed, JVM creates (at least) two threads.
  - main thread -- executes the application `main()`
  - GC thread -- does garbage collection (release unreferenced objects)

- Programmer may create additional threads, if required.

## Thread creation

- To create a thread
  - step 1: Implement a thread function (task to be done by the thread)
  - step 2: Create a thread (with above function)
- Method 1: extends Thread

```
class MyThread extends Thread {  
    @Override  
    public void run() {  
        // task to be done by the thread  
    }  
}
```

```
MyThread th = new MyThread();  
th.start();
```

- Method 2: implements Runnable

```
class MyRunnable implements Runnable {  
    @Override  
    public void run() {  
        // task to be done by the thread  
    }  
}
```

```
MyRunnable runnable = new MyRunnable();  
Thread th = new Thread(runnable);  
th.start();
```

- Java doesn't support multiple inheritance. If your class is already inherited from a super class, you cannot extend it from Thread class. Prefer Runnable in this case; otherwise you may choose any method.

```
// In Java GUI application is inherited from Frame class.  
// to create run() in the same class, you must use Runnable  
class MyGuiApplication extends Frame implements Runnable {  
    // ...  
    public void run() {  
        // ...  
    }  
}
```

```
// ...  
}
```

## start() vs run()

- run():
  - Programmer implemented code to be executed by the thread.
- start():
  - Pre-defined method in Thread class.
  - When called, the thread object is submitted to the (JVM/OS) scheduler. Then scheduler select the thread for execution and thread executes its run() method.

## Thread methods

- static Thread currentThread()
  - Returns a reference to the currently executing thread object.
- static void sleep(long millis)
  - Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
- static void yield()
  - A hint to the scheduler that the current thread is willing to yield its current use of a processor.
- Thread.State getState()
  - Returns the state of this thread.
  - State can be NEW, RUNNABLE, BLOCKED, WAITING, TIMED\_WAITING, TERMINATED
- void run()
  - If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called. If thread class extends from Thread class, this method should be overridden. The default implementation is empty.
- void start()
  - Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
- void join()
  - Waits for this thread to die/complete.
- boolean isAlive()
  - Tests if this thread is alive.

- `void setDaemon(boolean daemon);`
  - Marks this thread as either a daemon thread (true) or a user thread (false).
- `boolean isDaemon()`
  - Tests if this thread is a daemon thread.
- `long getId()`
  - Returns the identifier of this Thread.
- `void setName(String name)`
  - Changes the name of this thread to be equal to the argument name.
- `String getName()`
  - Returns this thread's name.
- `void setPriority(int newPriority)`
  - Changes the priority of this thread.
  - In Java thread priority can be 1 to 10.
  - May use predefined constants `MIN_PRIORITY(1)`, `NORM_PRIORITY(5)`, `MAX_PRIORITY(10)`.
- `int getPriority()`
  - Returns this thread's priority.
- `ThreadGroup getThreadGroup()`
  - Returns the thread group to which this thread belongs.
- `void interrupt()`
  - Interrupts this thread -- will raise `InterruptedException` in the thread.
- `boolean isInterrupted()`
  - Tests whether this thread has been interrupted.

## Daemon threads

- By default all threads are non-daemon threads (including main thread).
- We can make a thread as daemon by calling its `setDaemon(true)` method -- before starting the thread.
- Daemon threads are also called as background threads and they support/help the non-daemon threads.
- When all non-daemon threads are terminated, the Daemon threads get automatically terminated.

## Thread life cycle

- `Thread.State state = th.getState();`
- `NEW`, `RUNNABLE`, `BLOCKED`, `WAITING`, `TIMED_WAITING`, `TERMINATED`



- NEW: New thread object created (not yet started its execution).
- RUNNABLE: Thread is running on CPU or ready for execution. Scheduler picks ready thread and dispatch it on CPU.
- BLOCKED: Thread is waiting for lock to be released. Thread blocks due to synchronized block/method.
- WAITING: Thread is waiting for the notification. Waiting thread release the acquired lock.
- TIMED\_WAITING: Thread is waiting for the notification or timeout duration. Waiting thread release the acquired lock.
- TERMINATED: Thread terminates when run() method is completed, stopped explicitly using stop(), or an exception is raised while executing run().

## Synchronization

- When multiple threads try to access same resource at the same time, it is called as Race condition.
- Example: Same bank account undergo deposit() and withdraw() operations simultaneously.
- It may yield in unexpected/undesired results.
- This problem can be solved by Synchronization.
- The synchronized keyword in Java provides thread-safe access.
- Java synchronization internally use the Monitor object associated with any object. It provides lock/unlock mechanism.
- "synchronized" can be used for block or method.
- It acquires lock on associated object at the start of block/method and release at the end. If lock is already acquired by other thread, the current thread is blocked (until lock is released by the locking thread).
- "synchronized" non-static method acquires lock on the current object i.e. "this". Example:

```
class Account {  
    // ...  
    public synchronized void deposit(double amount) {  
        double newBalance = this.balance + amount;  
        this.balance = newBalance;  
    }  
    public synchronized void withdraw(double amount) {  
        double newBalance = this.balance - amount;  
        this.balance = newBalance;  
    }  
}
```

- "synchronized" static method acquires lock on metadata object of the class i.e. MyClass.class. Example:

```
class MyClass {
    private static int field = 0;
    // called by incThread
    public synchronized static void incMethod() {
        field++;
    }
    // called by decThread
    public synchronized static void decMethod() {
        field--;
    }
}
```

- "synchronized" block acquires lock on the given object.

```
// assuming that no method in Account class is synchronized.

// thread1
synchronized(acc) {
    acc.deposit(1000.0);
}

// thread2
synchronized(acc) {
    acc.withdraw(1000.0);
}
```

- Alternatively lock can be acquired using ReentrantLock since Java 5.0. Example code:

```
class Example {
    private final ReentrantLock r1 = new ReentrantLock();
    public void method() {
        r1.lock();
        try {
            // ...
        }
        finally {
            r1.unlock();
        }
    }
}
```

- Synchronized collections
  - Synchronized collections (e.g. Vector, Hashtable, ...) use synchronized keyword (block/method) to handle race conditions.

## Inter-thread communication

- wait()
  - Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
  - The current thread must own this object's monitor i.e. wait() must be called within synchronized block/method.
  - The thread releases ownership of this monitor and waits until another thread notifies.
  - The thread then waits until it can re-obtain ownership of the monitor and resumes execution.
- notify()
  - Wakes up a single thread that is waiting on this object's monitor.
  - If multiple threads are waiting on this object, one of them is chosen to be awakened arbitrarily.
  - The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object.
  - This method should only be called by a thread that is the owner of this object's monitor.
- notifyAll()
  - Wakes up all threads that are waiting on this object's monitor.
  - The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object.
  - This method should only be called by a thread that is the owner of this object's monitor.