

Core Java

Agenda

- Java NIO
- Multi-threading

Java NIO

- Java NIO (New IO) is an alternative IO API for Java.
- Java NIO offers a different IO programming model than the traditional IO APIs.
- Since Java 7.
- Java NIO enables you to do non-blocking (not fully) IO.
- Java NIO consist of the following core components:
 - Channels e.g. FileChannel, ...
 - Buffers e.g. ByteBuffer, ...
 - Selectors
- Java NIO also provides "helper" classes Paths & Files.
 - exists()
 - ...

Paths and Files

- A Java Path instance represents a path in the file system. A path can point to either a file or a directory. A path can be absolute or relative.

```
Path path = Paths.get("c:\\data\\myfile.txt");
```

- Files class (Files) provides several static methods for manipulating files in the file system.

```
static InputStream newInputStream(Path, OpenOption...) throws IOException;
static OutputStream newOutputStream(Path, OpenOption...) throws IOException;
static DirectoryStream<Path> newDirectoryStream(Path) throws IOException;
static Path createFile(Path, attribute.FileAttribute<?>...) throws
IOException;
static Path createDirectory(Path, attribute.FileAttribute<?>...) throws
IOException;
static void delete(Path) throws IOException;
static boolean deleteIfExists(Path) throws IOException;
static Path copy(Path, Path, CopyOption...) throws IOException;
static Path move(Path, Path, CopyOption...) throws IOException;
static boolean isSameFile(Path, Path) throws IOException;
static boolean isHidden(Path) throws IOException;
static boolean isDirectory(Path, LinkOption...);
static boolean isRegularFile(Path, LinkOption...);
static long size(Path) throws IOException;
```

```
static boolean exists(Path, LinkOption...);
static boolean isReadable(Path);
static boolean isWritable(Path);
static boolean isExecutable(Path);
static List<String> readAllLines(Path) throws IOException;
static Stream<String> lines(Path) throws IOException;
```

Channels and Buffers

- All IO in NIO starts with a Channel. A Channel is similar to IO stream. From the Channel data can be read into a Buffer. Data can also be written from a Buffer into a Channel.

NIO Channels

- Java NIO Channels are similar to IO streams with a few differences:
 - You can both read and write to a Channels. Streams are typically one-way (read or write).
 - Channels can be read and written asynchronously (non-blocking).
 - Channels always read to, or write from, a Buffer.
- Channel Examples
 - FileChannel
 - DatagramChannel // UDP protocol
 - SocketChannel, ServerSocketChannel // TCP protocol

NIO Buffers

- A buffer is essentially a block of memory into which you can write data, which you can then later read again. This memory block is wrapped in a NIO Buffer object, which provides a set of methods that makes it easier to work with the memory block.
- Using a Buffer to read and write data typically follows this 4-step process:
 - Write data into the Buffer
 - Call `buffer.flip()`
 - Read data out of the Buffer
 - Call `buffer.clear()` or `buffer.compact()`
- Buffer Examples
 - ByteBuffer
 - CharBuffer
 - DoubleBuffer
 - FloatBuffer
 - IntBuffer
 - LongBuffer
 - ShortBuffer

Channel and Buffer Example

```
RandomAccessFile aFile = new RandomAccessFile("somefile.txt", "rw");
FileChannel inChannel = aFile.getChannel();
```

```
ByteBuffer buf = ByteBuffer.allocate(32);

int bytesRead = inChannel.read(buf); // write data into buffer (from channel)
while (bytesRead != -1) {
    System.out.println("Read " + bytesRead);
    buf.flip(); // switch buffer from write mode to read mode

    while(buf.hasRemaining()){
        System.out.print((char) buf.get()); // read data from the buffer
    }

    buf.clear(); // clear the buffer
    bytesRead = inChannel.read(buf);
}
aFile.close();
```

RandomAccessFile

- RandomAccessFile class from java.io package.
- Capable of reading and writing into a file (on a storage device).
- Internally maintains file read/write position/cursor.
- Homework: Read docs.

Java NIO vs Java IO

- IO: Stream-oriented
- NIO: Buffer-oriented
- IO: Blocking IO
- NIO: Non-blocking IO

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)

Process vs Threads

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.