Multithreading

Why Java is multi-threaded

- When we start execution of Java application then JVM starts execution of main thread and garbage collector(GC). Due to these two threads every java application is multithreaded.
    - Main Thread
        - It is user thread / non daemon thread.
        - It is responsible for calling/invoking main method.
        - Its default priority in 5(In Java, Thread.NORM_PRIORITY).
    - Garbage Collector / Finalizer
        - It is daemon thread / backgrounf thread.
        - It is responsible for releasing / reclaiming / deallocating memory of unused(whose reference count is 0) instances.
        - Before releasing memory of unused instance, GC invoke finalize(similar to destructor in C++) method on instance.
        - Its default priority in 8(In Java, Thread.NORM_PRIORITY + 3).
- If we want to use non java resource into Java then we must use Java Native Interface framework.
- Thread is non Java resource and to access it we require JNI. But Java application developer need not to worry avout it. Because SUN/ORACLE developers has already written logic to access OS thread in Java. In other words, Java has built-in support to thead. Hence java is considered as multithreaded programming langauge.

Runnable

- It is a functional interface declared in java.lang pckage.
- "void run( )" is a method of java.lang.Runnable interface. In the context of multi-threading, run() method is called as business logic method.
- If we want to create thread in Java then we should use Runnable interface.

```
class Task implements Runnable{
  @Override
  public void run() {
    System.out.println("Hello from run method()");
  }
}
public class Program {
  public static void main(String[] args) {
    Runnable target = new Task();   //Upcasting
    Thread th = new Thread(target);
    th.start();
  }
}
```

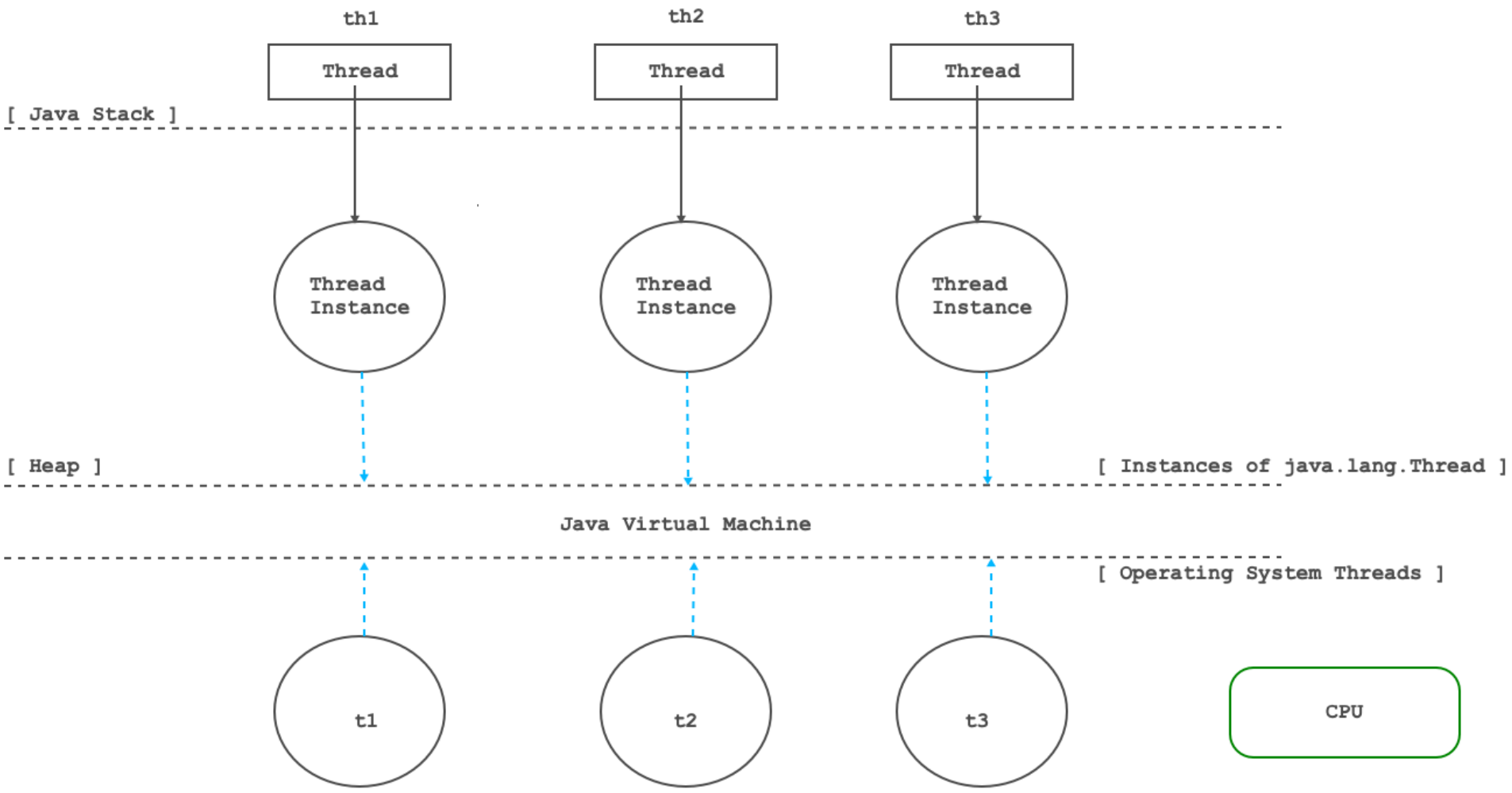
Thread

- java.lang.Thread class is a sub class of java.lang.Object class and it implements java.lang.Runnable interface.

```
public class Thread
  extends Object
  implements Runnable
```

- Thread is non Java resource. In other words, it is unmanaged resource. Hence developer must take care of its creation as well as termination / dispose.

- Instance of java.lang.Thread is not a operating system thread. Rather it represents operating system thread.



- If we want to utilize hardware resources efficiently then we should use thread.
- Method(s) of java.lang.Object class:
    - public String toString();
    - public boolean equals(Object obj);
    - public native int hashCode( );
    - protected native Object clone( )throws CloneNotSupportedException
    - protected void finalize()throws Throwable
    - public final native Class<?> getClass();
    - public final void wait( )throws InterruptedException
    - public final native void wait(long timeOut)throws InterruptedException
    - public final void wait(long timeOut, int nanos)throws InterruptedException
    - public final void notify();
    - public final void notifyAll();
- Method(s) of java.lang.Runnable interface
    - void run( );
- Nested type of java.lang.Thread class
    - java.lang.Thread.State is nested enum defined inside java.lang.Thread class.

```
package java.lang;
public class Thread extends Object implements Runnable{
  public static enum State{
    NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED;
  }
}
```

    - ajava.lang.Thread.UncaughtExceptionHandler is nested functional interface defined inside java.lang.Thread class.

```
package java.lang;
public class Thread extends Object implements Runnable{
  @FunctionalInterface
  public static interface UncaughtExceptionHandler {
      void uncaughtException(Thread t, Throwable e);
  }
}
```

- Fields of java.lang.Thread class
    - public static final int MIN_PRIORITY = 1; //Thread.MIN_PRIORITY

- public static final int NORM_PRIORITY = 5; //Thread.NORM_PRIORITY
- public static final int MAX_PRIORITY = 10; //Thread.MAX_PRIORITY

- Constructor Summary of java.lang.Thread class:
  - public Thread();

```
Thread th = new Thread( );
```

  - public Thread(String name);

```
Thread th = new Thread( "User Thread-1" );
```

  - public Thread(Runnable target);

```
Runnable target = new Task();
Thread th = new Thread( target );
```

  - public Thread(Runnable target, String name);

```
Runnable target = new Task();
Thread th = new Thread( target, "User Thread-1" );
```

  - public Thread(ThreadGroup group, Runnable target);

```
ThreadGroup group = new ThreadGroup("CDAC-Thread-Group");
Runnable target = new Task();
Thread th = new Thread( group, target );
```

  - public Thread(ThreadGroup, String name);

```
ThreadGroup group = new ThreadGroup("CDAC-Thread-Group");
Thread th = new Thread( group,  "User Thread-1");
```

  - public Thread(ThreadGroup group, Runnable target, String name);

```
ThreadGroup group = new ThreadGroup("CDAC-Thread-Group");
Runnable target = new Task();
Thread th = new Thread( group, target, "User Thread-1");
```

  - public Thread(ThreadGroup group, Runnable targer, String name , long stackSize) ;

```
ThreadGroup group = new ThreadGroup("CDAC-Thread-Group");
Runnable target = new Task();
Thread th = new Thread( group,  target, "User Thread-1", 0 );
```

- In Java, we can create thread using java.lang.Thread class.

```
class Task extends Thread{
  @Override
  public void run() {
    System.out.println("Hello from run method");
  }
}
public class Program {
  public static void main(String[] args) {
    Thread th = new Task( );    //Upcasting
    th.start();
  }
}
```

- Method Summary of java.lang.Thread class
  - public static Thread currentThread()
  - public long getId()
  - public final String getName()
  - public final void setName(String name)
  - public final int getPriority()
  - public final void setPriority(int newPriority)
  - public Thread.State getState()
  - public final ThreadGroup getThreadGroup()
  - public Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()
  - public static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)
  - public void interrupt()
  - public static boolean interrupted()
  - public boolean isInterrupted()
  - public final boolean isAlive()
  - public final boolean isDaemon()
  - public final void join() throws InterruptedException
  - public final void setDaemon(boolean on)
  - public static void sleep(long millis) throws InterruptedException
  - public static void yield()
- Getting information of main thread

```
public static void main(String[] args) {
  Thread thread = Thread.currentThread();
  System.out.println( thread.toString() );  //Thread[main,5,main]
  //main :  thread's name
  //5 : priority
  //main : thread group.
}
```

```
public static void main(String[] args) {
    Thread thread = Thread.currentThread();

    String name = thread.getName();
    System.out.println("Thread Name     :   "+name);

    int priority= thread.getPriority();
    System.out.println("Thread Priority     :   "+priority);

    ThreadGroup group = thread.getThreadGroup();
    System.out.println("Thread Group        :   "+group.getName());

    State state = thread.getState();
    System.out.println("Thread State        :   "+state.name());

    boolean type = thread.isDaemon();
    System.out.println("Thread Type     :   "+( type ? "Daemon Thread" : "User Thread") );

    boolean status = thread.isAlive();
    System.out.println("Thread Status       :   "+( status ? "Alive" : "Dead") );
}
```

- Output is:

```
  Thread Name       :   main
  Thread Priority       :   5
  Thread Group      :   main
  Thread State      :   RUNNABLE
  Thread Type       :   User Thread
  Thread Status     :   Alive
```

- Getting information of garbage collector

```
class Sample{
  public Sample( ) {
    System.out.println("Inside constructor of "+this.getClass().getSimpleName());
  }
  public void print( ) {
    System.out.println("Inside print method of "+this.getClass().getSimpleName());
  }
  @Override
  protected void finalize() throws Throwable {
    System.out.println("Inside finalize method of "+this.getClass().getSimpleName());

    Thread thread = Thread.currentThread();

    String name = thread.getName();
    System.out.println("Thread Name       :   "+name);

    int priority= thread.getPriority();
    System.out.println("Thread Priority       :   "+priority);

    ThreadGroup group = thread.getThreadGroup();
    System.out.println("Thread Group      :   "+group.getName());

    State state = thread.getState();
    System.out.println("Thread State      :   "+state.name());

    boolean type = thread.isDaemon();
    System.out.println("Thread Type       :   "+( type ? "Daemon Thread" : "User Thread") );

    boolean status = thread.isAlive();
    System.out.println("Thread Status     :   "+( status ? "Alive" : "Dead") );
  }
}
```

```
  public static void main1(String[] args) {
  Sample sample = null;
  sample = new Sample();
  sample.print();
  sample = null;
  //System.gc();  //or
  Runtime.getRuntime().gc();
}
```

- Output is

```
Thread Name     :   Finalizer
Thread Priority     :   8
Thread Group        :   system
Thread State        :   RUNNABLE
Thread Type     :   Daemon Thread
Thread Status       :   Alive
```

What do you know about final/finally/finalize?

- final
  - It is keyword, we can use with local variable, field, method and class.
    - If we use final modifier with method local variable and field then it is considered as constant.
    - If we use final modifier with method then we can not override/redefine method inside sub class
    - If we use final modiifer with class then we can not extend it.
- finally
  - It is a block that we can use with try/catch.
    - If we want to close local resources then we should use finally block.

```
public static void main(String[] args) {
  Scanner sc = null;  //Method local variable
  try{
    sc = new Scanner( System.in );
    //TODO
  }catch( Exeption ex ){
    ex.printStackTrace();
```

```
    }finally{
      sc.close(); //Releasing local resource
    }
  }
```

- finalize
  - It is non final method of java.lang.Object class.
    - If we want to release class level(which is declared as field) resources then we should use finalize method.

```
class Sample{
  private Scanner sc; //Field
  public Sample( ) {
    this.sc = new Scanner( System.in);
  }
  //TODO
  @Override
  protected void finalize() throws Throwable {
    this.sc.close();
  }
}
public class Program {
  public static void main(String[] args) {
    Sample sample = null;
    sample = new Sample();
    sample.print();
    sample = null;
    System.gc
  }
}
```

Thread Creation in Java

- In Java, we can create thread using 2 ways:
  - By implementing Runnable interface
  - By extending Thread class.

Thread Creation in by implementing Runnable interface

- If we create Thread instance but do not call start() method on it then is considered in NEW state.

```
Runnable target = new Task();   //Upcasting
Thread th = new Thread(target);   //NEW
```

- To start execution of thread we should start() method.
- If we call start() method on thread instance then thread is considered in RUNNABLE state.

```
Runnable target = new Task();   //Upcasting
Thread th = new Thread(target);   //NEW
th.start(); //RUNNABLE
```

- If we call start() method on already started thread then start() method throws IllegalThreadStateException

```
Runnable target = new Task();   //Upcasting
Thread th = new Thread(target);   //NEW
th.start(); //RUNNABLE
th.start(); //IllegalThreadStateException
```

- When control come out of run method then thread gets terminated. In this case thread is considered in TERMINATED state.
- In following cases, control can come out of run method and thead can terminate:
  - Successful completion of run method.
  - Getting exception during execition of run method.
  - Execution of jump statement(return statement) inside run method.
- If we want to suspend execution of running thread then we should use sleep() method. It is a static method of java.lang.Thread class.

```
public static void sleep(long millis) throws InterruptedException
```

  - millis - the length of time to sleep in milliseconds
    - 1000 millisecond = 1 second
  - Usage:

```
Thread.sleep( 250 )
```

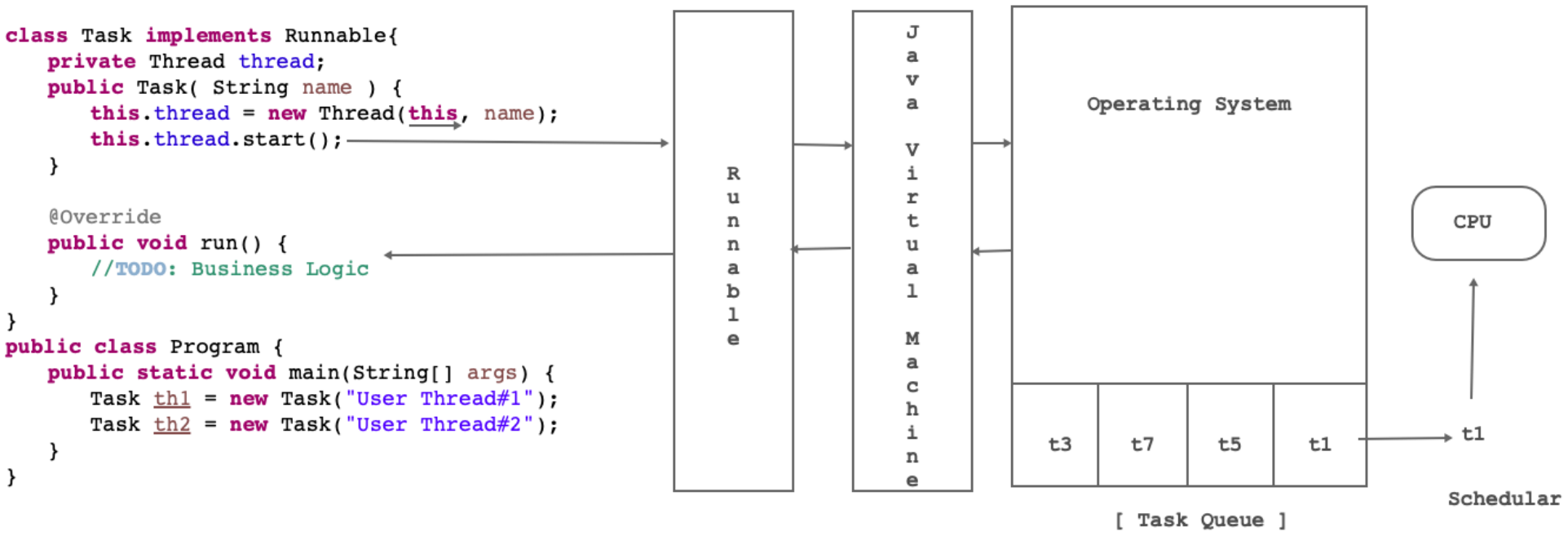
- Following methods calls are considered as blocking calls:
  - sleep()
  - suspend()
  - join()
  - wait()
  - Input operations in run method
- Generally we should use blocking calls very carefully.
- If we want to group functionally realated threads together then we should use ThreadGroup.

```
public static void main(String[] args) {
  Runnable target = new Task();

  ThreadGroup threadGroup = new ThreadGroup("acts");

  Thread th1 = new Thread(threadGroup, target);
  th1.setName("User Thread#1");
  th1.start();

  Thread th2 = new Thread(threadGroup, target);
  th2.setName("User Thread#2");
  th2.start();
}
```

- Thread creation using Runnable

```
class Task implements Runnable{
  private Thread thread;
  public Task( String name ) {
    this.thread = new Thread(this, name);
    this.thread.start();
  }

  @Override
  public void run() {
    //TODO: Business Logic
  }
}
public class Program {
  public static void main(String[] args) {
    Task th1 = new Task("User Thread#1");
    Task th2 = new Task("User Thread#2");
  }
}
```

- what is the relation between start and run method?



  - start() method do not call run() method.
  - If we call start() method on java thread instance then it is considered as request to the JVM to get thread from operating system and to map it with java thread instance.
  - Schedular is responsible for picking thread from queue and assiging CPU to it. When schedular assign CPU to the OS thread then JVM invoke run method on instance who has implemented java.lang.Runnable interface.

Thread Creation in by extending Thread class

```
class Task extends Thread{
  public Task( String name ) {
    super(name);
    this.start();
  }
  @Override
  public void run() {
    //TODO : Business logic
  }
}
public class Program {
  public static void main(String[] args) {
    Task th1 = new Task("User Thread#1");
  }
}
```

What will happen if we call run method instead of start method on thread instance

```
class Task extends Thread{
  public Task( String name ) {
    super( name );
  }
  @Override
  public void run() {
    System.out.println("Inside run method : "+Thread.currentThread().getName());
  }
}
public class Program {
  public static void main(String[] args) {
    Thread thread = new Task( "Thread#1");
    //thread.start();   //Inside run method : Thread#1
    thread.run();//Inside run method : main
  }
}
```

- If we call start() method on thread instance then JVM starts execution of new Thread. But if we call run() method on thread instance then JVM do not start execution of new Thread. In above code, main therad is calling main method and main method is calling run() method. In directly main thread is calling run() method.

What is the difference between creating thread using Runnable and Thread class

Runnable

- Consider Thread creation using Runnable interface:

```
class A implements Runnable{
  private Thread thread;
  public A( ){
    this.thread = new Thread( this);
    this.thread.start();
  }
  @Override
  public void run( ){
    //TODO: Write business logic here
  }
}
```

  - Here class can still extend another class.
- Let us try to create sub class of class A

```
class B extends A{
  public B( ){
  }
  @Override
  public void run( ){
    //TODO: Write business logic here
  }
}
```

```
class C extends B{
  public C( ){
  }
  @Override
  public void run( ){
    //TODO: Write business logic here
  }
}
```

```
class D extends C{
public D( ){
}
@Override
public void run( ){
  //TODO: Write business logic here
}
}
```

- If create instance of class B then first A class and then B class constructor will call. In class A, Thread registration process will start and when OS thread will get CPU then B class run method will call. Same is the case of class C and D. In simple words, If class implement Runnable interface then that class and all its sub classes must participate in Threading behavior.

Thread

- Consider Thread creation by extending Thread class:

```
class A extends Thread{
  public A( ){
    this.start();
  }
  @Override
  public void run( ){
    //TODO: Write business logic here
  }
}
```

  - In Java, class can extend more than one class. So here we can not extend another class / if class already sub class of another class then we can not extend Thread.

- Let us try to create sub class of class A

```
class B extends A{
  public B( ){
  }
  @Override
  public void run( ){
    //TODO: Write business logic here
  }
}
```

```
class C extends B{
  public C( ){
  }
  @Override
  public void run( ){
    //TODO: Write business logic here
  }
}
```

```
class D extends C{
  public D( ){
  }
  @Override
  public synchronized void start() {
    //Keep Empty
  }
}
```

  - In above code class A, B and C must participate into Threading behavior. But by overriding start( ) method class D can come out of threading behavior.

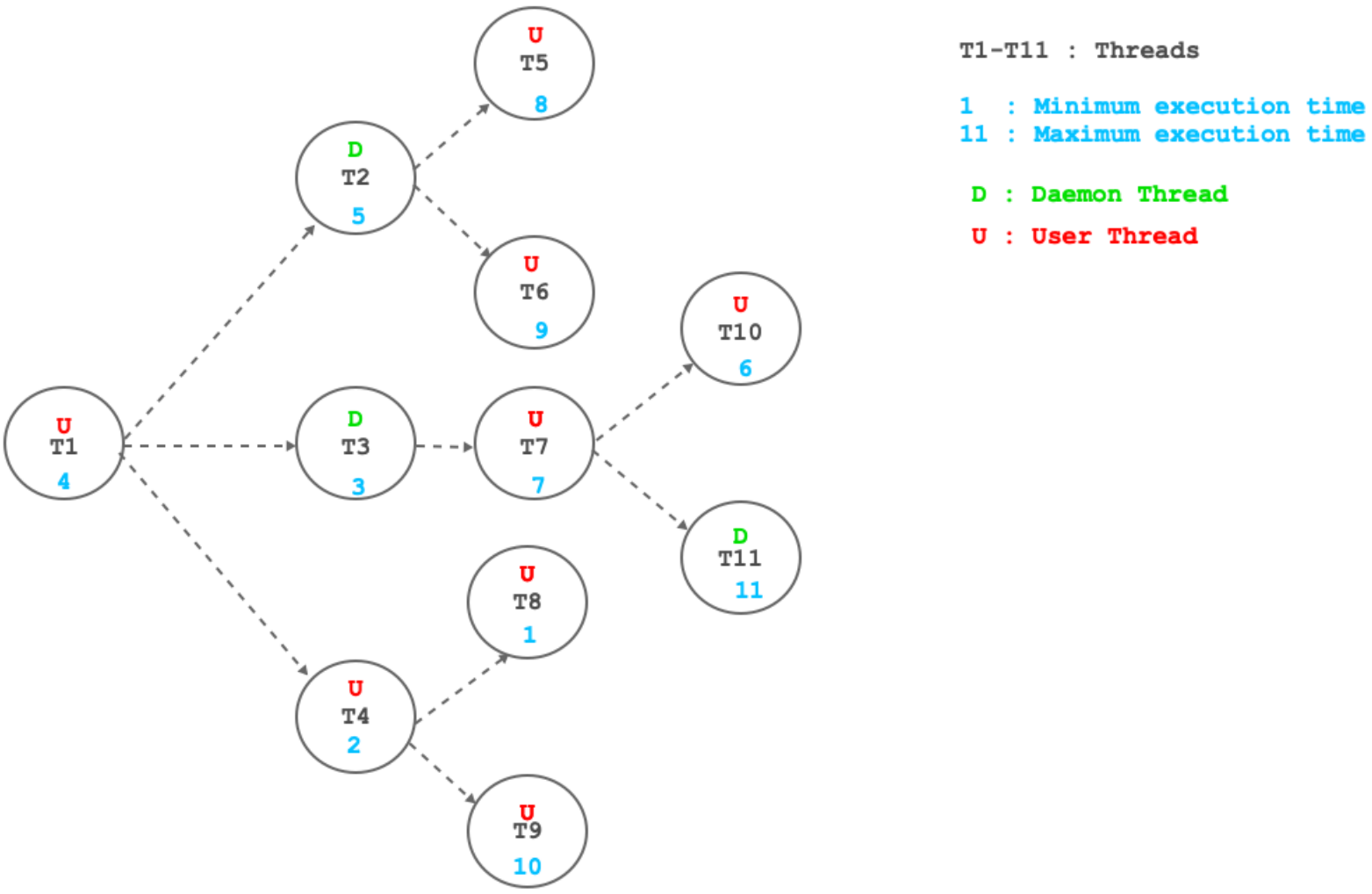
Types of Thread in java:

- User Thread
  - It is also called as non daemon thread.
  - If we create any thread from main method / another user thread then it is by default considered as user thread.
  - Default Properties of main thead:
    - Type: User Thread
    - Priority: 5
    - Thread group: main
- Daemon Thread
  - It is also called as background thread.
  - If we create any thread from finalze method / another deamon thread then it is by default considered as daemon thread.
  - Default Properties of GC thead:
    - Type: Daemon Thread
    - Priority: 8
    - Thread group: system
- Using "public final boolean isDaemon()" method we can check type of thread.
- Using "public final void setDaemon(boolean on)" method we can marks thread as either a daemon thread or a user thread.

```
Thread th = new Thread( target, name );
th.setDaemon( true ); //Not thread will be considered as Daemon thread.
th.start();
```

What is the difference between User Thread and Daemon Thread

- User threads have higher priority than daemon threads, which means that the JVM will keep them running as long as the application is running. Daemon threads, on the other hand, have lower priority, and the JVM will stop them as soon as all user threads have terminated.



interrupt() / interrupted() / isInterrupted()

- An interrupt() is a notification sent to a thread to request that it should stop what it's doing and do something else.

```
  public void interrupt()
```

```
class Task implements Runnable {
  @Override
  public void run() {
    for (int count = 1; count <= 100; ++count) {
        System.out.println("Count   :   " + count);
        if( count == 50 )
          Thread.currentThread().interrupt();
    }
  }
}
```

- When the interrupt signal is sent to the thread, it will set an interrupted flag on the thread. We can check that flag using isInterrupted() method.

```
  public boolean isInterrupted();
```

```
for (int count = 1; count <= 100; ++count) {
  if( !Thread.currentThread().isInterrupted()) {
    System.out.println("Count   :   " + count);
    if( count == 50 ) {
      Thread.currentThread().interrupt();
    }
  }
}
```

- isInterrupted() method only check interrupted status but do not reset the status.
- Using interrupted() method we can check interrupted status but it reset interrupted status. In other words, if this method were to be called twice in succession, the second call would return false

```
public static boolean interrupted()
```

```
class Task implements Runnable {
  @Override
  public void run() {
    for (int count = 1; count <= 100; ++count) {
      if (!Thread.currentThread().isInterrupted()) {
        System.out.println("Count :   " + count);
        if (count == 50) {
          Thread.currentThread().interrupt();
        }
      }
    }

    while (!Thread.interrupted()) {
      for (int count = 100; count <= 120; ++count)
        System.out.println("Count :   " + count);
    }
  }
}

public class Program {
  public static void main(String[] args) throws Exception {
    Thread thread = new Thread(new Task());
    thread.start();
  }
```

Thread Priority

- OS schedular is responsible for assigning CPU to the thread.
- On the basis of thread priority schedular assign CPU to the thread.
- Thread priorities of Java and OS are different.
- Thread priorities in Java:
  - Thread.MIN_PRIORITY = 1;
  - Thread.NORM_PRIORITY = 5;
  - Thread.MAX_PRIORITY = 10;
- How will you read thread priority in Java?

```
public static void main(String[] args) {
    Thread thread = Thread.currentThread();
    int priority = thread.getPriority();
    System.out.println("Priority of "+thread.getName()+" is "+priority);
}
```

- How will you set thread priority in Java?

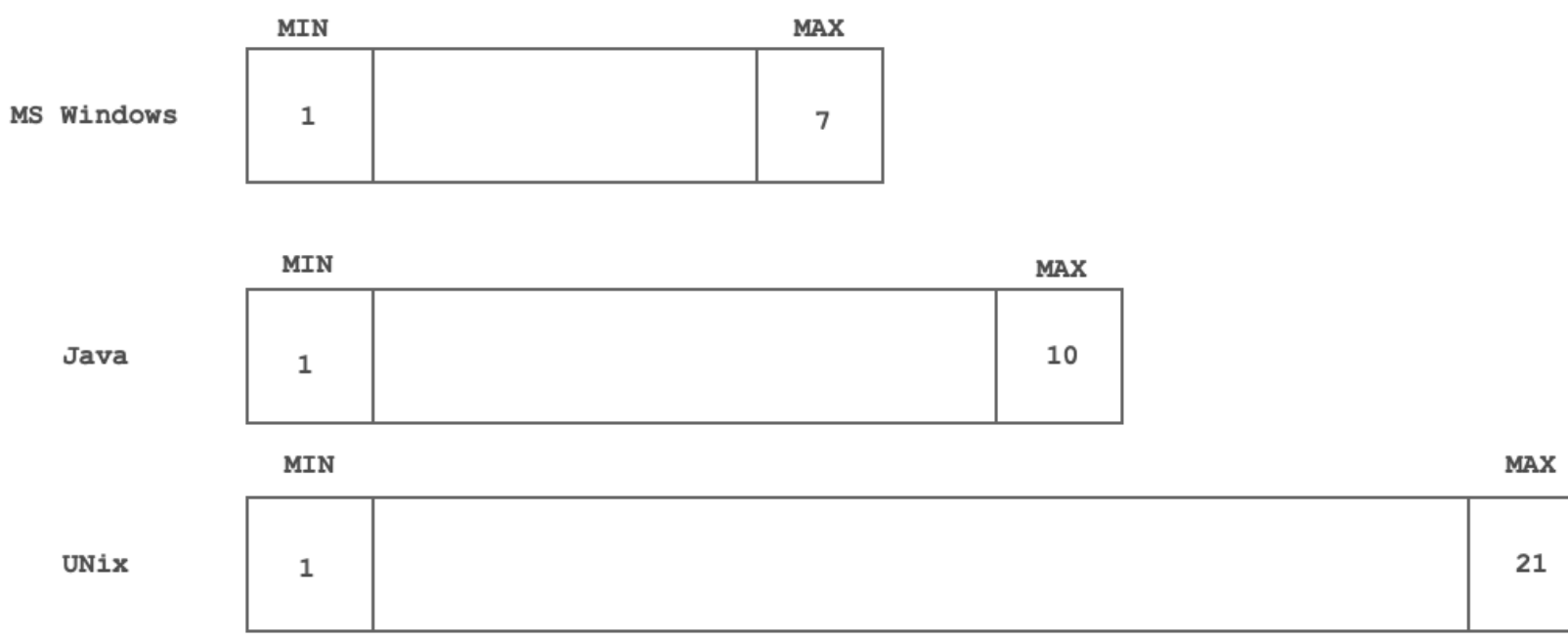
```
public static void main(String[] args) {
    Thread thread = Thread.currentThread();
    thread.setPriority(thread.getPriority() + 2);
    System.out.println("Priority of "+thread.getName()+" is "+thread.getPriority());
    //Priority of main is 7
}
```

```
public static void main(String[] args) {
    Thread thread = Thread.currentThread();
    thread.setPriority(thread.getPriority() + 6); //IllegalArgumentException
    System.out.println("Priority of "+thread.getName()+" is "+thread.getPriority());
}
```

- Default priority of any thread depends on priority of its parent thread.
- Consider following code:

```
class Task implements Runnable{
  private Thread thread;
```

```
  public Task(){
    this.thread = new Thread( this );
    this.thread.setPriority( Thread.MAX_PRIORITY);  //OK
    this.thread.start();
  }
  public void run( ){
    //TODO
  }
}
```



Threads	Java	Unix	Windows
t1	8	16	7
t2	7	14	7
t3	3	6	3

- If we set priority of a thread in Java then OS map it differently on differnt system. Hence Same Java application produces different behavior on different OS.
- Which features of Java makes Java application platform dependent:
  - Abstract Window Toolkit(AWT) Components
    - AWT components implicitly use peer classes and these classes has been written into C++ which are OS specific.
  - Thread priorities

Thread Join

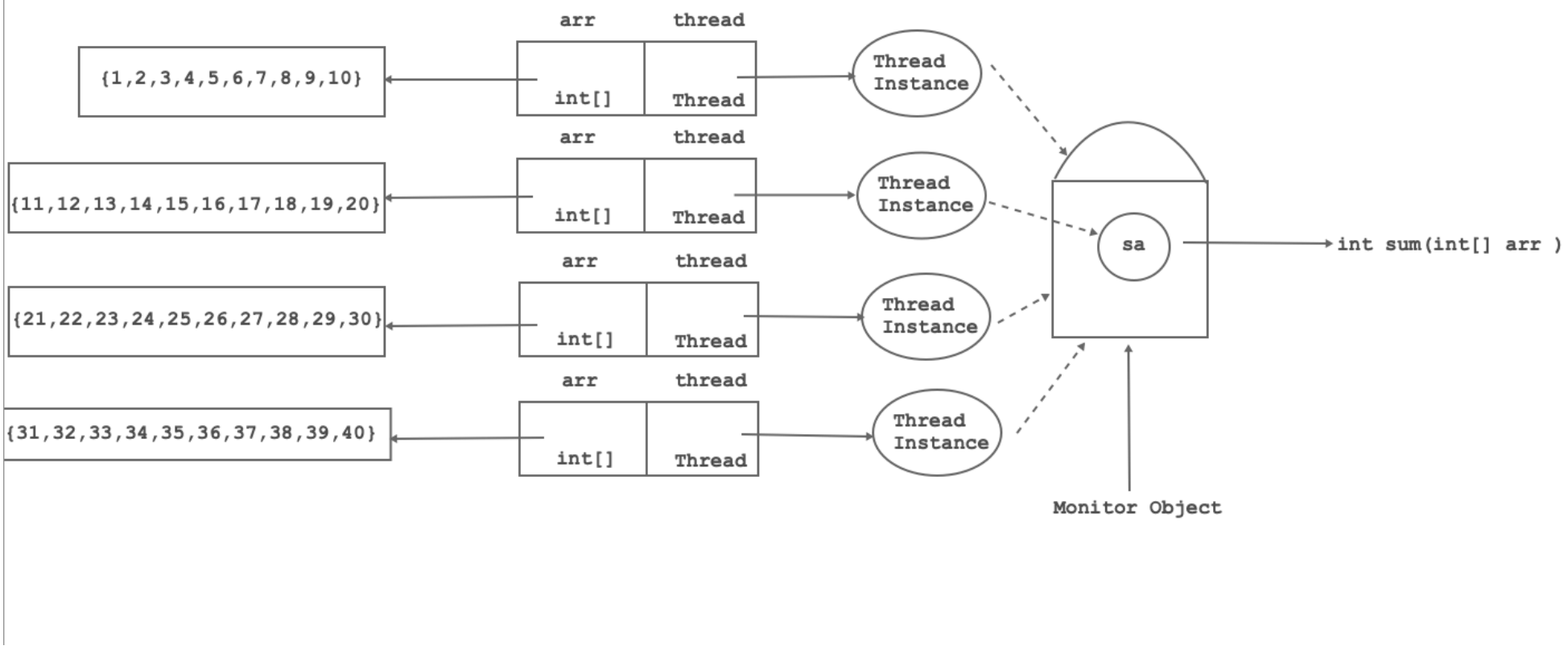
- If we call join method on thread instance then it blocks execution of all other threads
- Consider following code:

```
class Task extends Thread{
  public Task( String name ) {
    super( name );
    this.start();
  }
  @Override
  public void run() {
    System.out.println("Start of run method");
    try {
      for( int count = 1; count <= 10; ++ count ) {
        System.out.println(this.getName()+" : Count   :   "+count);
        Thread.sleep(250);
      }
    } catch (InterruptedException cause) {
      throw new RuntimeException( cause );
    }
    System.out.println("End of run method");
  }
}
public class Program {
  public static void main(String[] args) throws InterruptedException {
    Thread th1 = new Task("Thread#01");
```

```
        th1.join();

        Thread th2 = new Task("Thread#02");
        th2.join();

        Thread th3 = new Task("Thread#03");
        th3.join();

        Thread th4 = new Task("Thread#04");
        th4.join();

        Thread th5 = new Task("Thread#05");
        th5.join();
      }
    }
```

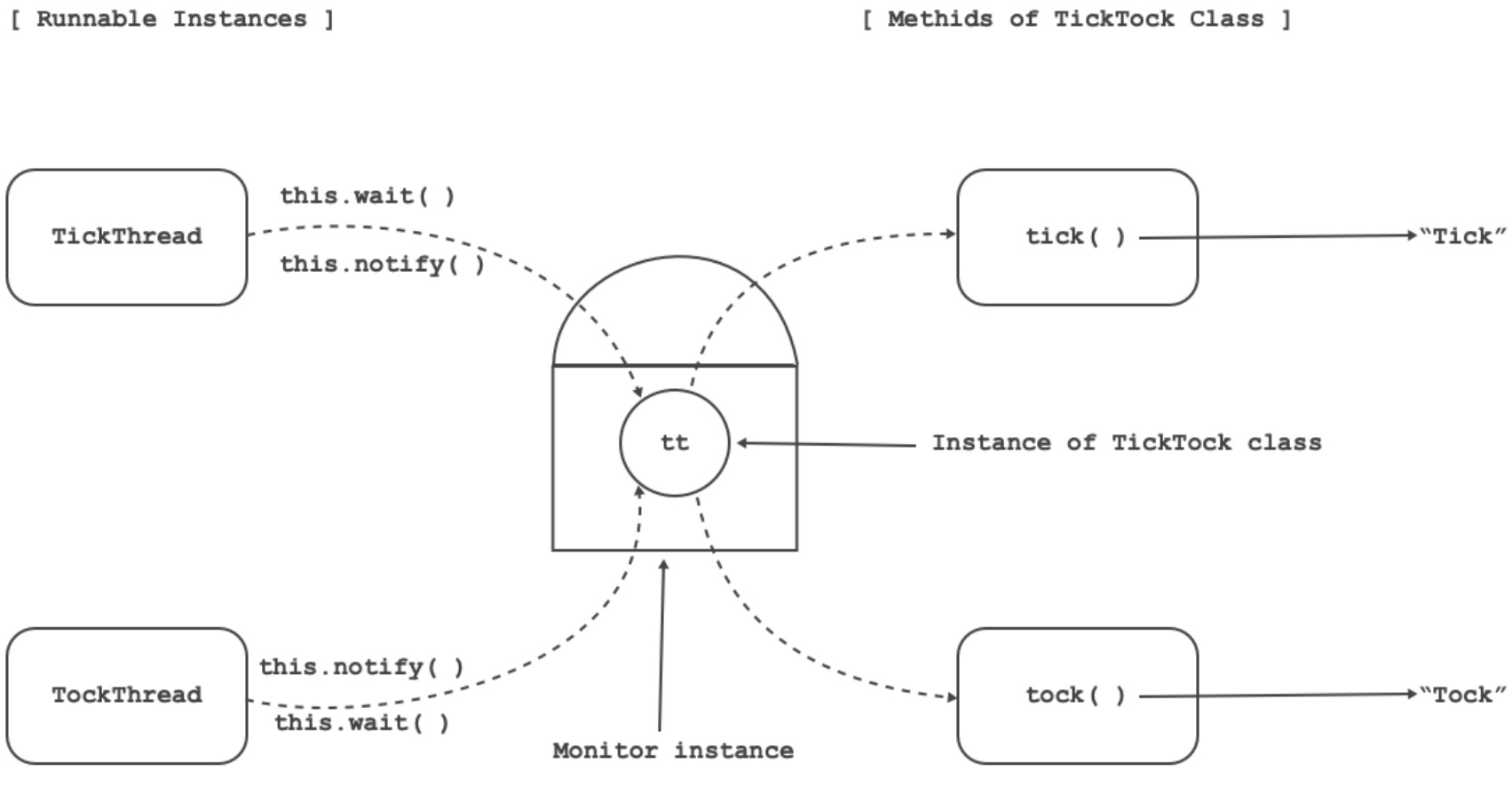
Race Condition

- A race condition is a situation that occurs in a concurrent system when two or more threads or processes access a shared resource or variable in an uncontrolled order, resulting in unpredictable and often incorrect behavior.
- To prevent race conditions, concurrency control mechanisms such as locks, semaphores, and monitors can be used to ensure that only one thread at a time can access and modify a shared resource.
- In Java,we can use synchronized keyword with block/method to avoid race condition.
- If threads are waiting to get monitor object associated with shared resource then it is consider in BLOCKED state.



Inter Thread Communication

- Synchronization is the process of controlling the access of multiple threads to a shared resource.
- Inter-thread communication is a mechanism that allows threads in a multi-threaded application to communicate with each other and coordinate their actions.



[Inter Thread Communication]

- In Java, inter-thread communication is achieved using the wait(), notify() and notifyAll() methods, which are defined in the Object class.
    - **wait()** : This method causes the current thread to wait until another thread invokes the notify() or notifyAll() method for the same object. The thread will release the lock it holds on the object and wait until it's notified by another thread.
    - **notify()** : This method wakes up a single thread that is waiting on the object. If there are multiple threads waiting, only one thread will be awakened. The awakened thread will not be able to proceed until it regains the lock on the object.
    - **notifyAll()** : This method wakes up all the threads that are waiting on the object. All the threads will then compete for the lock on the object.
- The JVM throws the IllegalMonitorStateException when a thread attempts to call the wait(), notify(), or notifyAll() methods on an object without holding the object's monitor.

```
class TickTock{
  public void tick() throws InterruptedException {
    synchronized( this ) {
      System.out.print("Tick  ");
      this.notify();
      this.wait( 1000 );  //To avoid deadlock pass time
    }
  }
  public void tock() throws InterruptedException {
    synchronized( this ) {
      System.out.println("    Tock");
      this.notify();
      this.wait( 1000 );  //To avoid deadlock pass time
    }
  }
}
```

```
class Task implements Runnable{
  Thread thread;
  public Task( String name) {
    this.thread = new Thread(this, name);
    this.thread.start();
  }
  private static TickTock tt = new TickTock();
  @Override
  public void run() {
    try {
      if( Thread.currentThread().getName().equals("TickThread")) {
        for( int count = 1; count <= 5; ++ count ) {
          tt.tick();
          Thread.sleep(250);
        }
      }else {
        for( int count = 1; count <= 5; ++ count ) {
```

```
            tt.tock();
            Thread.sleep(250);
          }
        }
      } catch (InterruptedException cause) {
        throw new RuntimeException( cause );
      }
    }
  }
```

```
public class Program {
  public static void main(String[] args)throws Exception{
    Task t1 = new Task("TickThread");

    Task t2 = new Task("TockThread");
  }
}
```

Why wait/notify/notifyAll methods are declared in java.lang.Object class?

- The wait(), notify(), and notifyAll() methods are designed to work with monitors, which are associated with objects rather than threads.
- When a thread calls wait() on an object, it releases the monitor associated with that object and enters a waiting state until another thread notifies it by calling notify() or notifyAll() on the same object.
- Therefore, it makes sense for these methods to be declared in the java.lang.Object class.

```
public class Program {
  public void print( String str ) throws InterruptedException {
    synchronized( this ) {
      System.out.println(str);
      //str.wait(1000); //IllegalMonitorStateException
      this.wait( 1000 );
    }
  }
  public static void main(String[] args)throws Exception{
    Program p = new Program();
    p.print("Hello");
  }
}
```

The Producer-Consumer problem

- It is a classic synchronization problem in computer science where there are two threads - a producer and a consumer - sharing a common resource.
- Consider a problem can be seen in a fast-food restaurant's kitchen:
  - the kitchen is the common resource
  - the chef is the producer who produces the food
  - waiter is the consumer who serves the food to the customers.
- We will try to solve this problem using synchronization mechanisms like wait(), notify(), and notifyAll().
- Consider code for Kitchen class:

```
import java.util.LinkedList;
public class Kitchen {
  private LinkedList<String> orders = new LinkedList<>();
  private int maxOrders = 10;

  public synchronized void addOrder(String order) throws InterruptedException {
      while (orders.size() == maxOrders) {
          wait();
      }
      orders.add(order);
      System.out.println("Order added: " + order);
      notifyAll();
  }

  public synchronized String getOrder() throws InterruptedException {
      while (orders.size() == 0) {
          wait();
      }
```

```
        String order = orders.removeFirst();
        System.out.println("Order removed: " + order);
        notifyAll();
        return order;
    }
}
```

- Consider code for Chef class:

```
public class Chef implements Runnable {
  private Kitchen kitchen;

  public Chef(Kitchen kitchen) {
    this.kitchen = kitchen;
  }

  @Override
  public void run() {
    try {
      while (true) {
        String order = "Burger"; // produce food
        kitchen.addOrder(order);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

- Consider code for Waiter class:

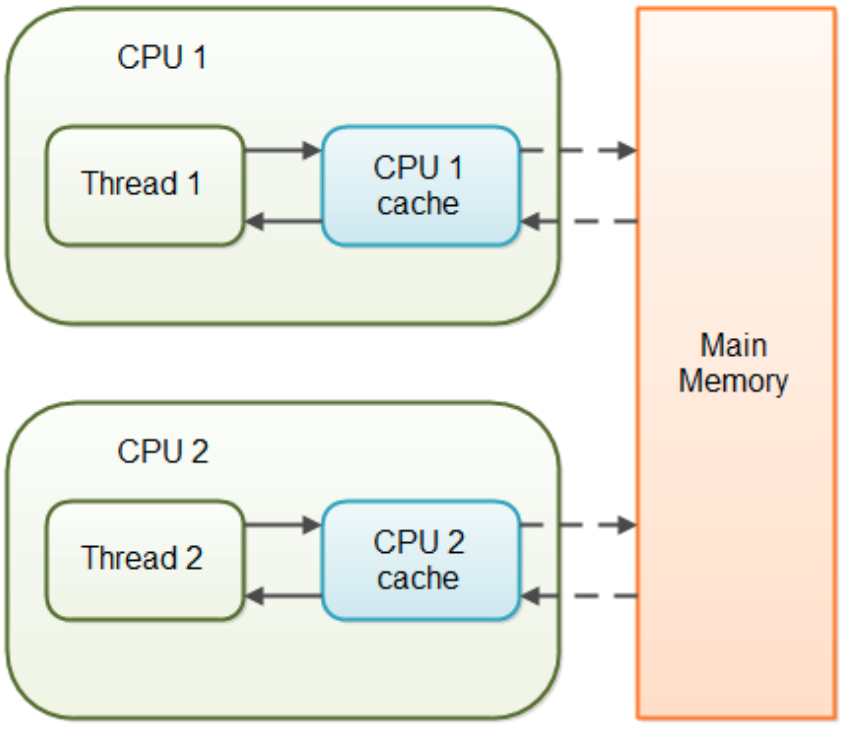
```
  public class Waiter implements Runnable {
    private Kitchen kitchen;
    public Waiter(Kitchen kitchen) {
      this.kitchen = kitchen;
    }

    @Override
    public void run() {
      try {
        while (true) {
          String order = kitchen.getOrder();
          serve(order); // consume food
          Thread.sleep(1000);
        }
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    private void serve(String order) {
      System.out.println("Serving " + order);
    }
  }
```

- Consider code for Waiter class:

```
public class Restaurant {
    public static void main(String[] args) {
    Kitchen kitchen = new Kitchen();
    Thread chefThread = new Thread(new Chef(kitchen));
    Thread waiterThread = new Thread(new Waiter(kitchen));
    chefThread.start();
    waiterThread.start();
  }
}
```

Volatile Fields:

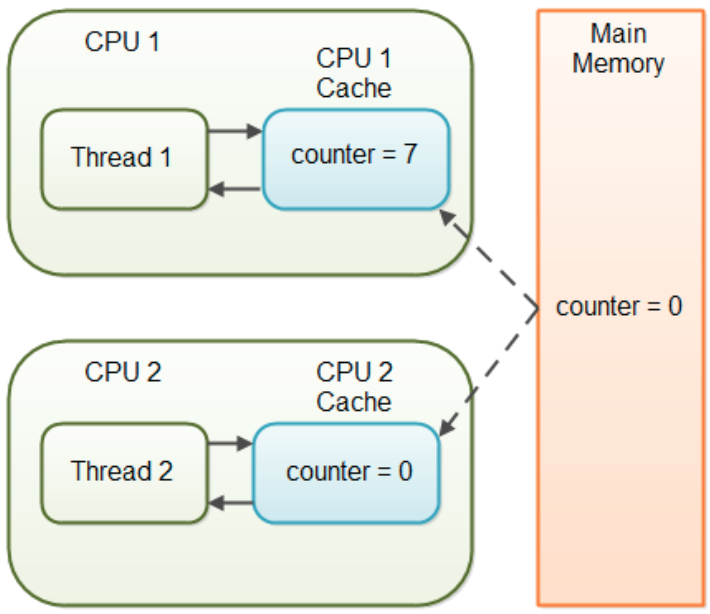
- Computers with multiple processors can temporarily hold memory values in registers or local memory caches. As a consequence, threads running in different processors may see different values for the same memory location.



- Consider following code:

```
class SharedInstance{
  int counter = 0;
}
```

  - Here we assume that there are two threads are working on SharedInstance.
  - If these two threads run on different processors then each thread will have its own local copy of counter.
  - If we modifies value of one thread, then its value might not reflect in the original one in the main memory instantly. It is totally depends on the write policy of cache.
  - Now the other thread is not aware of the modified value which leads to data inconsistency.



  - If the counter variable is not declared volatile there is no guarantee about when the value of the counter variable is written from the CPU cache back to main memory. This means, that the counter variable value in the CPU cache may not be the same as in main memory.
- volatile is a keyword in Java which is applicable only for fields.

```
Access_Modifier volatile Data_Type variableName;
```

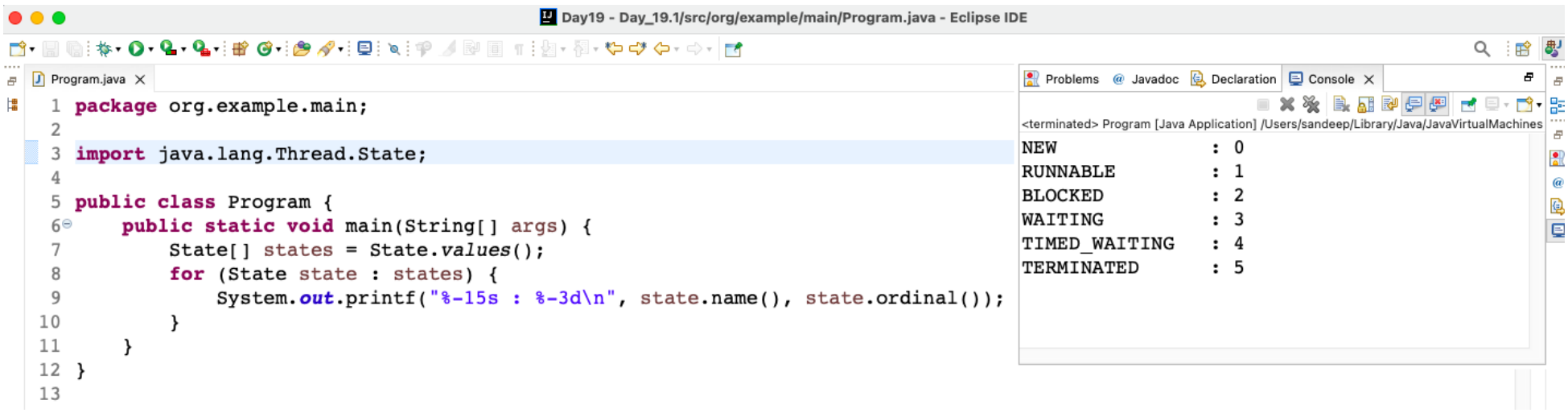
- volatile keyword in Java guarantees that value of the volatile variable will always be read from main memory and not from Thread's local cache.
- The volatile keyword tells the computer that a variable may be accessed by multiple threads at the same time. This means that any changes made to the variable are immediately visible to all other threads. It works like a special type of variable that is shared among all threads and stored in main memory, so every time a thread wants to access it, it reads the updated value from main memory and writes any changes directly back to main memory. Any change to a volatile variable is visible to all other threads immediately after it happens.

Thread life cycle:

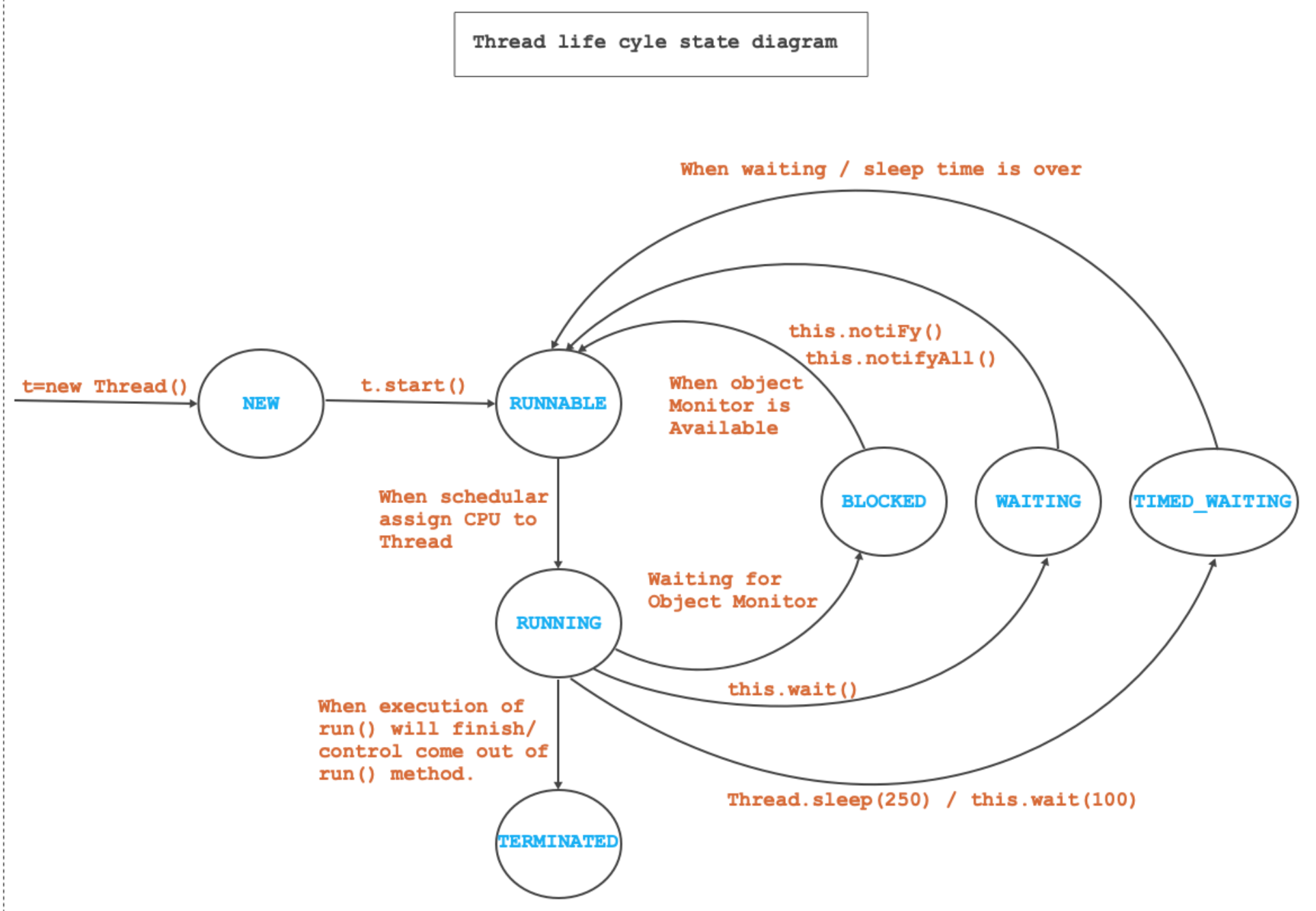
- Thread.State is nested enum declared in java.lang.Thread class.

```
public enum State {
  NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED;
}
```

- Accessing enum in Java



- The life cycle of a thread in Java consists of several states, which are as follows:



- A thread can be in only one state at a given point in time.
- These states are virtual machine states which do not reflect any operating system thread states.
  - **New**: The thread is in the new state when it is created but has not yet started executing. The thread remains in this state until the start() method is called.
  - **Runnable**: The thread is in the runnable state when it is ready to run, but the thread scheduler has not selected it to run yet. When the thread scheduler selects the thread, it moves to the running state.
  - **Running**: The thread is in the running state when its run() method is being executed.
  - **Blocked**: The thread is in the blocked state when it is waiting for a monitor lock to be released, so it can enter a synchronized block or method.
  - **Waiting**: The thread is in the waiting state when it is waiting for some other thread to perform a particular action before it can continue executing.
  - **Timed Waiting**: The thread is in the timed waiting state when it is waiting for a specific amount of time for some other thread to perform a particular action before it can continue executing.

- **Terminated**: The thread is in the terminated state when its run() method has completed execution.

How to analyze thread dumps:

- URL: https://fastthread.io/