1.Úvod

Java je univerzální objektově orientovaný programovací jazyk založený na třídách a navržený tak, aby měl co nejméně implementačních závislostí. Je založené na pravidle "Write Once, Run Anywhere" (WORA), což znamená, že zkompilovaný kód v jazyce Java lze spustit na všech platformách, které jazyk Java podporují, bez nutnosti překompilování. Aplikace v jazyce Java se obvykle kompilují do ***bytecodu***, který lze spustit na libovolném virtuálním stroji Java (JVM) bez ohledu na architektuře počítače. Syntaxe jazyka Java je podobná jazykům C a C++, ale má méně nízkoúrovňových nástrojů než oba tyto jazyky. V roce 2019 byla Java podle serveru GitHub jedním z nejoblíbenějších používaných programovacích jazyků, zejména pro webové aplikace typu klient-server, s údajně 9 miliony vývojářů.

Původně byla Java vyvinuta Jamesem Goslingem ve společnosti Sun Microsystems (kterou mezitím koupila společnost Oracle) a vydána v roce 1995 jako základní součást platformy společnosti Sun Microsystems. Pokud jde o verze Java SE, Java SE 23 je nejnovější, zatímco 21, 17, 11 a 8 jsou aktuálně "verze dlouhodobé podpory" (LTS). Oracle vydává bezplatné veřejné aktualizace verze Java 8 pro vývoj a osobní použití na dobu neurčitou.

Společnost Sun podporuje čtyři edice Javy určené pro různá aplikační prostředí. Těmito platformami jsou:

- **Java Card** pro čipové karty.
- **Java Platform, Micro Edition (Java ME)** - zaměřená na prostředí s omezenými zdroji.
- **Java Platform, Standard Edition (Java SE)** - zaměřená na prostředí pracovních stanic.
- **Java Platform, Enterprise Edition (Java EE)** - zaměřená na velká distribuovaná podniková nebo internetová prostředí.

Jazyk Java je klíčovým pilířem systému Android. Ačkoli je Android, postavený na linuxovém jádře, napsaném převážně v jazyce C, sada Android SDK používá jazyk Java jako základ pro aplikace, ale nepoužívá žádný z jeho zavedených standardů. Jazyk **bytecode** podporovaný sadou Android SDK je nekompatibilní s jazykem **Java bytecode** a běží na vlastním virtuálním stroji. V závislosti na verzi systému Android je **bytecode** interpretován virtuálním strojem Dalvik nebo kompilován do nativního kódu pomocí prostředí Android Runtime.

Multitasking

Existují dva různé typy multitaskingu: založený na procesech a na vláknech:

- Multitasking založený na procesech umožňuje počítači spouštět dva nebo více programů současně.
    - Procesy jsou těžké úlohy, které vyžadují vlastní oddělené adresové prostory.
    - Komunikace mezi procesy je nákladná, omezená
    - Nákladné je i přepínání kontextu z jednoho procesu na druhý.
- Při multitaskingu založeném na vláknech je vlákno nejmenší jednotkou spustitelného kódu. To znamená, že jeden program může provádět dvě nebo více úloh současně.
    - Vlákna multitaskingu vyžadují menší režii než víceprocesní procesy.
    - Sdílejí stejný adresový prostor a kooperativně sdílejí stejný těžkotonážní proces.
    - Komunikace mezi vlákny je levná a přepínání kontextu z jednoho vlákna na druhé je méně nákladné.

Programy v Javě sice využívají prostředí pro multitasking založený na procesech, ale není pod kontrolou Javy, vícevláknový však ano.

Vlákna Javy

Vlákna mohou být v několika stavech. Může být "running" nebo "ready to run". Běžící vlákno může být "suspended", čímž se jeho činnost dočasně zastaví. Pozastavené vlákno lze poté obnovit, což mu umožní pokračovat tam, kde skončilo. Vlákno může být "blocked", když čeká na zdroj a lze ho kdykoli ukončit, čímž se jeho činnost okamžitě zastaví (terminated). Jakmile je vlákno ukončeno, nelze v něm pokračovat.

Java přiřazuje každému vláknu prioritu, která určuje, jak má být s daným vláknem zacházeno ve vztahu k ostatním. Priorita vlákna se používá k rozhodnutí, kdy se přepnout z jednoho běžícího vlákna na druhé. Pravidla, která určují, kdy dojde k přepnutí kontextu, jsou jednoduchá:

- Vlákno se může dobrovolně vzdát CPU. To se provádí explicitním odevzdáním, uspáním nebo blokováním čekajících vstupů a výstupů.
- Vlákno může být upozaděno vláknem s vyšší prioritou.

V případech, kdy o cykly procesoru soupeří dvě vlákna se stejnou prioritou, je situace poněkud komplikovanější. V operačních systémech, jako je Windows, jsou vlákna se stejnou prioritou automaticky časově rozdělena způsobem round-robin. U jiných typů operačních systémů musí vlákna se stejnou prioritou dobrovolně přenechat řízení svým kolegům. Pokud tak neučiní, ostatní vlákna nepoběží.

2. Vytváření vláken

Systémové vlákno

Instance třídy Thread a lze jej vytvořit jednoduše pomocí konstruktoru:

```
Thread thread = new Thread(Runnable);
```

Kde Runnable představuje instanci třídy s rozhraním Runnable. Místo Runnable lze také použít lambda výraz.

Jiný způsob vytváření vláken je použít factory:

```
ThreadFactory platFactory = Thread.ofPlatform().factory();

Thread pt = platFactory.newThread(()->{
          //kód vykonávaný vláknem;
     });
```

Systémová vlákna jsou spravována operačním systémem a jejich vytvoření je nákladná operace, jelikož je potřeba volat jádro OS.

Virtuální vlákno

Virtuální vlákna jsou v Javě od verze 19. Vytváří se podobným způsobem jako systémová vlákna. Při použití factory stačí nahradit ofPlatform za ofVirtual a vytvořená vlákna boudou virtuální.

```
     ThreadFactory virtualFactory = Thread.ofVirtual().factory();

Thread vt = virtualFactory.newThread(()->{
          //kód vykonávaný vláknem;
     });
```

Virtuální vlákna jsou spravována JDK a jejich vytváření je tak méně nákladné.

Takhle vytvořená virtuální i systémová vlákna lze spustit použitím

```
vlakno.start();
```

Pokud potřebujeme počkat na dokončení práce jiného vlákna tak lze použít

```
vlakno.join();
```

ThreadPool

ThreadPool slouží k optimalizaci vytváření vláken. Tak že poskytuje úlohám vlákna a po skončení úlohy je vlákno vráceno do poolu a může být přiděleno jiné úloze. Dochází tak k vytváření menšího počtu vláken.

ThreadPool může být buď fixed kdy má stanovený maximální počet vláken, která lze přidělit. Nebo cached, kdy se vlákna vytváří dle potřeby.

```
ExecutorService pool = Executors.newFixedThreadPool(3);

      for(int i =0;i<5;i++){
            pool.submit(new MyTask(i));
      }
```

Použitím

```
      pool.shutdown();
```

Dojde k ukončení poolu. Takže vlákna již nelze přidělit a volná vlákna zanikají.

Dokončení práce všech vláken lze ověřit pomocí

```
      pool.isTerminated();
```

ForkJoinPool

Využívá přístup rozděl a panuj. Úloha je nejdříve rekursivně rozdělena na nezávislé podúlohy (fork), které se paralelně zpracují. Následně jsou výsledky podúloh spojeny do jednoho výsledku celé úlohy (join).

Pro práci se používá třída `RecursiveAction` pro úlohy bez návratové hodnoty, případně třída `RecursiveTask<T>` pro úlohy s návratovou hodnotou. Výpočet je definován v metodě `compute()`.

3. Základní synchronizační nástroje v jazyce Java

Cíl: Tento handout poskytuje přehled základních synchronizačních nástrojů v jazyce Java, které slouží k řízení přístupu vláken k sdíleným zdrojům.

`synchronized` (nízká úroveň abstrakce)

- **Co to je**: Klíčové slovo pro synchronizaci metod nebo bloků kódu.
- **Jak to funguje**: Zajišťuje, že pouze jedno vlákno může současně vykonávat synchronizovanou metodu nebo blok kódu.
- **Výhody**: Jednoduché na použití, automatické řízení zámků.
- **Nevýhody**: Méně flexibilní, čekání na uvolnění zámku je blokující.

Příklad:

```
public synchronized void increment() {
    counter++;
```

```
}
```

Lock (vyšší úroveň abstrakce)

- **Co to je**: Lock poskytuje pokročilejší způsob synchronizace s větší kontrolou nad uzamykáním a uvolňováním kritických sekcí.
- **Jak to funguje**: Na rozdíl od `synchronized` musí programátor explicitně zamykat a odemykat kritické sekce.
- **Typy**: `ReentrantLock`, `ReadWriteLock`.
- **Výhody**: Možnost časově omezeného čekání, spravedlivý přístup, ruční řízení zámků.

Příklad:

```
Lock lock = new ReentrantLock();
lock.lock();
try {
    // kritická sekce
} finally {
    lock.unlock();
}
```

Synchronized bloky

- **Co to je**: Bloky kódu, které jsou synchronizované na konkrétním objektu.
- **Jak to funguje**: Umožňuje synchronizaci pouze části metody, nikoliv celé metody.

Příklad:

```
public void method() {
    synchronized(this) {
        // kritická sekce
    }
}
```

Condition (pro podmíněné synchronizace)

- **Co to je**: Používá se v kombinaci s Lock pro čekání a signalizaci mezi vlákny.
- **Jak to funguje**: Umožňuje vláknům čekat na splnění podmínky nebo signalizovat ostatním vláknům, že podmínka byla splněna.

- **Metody**: `await()`, `signal()`, `signalAll()`.

Příklad:

```
Lock lock = new ReentrantLock();
Condition condition = lock.newCondition();

lock.lock();
try {
    while (conditionNotMet) {
        condition.await();
    }
    // kritická sekce
    condition.signalAll();
} finally {
    lock.unlock();
}
```

- **Stavy objektů**:
  - **`await()`**: Vlakno čeká na splnění podmínky.
  - **`signal()`**: Uvolňuje jedno čekající vlákno.
  - **`signalAll()`**: Uvolňuje všechna čekající vlákna.

Semafory (Semaphore)

- **Co to je**: Synchronizační nástroj pro řízení přístupu k omezeným zdrojům.
- **Jak to funguje**: Semafor umožňuje určitý počet vláken přistupujících k sdíleným prostředkům, přičemž ostatní čekají na uvolnění místa.
- **Typy**: `Semaphore` a `SemaphoreFair`.

Příklad:

```
Semaphore semaphore = new Semaphore(2);  // Maximálně 2 vlákna mohou současně přistupovat k prostředku

semaphore.acquire();
try {
    // přístup k omezenému zdroji
} finally {
    semaphore.release();
}
```

Monitory

- **Co to je**: Monitor je objekt, který synchronizuje přístup k metodám a polím objektu.
- **Jak to funguje**: V každém objektu v Javě je implicitně monitor, který se používá při synchronizovaných metodách nebo blocích kódu.

Vlastnosti monitoru:

  - Objekt má zámek, který zajišťuje, že pouze jedno vlákno může vykonávat metody synchronizované na tomto objektu.
  - Používá se pro řízení přístupu k soukromým (private) polím a metodám.

Další synchronizační nástroje v Javě

- **`CountDownLatch a CyclicBarrier`**: Jsou pro synchronizaci skupin vláken
- **`Exchanger`**: Umožňuje výměnu dat mezi dvěma vlákny.
- **`ReadWriteLock a StampedLock`**: Jsou to pokročilé zámky pro optimalizaci čtení a zápisu.
- **`Phaser`**: Je to flexibilní synchronizační nástroj pro řízení fází.

Závěr: V Java poskytuje několik nástrojů pro synchronizaci vláken, které se liší úrovní abstrakce a flexibilitou. Pro základní synchronizaci mohou být vhodné `synchronized` metody a bloky, zatímco pro pokročilejší scénáře, kde je potřeba větší kontrola nad přístupem k zdrojům, mohou být využity `Lock`, `Condition` a semafory. Monitory, i když implicitní, jsou základní součástí synchronizačních mechanismů Javy.

Další možnosti synchronizace a paralelizace

Atomické proměnné

Java poskytuje i nízkoúrovňovější nástroje k synchronizaci. Konkrétně se jedná zejména o atomické proměnné. Ty jsou implementovány jako objekty, které zapouzdřují hodnotu vybraného typu a poskytují sadu atomických operací.

Základní atomické proměnné jsou **AtomicInteger**, **AtomicLong** a **AtomicBoolean**. Ty poskytují atomické operace pro jednu proměnnou typu int, long nebo boolean.

Balíček *java.util.concurrent.atomic* dále poskytuje například:

- AtomicReference
- AtomicIntegerArray
- AtomicLongArray
- AtomicReferenceArray
- AtomicIntegerFieldUpdater
- AtomicLongFieldUpdater
- AtomicReferenceFieldUpdater
- LongAdder
- LongAccumulator

Většina atomických proměnných poskytuje metody get() a set() pro atomické čtení/nastavení hodnoty. Různé třídy pak mají další různé metody. AtomicInteger, AtomicLong, AtomicBoolean a AtomicReference například poskytují:

- getAndSet(newValue)
    - Atomicky nastaví na novou hodnotu a vrátí původní
- compareAndSet(expectedValue, newValue)
    - Atomicky nastaví na novou hodnotu, pokud má očekávanou hodnotu. Vrátí true pokud došlo k nastavení hodnoty, jinak false.
- compareAndExchange(expectedValue, newValue)
    - Atomicky nastaví na novou hodnotu, pokud má očekávanou hodnotu. Vrátí průvodní hodnotu.
- weakCompareAndSet...(expectedValue, newValue)
    - Atomicky se pokusí nastavit na novou hodnotu, pokud má očekávanou hodnotu. Může ale selhat, i když tato podmínka platí. Vrátí true pokud došlo k nastavení hodnoty, jinak false.

AtomicInteger pak poskytuje například i:

- getAndIncrement()
    - Atomicky inkrementuje hodnotu a vrátí původní hodnotu
- getAndAdd(delta)
    - Atomicky zvýší hodnotu o delta a vrátí původní hodnotu
- getAndUpdate(updateFunction)
    - Atomicky nastaví hodnotu na výsledek aplikace funkce na původní hodnotu a vrátí původní hodnotu.
- getAndAccumulate(x, accumulatorFunction)
    - Atomicky nastaví hodnotu na výsledek aplikace funkce na danou hodnotu x a původní hodnotu a vrátí původní hodnotu.

```
AtomicInteger counter = new AtomicInteger();
```

```
List<Thread> threads = new ArrayList<>();
for(int i = 0; i < 20; i++){
    Thread thread = Thread.ofPlatform().factory()
            .newThread(counter::incrementAndGet);
    thread.start();
}
for(Thread thread : threads){
    try {
        thread.join();
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}
System.out.println(counter.get());
```

AtomicIntegerFieldUpdater, AtomicLongFieldUpdater a AtomicReferenceFieldUpdater fungují obdobně jako AtomicInteger, AtomicLong a AtomicReference, ale pracují na atributech jiného objektu.

LongAdder, LongAccumulator slouží k distribuci zátěže velkého množství atomických inkrementací/nastavení na více buněk (dílčích hodnot), kdy různá vlákna upravují různé bunňky.

Thread-safe kolekce

Práce s běžnými kolekcemi jako je ArrayList, LinkedList, HashMap z více vláken není bezpečná. Může dojít k nejrůznějším chybám od vyvolání výjimky (například ConcurrentModificationException) až po nekonzistentní stav samotné kolekce (například chybějící prvky).

Souběžné kolekce

Jedno možné řešení je použití souběžných kolekcí (*concurrent collections*). Ty představují upravené implementace běžných kolekcí, umožňující souběžný přístup z více vláken:

- CopyOnWriteArrayList - místo ArrayList
- ConcurrentHashMap - místo HashMap
- CopyOnWriteArraySet - místo ArraySet
- LinkedBlockingQueue - místo LinkedList
- ConcurrentSkipListMap - místo TreeMap

Protože splňují stejné rozhraní jako běžné kolekce (například List<T>), je možné je používat stejným způsobem. Umožňují bezpečně přidávat, odstraňovat a vyhledávat prvky, iteraci na prvcích, používání proudů. Přístup (například při čtení) je skutečně

souběžný – tedy v jednu chvíli čte více vláken. V některých případech, například u ConcurrentHashMap je možný i souběžný zápis.

```
Random random = new Random();
List<Integer> list = new CopyOnWriteArrayList<>();
Thread.ofPlatform().factory()
        .newThread(() ->
        {
            while(true) {
                list.add(random.nextInt(Integer.MAX_VALUE));
            }
        }).start();

Thread.ofPlatform().factory()
        .newThread(() ->
        {
            while(true) {
                for(int i : list){
                    System.out.println(i);
                }
            }
        }).start();
```

Synchronizované kolekce

Alternativní možností jsou synchronizované kolekce. Ty lze vytvořit z běžných kolekcí pomocí statických metod třídy Collections:

- Collections.synchronizedCollection
- Collections.synchronizedList
- Collections.synchronizedMap
- Collections.synchronizedSet

Takto vzniklé kolekce je pak možné bezpečně používat pro přidávání, odstraňování a vyhledávání prvků. Výhodou také je, že metody přijímají libovolné instance rozhraní List, Map, Set, Collection. Collections.synchronizedList by tedy šlo použít jak s instancí ArrayList, tak LinkedList nebo i instancí kolekce třetí strany (například z knihovny).

Má to ale i několik nedostatků. Iterace a používání proudů není bezpečné – je třeba ručně vzniklou synchronizovanou kolekci zamknout pomocí synchronized. Dále je zakázáno přímo modifikovat původní běžnou kolekci.

Ve skutečnosti samotná synchronizovaná kolekce hojně používá synchronized bloky. Jedná se totiž o vlastně třídu s vhodným rozhraním (například List<T>), která si uchová

původní běžnou kolekci a ve svých metodách obsahuje synchronized bloky na společném objektu, uvnitř kterých volá metody původní kolekce. Z toho plyne také, že synchronizované kolekce jsou pomalejší, protože i při čtení dojde k zamknutí pro všechna ostatní vlákna – tedy číst může jen jedno vlákno zároveň.

```
Random random = new Random();
List<Integer> list = Collections.synchronizedList(new ArrayList<>());
Thread.ofPlatform().factory()
        .newThread(() ->
        {
            while(true) {
                list.add(random.nextInt(Integer.MAX_VALUE));
            }
        }).start();

Thread.ofPlatform().factory()
        .newThread(() ->
        {
            while(true) {
                synchronized (list){
                    for(int i : list){
                        System.out.println(i);
                    }
                }
            }
        }).start();
```

Paralelní proudy

Proudy vytvořené pomocí Collection.stream nebo Stream.of jsou sekvenční. Java ale umožňuje vytváření paralelních proudů ze sekvenčních, a naopak pomocí Stream.parallel a Stream.sequential. Alternativně lze v případě kolekcí použít Collections.parallelStream (specifikace však povoluje vrácení sekvenčního proudu).

Paralelní proudy vzniknou rozdělením původního proudu na více podproudů, které jsou zpracovávány více vlákny. Na pozadí dochází k používání ForkJoinPool. Použitá vlákna jsou z ForkJoinPool.commonPool, jehož velikost závisí na počtu jader CPU. Použití parallel() nevede vždy k paralelnímu výpočtu a lepšímu výkonu. Záleží na použitých operacích a datech.

```
List<Long> numbers = LongStream.range(0,100000).boxed().toList();
List<Long> squares = numbers.stream().parallel()
        .map(i -> i * i).toList();
```

Možnosti Distribuovaných Systémů v Javě

Java má nepřeberné množství knihoven a frameworků, které lze využít při vývoji na distribuovaných systémech.
V Maven Respository je publikováno přes 15 000 000 balíčků (2024); mezi nimi můžeme najít i knihovny pro práci se síťovými protokoly a distribuovanými systémy. Zde je např. HTTP Client https://repo1.maven.org/maven2/HTTPClient/HTTPClient/0.3-3/.

Remote Method Invocation (RMI)

Jedním z tradičních přístupů pro komunikaci mezi vzdálenými aplikacemi je Remote Method Invocation (RMI), který umožňuje objektům volat metody na vzdálených objektech. Tento přístup vychází z principu Remote Procedure Call (RPC) a umožňuje, aby **objekty běžící v jedné instanci JVM (Java Virtual Machine) komunikovaly s objekty v jiné instanci JVM**, i když se nachází na jiném serveru.

Java Message Service (JMS) (nyní Jakarta Messaging)

Další významnou technologií pro distribuované systémy je Java Message Service (JMS), nyní známá jako Jakarta Messaging. JMS představuje standard pro asynchronní komunikaci mezi aplikacemi pomocí zasílání zpráv. Tento přístup umožňuje jednotlivým komponentám aplikace vytvářet, odesílat, přijímat a číst zprávy nezávisle na jejich umístění. JMS podporuje dva základní modely:

- Point-to-Point (P2P): Zpráva má vždy jednoho příjemce.
- Publish/Subscribe (Pub/Sub): Zprávy jsou zasílány všem přihlášeným odběratelům.

Populární implementací JMS je Open Message Queue (OpenMQ), která poskytuje robustní platformu pro asynchronní messaging a škálovatelnost. Tato technologie se hodí zejména tam, kde je důležité odolnost vůči výpadkům a potřeba zpracovávat velký objem zpráv.

RESTful a SOAP Služby

Pro komunikaci mezi aplikacemi, zejména v prostředí mikroservis, jsou často využívány RESTful služby a SOAP služby.

RESTful služby využívají architektonický styl založený na HTTP metodách (GET, POST, PUT, DELETE) a umožňují snadné vytváření API. Pro vývoj REST API v Javě se používá

framework JAX-RS s implementacemi jako Jersey nebo RESTEasy, které usnadňují návrh a správu REST služeb. REST je oblíbený díky své jednoduchosti a vysoké interoperabilitě mezi různými platformami.

Na druhou stranu, SOAP služby používají XML pro strukturovanou komunikaci mezi systémy a zajišťují vysokou míru bezpečnosti a spolehlivosti. V Javě se pro vývoj SOAP služeb využívá framework JAX-WS, který umožňuje snadno implementovat SOAP-based webové služby. SOAP je často preferován v enterprise prostředí, kde jsou klíčové transakční vlastnosti a bezpečnost.

Spring Boot a Mikroservisní Architektura

V moderním vývoji se čím dál tím více prosazuje Spring Boot, což je rozsáhlý framework pro tvorbu robustních aplikací v Javě. Spring Boot podporuje rychlý vývoj aplikací založených na mikroservisní architektuře, které umožňují modularitu, škálovatelnost a snadnou správu.

Spring Boot podporuje různé komunikační technologie jako REST API, SOAP, a WebSocket pro real-time komunikaci. Navíc ve spojení s Spring Cloud umožňuje využívat koncepty jako Service Discovery (automatické objevování služeb), Load Balancing (vyvažování zátěže) a další nástroje potřebné pro správu mikroservis. Pro orchestraci a nasazování kontejnerizovaných aplikací lze Spring Boot integrovat s Kubernetes, což usnadňuje volbu lídra a správu distribuovaných aplikací.

Dalším užitečným nástrojem pro zvýšení odolnosti aplikací je Resilience4j, který poskytuje funkce jako Circuit Breaker, což pomáhá předcházet selhání celého systému v případě problémů s některými službami.

JGroups a Hazelcast

Pro potřeby komunikace v distribuovaných systémech a synchronizaci stavů aplikací se často používá JGroups. Tento nástroj umožňuje aplikacím vytvářet clustery, synchronizovat svůj stav a automaticky volit lídra v případě změn v síti. To je užitečné zejména pro systémy, které vyžadují vysokou dostupnost a škálovatelnost.

Dalším významným nástrojem pro práci s distribuovanými daty je Hazelcast, který poskytuje platformu pro distribuované kešování, sdílené datové úložiště a paralelní zpracování dat. Hazelcast umožňuje realizovat distribuované výpočty pomocí techniky MapReduce, což zvyšuje efektivitu práce s velkým objemem dat a umožňuje aplikacím škálovat horizontálně.

Open-source knihovny a frameworky

Zde se můžete podívat, pokud chcete, na zdrojové kódy knihoven a frameworků zmíněné v této kapitole.

- https://github.com/hazelcast/hazelcast
- https://github.com/belaban/JGroups

- https://github.com/spring-projects/spring-boot
- https://github.com/jakartaee/messaging
- https://github.com/javaee/metro-jax-ws

Reference

[1] *(Wikipedia) Java (programming language)*

https://en.wikipedia.org/wiki/Java_(programming_language)

[2] *(Wikipedia) Java version history*

https://en.wikipedia.org/wiki/Java_version_history

[3] *Herbert Schildt (2011) - Java The Complete Reference, 8th Edition*

[4] *(Baeldung) Life Cycle of a Thread in Java*

https://www.baeldung.com/java-thread-lifecycle

[5] (Oracle) Thread.State

https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.State.html

[6] *(Oracle) Class Thread*

https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

[7] *(Baeldung) Priority of a Thread in Java*

https://www.baeldung.com/java-thread-priority

[8] *(Oracle) Package java.util.concurrent.atomic*

https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/concurrent/atomic/package-summary.html

[9] *(Baeldung) An Introduction to Atomic Variables in Java*

https://www.baeldung.com/java-atomic-variables

[10] *(Baeldung) An Introduction to Synchronized Java Collections*

https://www.baeldung.com/java-synchronized-collections