Parul® University

Concurrent Programming

Prof. Jayvirsinh Kher, Assistant Professor
Computer Science & Engineering

Parul®
University

DIGITAL LEARNING CONTENT

CHAPTER-5

Concurrent Programming

What is Concurrency ?

- According to dictionary : “Two or more events happening at the same time”
- In Computer Science : “The execution of two or more tasks in overlapping time periods that are not necessarily simultaneous”

Image source : Google

Parul®
University

DIGITAL LEARNING CONTENT

Multi Threaded Programming

Parul®
University

DIGITAL LEARNING CONTENT

INTRODUCTION

- Contains two or more parts that can run concurrently.
- Each part of such program is called a thread
- Thread defines a separate path of execution
- It cannot exist on its own. It must be part of a process.
- Executing several tasks simultaneously is called “MultiTasking”.

There are two types of multitasking

(1) Process Based Multitasking

(2) Thread Based Multitasking

PROCESS BASED MULTITASKING

- Each task separately independent process

- Example : While typing a java program in editor we can able to listen audio songs by MP3 Player in the system. At the same time we can download a file from the internet. All these tasks are executing simultaneously and independent of each other. Hence it is process based multitasking.

- Best suitable at OS level

THREAD BASED MULTITASKING

- Each task separately independent part of the same program and each independent part is called a Thread.
- Best suitable for programmatic level
- Java provides in built support for multithreading
- The main important application areas of multithreading are:
    1. To implement multimedia graphics.
    2. To develop video games etc.
    3. To develop web and application servers

DEFINING THREAD

- **A process is a program in execution.**
- **Threads are light-weight processes within a process**

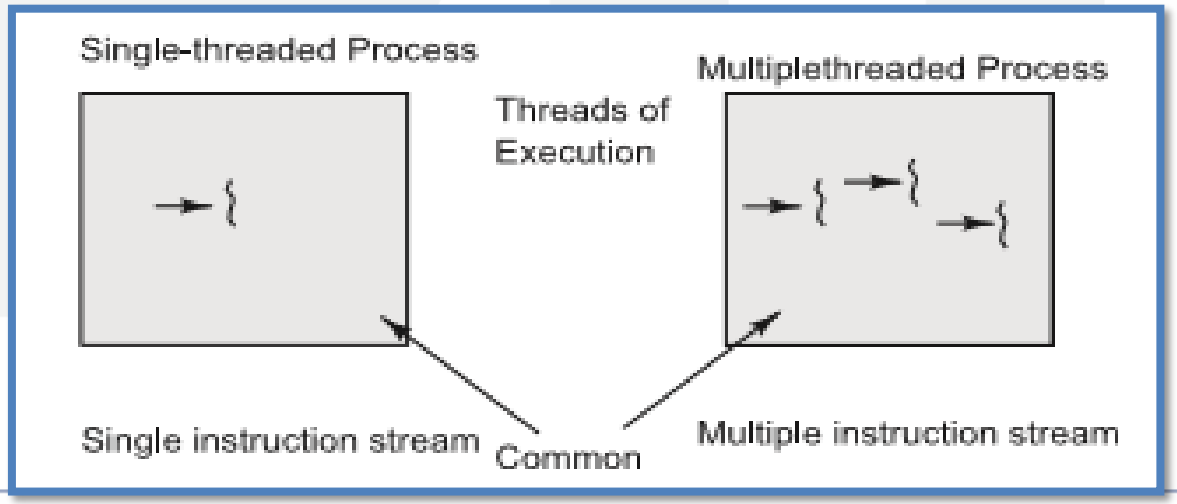


Image source : Google

THREADS IN JAVA

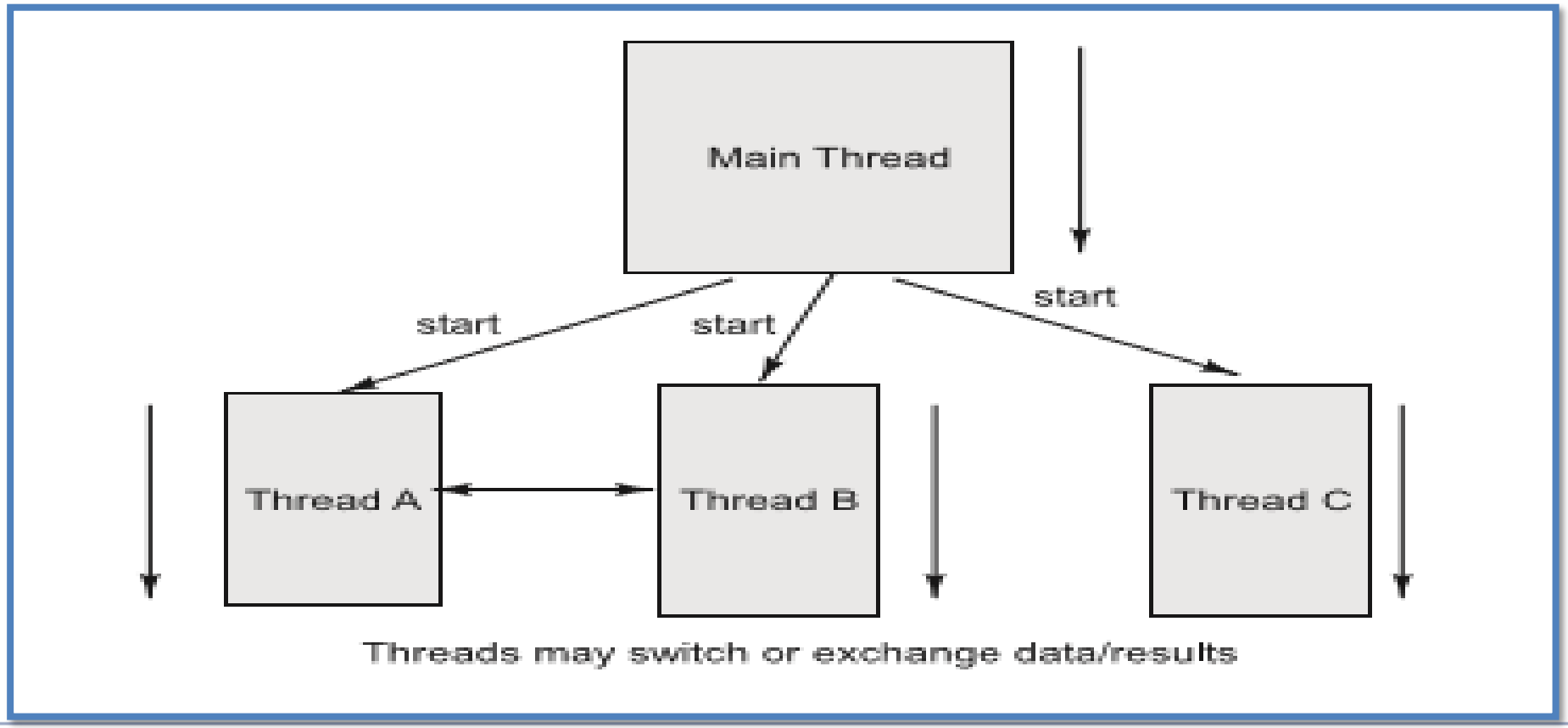


Image source : Google

THREADS IN JAVA

- **We can define a thread in following ways:**

  [a] By extending Thread class

  [b] By implementing Runnable interface

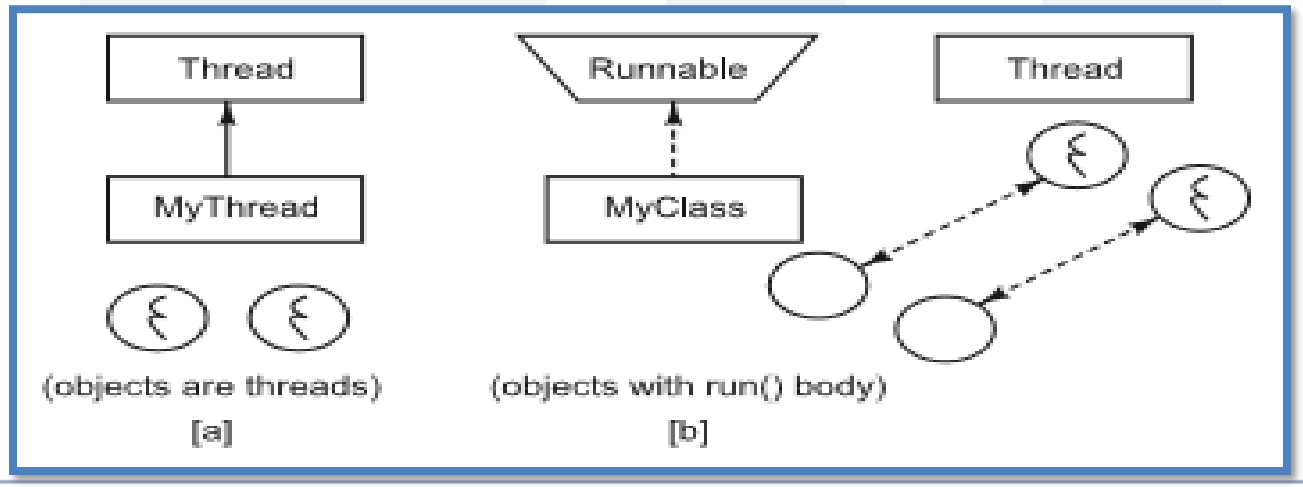


Image source : Google

DEFINING THREAD BY EXTENDING THREAD CLASS

- **The steps for creating a thread by using the first mechanism are:**

1. Create a class by extending the Thread class and override the run() method:

```
class MyThread extends Thread
{
        public void run() {
                // thread body of execution
        }
}
```

DEFINING THREAD BY EXTENDING THREAD CLASS

2. Create a thread object:

```
MyThread t = new MyThread();
```

3. Start Execution of created thread:

```
t.start();
```

DEFINING THREAD BY EXTENDING THREAD CLASS

EXAMPLE:

```
class MyThread extends Thread
{
   public void run()
   {
        for(int i=0;i<10;i++)
        {
            System.out.println("child Thread");
        }
   }
}
```

defining a Thread.

Job of a Thread.

DEFINING THREAD BY EXTENDING THREAD CLASS

EXAMPLE:

```
class ThreadDemo {
        public static void main(String[] args)  {
                MyThread t=new MyThread();  //Instantiation of a Thread
                t.start();  //starting of a Thread
                for(int i=0;i<5;i++)   {
                        System.out.println("main thread");
                }
        }
}
```

DEFINING THREAD BY IMPLEMENTING RUNNABLE INTERFACE

- **The steps for creating a thread by using the second mechanism are:**

1. Create a class that implements the interface Runnable and override run() method:

```
class MyRunnable implements Runnable {    …
        public void run() {
                // thread body of execution
        }
}
```

DEFINING THREAD BY IMPLEMENTING RUNNABLE INTERFACE

2. Creating Object:

```
MyRunnable myObject = new MyRunnable ();
```

3. Creating Thread Object:

```
Thread t = new Thread(myObject);
```

4. Start Execution:

```
t.start();
```

DEFINING THREAD BY IMPLEMENTING RUNNABLE INTERFACE

EXAMPLE:

```
class MyRunnable implements Runnable
{
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            System.out.println("child Thread");
        }
    }
}
```

defining a Thread

job of a Thread

DEFINING THREAD BY IMPLEMENTING RUNNABLE INTERFACE

EXAMPLE:

```
class ThreadDemo {
        public static void main(String[] args){
                MyRunnable myObject =new MyRunnable();
                Thread t=new Thread(myObject);
                //here myObject is a Target Runnable   t.start();
                for(int i=0;i<10;i++)  {
                        System.out.println("main thread");}
                }
        }
```

THREAD STATES (THREAD LIFE CYCLE)

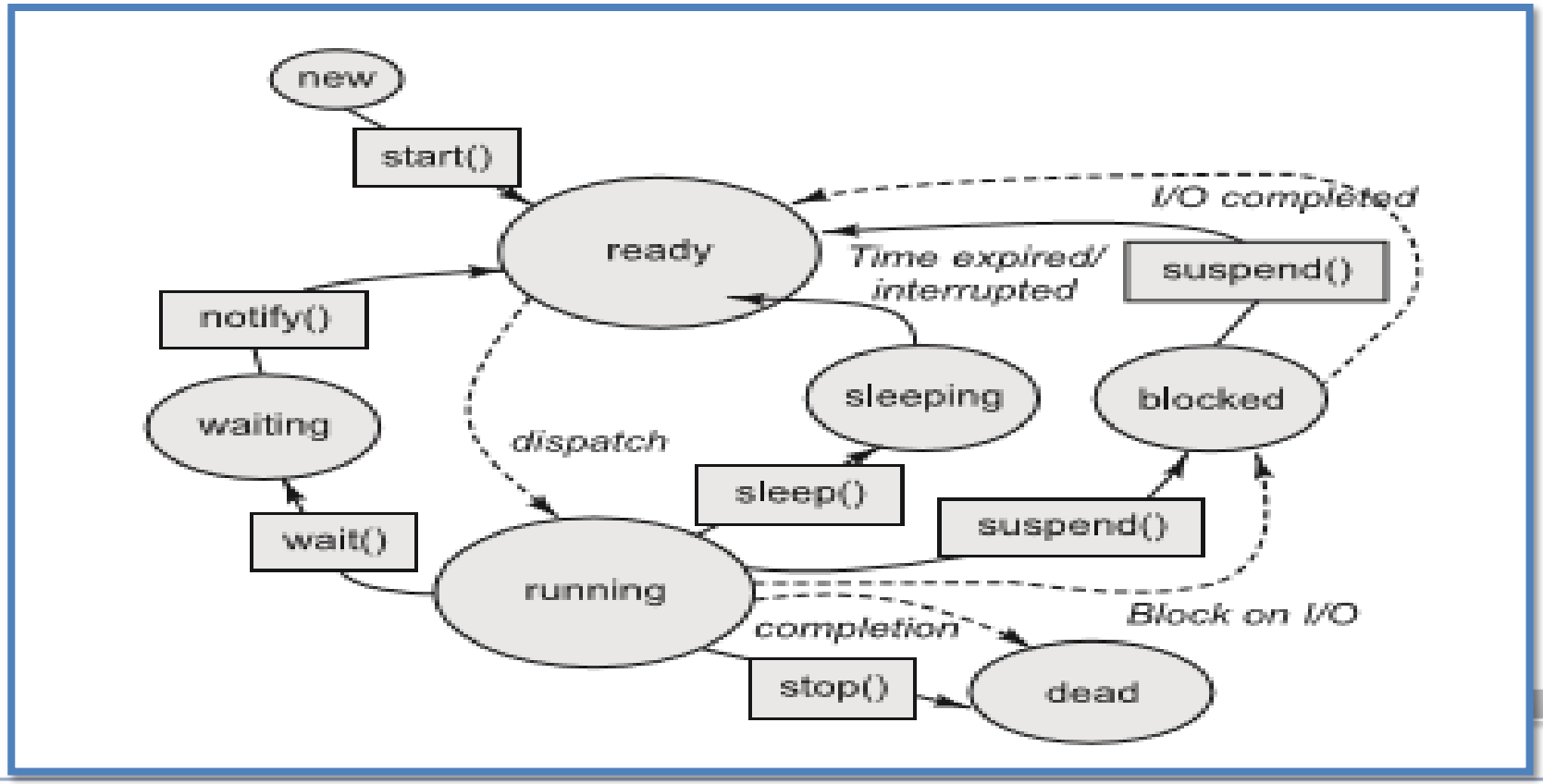
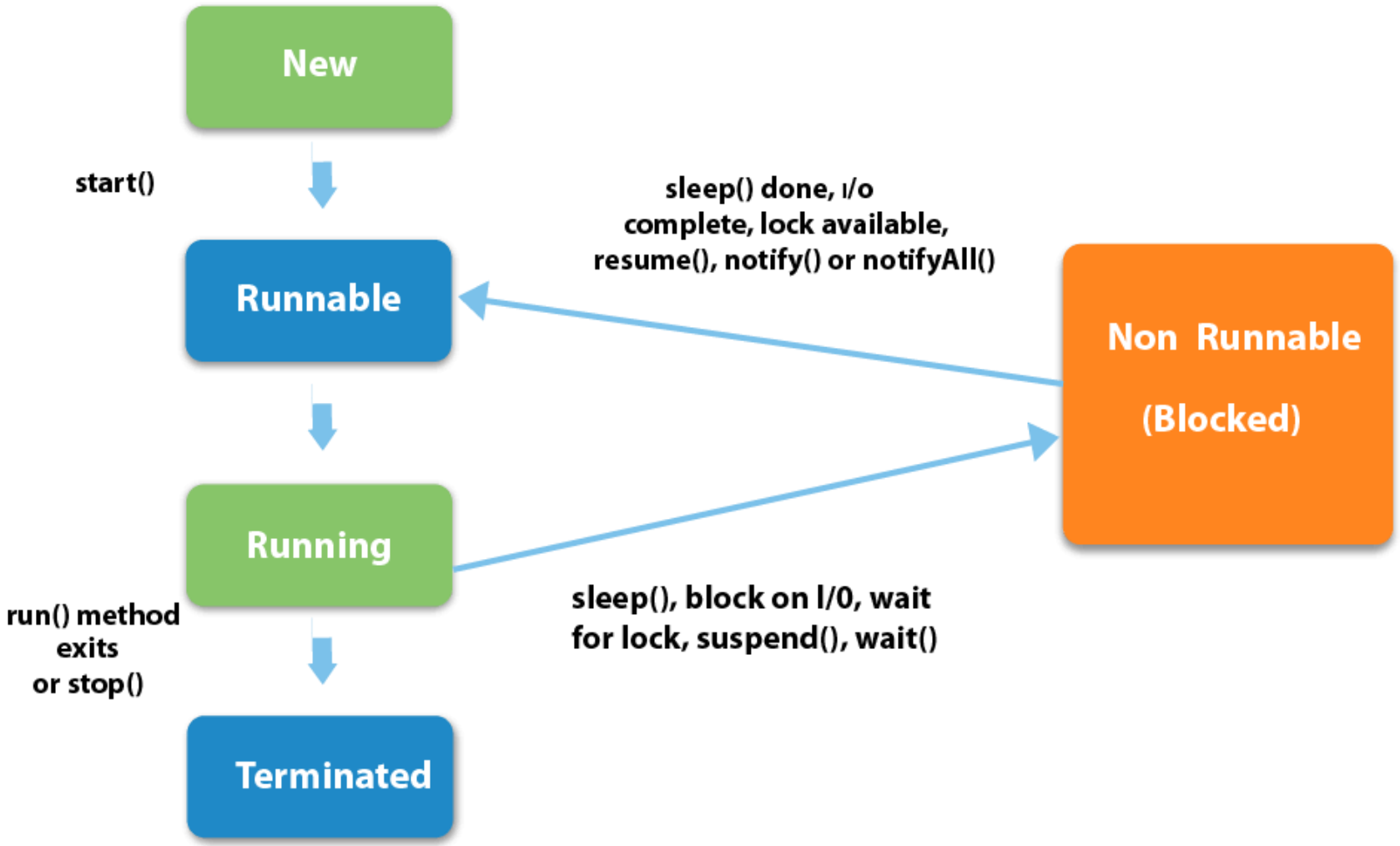


Image source : Google

THREAD STATES (THREAD LIFE CYCLE)

- New
- Runnable (Ready → Running)
- Running
- Non-Runnable (Blocked)
- Terminated


New
start()
Runnable
Running
run() method
exits
or stop()
Terminated
sleep() done, I/o
complete, lock available,
resume(), notify() or notifyAll()
Non Runnable
(Blocked)
sleep(), block on I/0, wait
for lock, suspend(), wait()

1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

THREAD PRIORITIES

- Every Thread in java has some priority it may be default priority generated by JVM (or) explicitly provided by the programmer.
- The valid range of Thread priorities is 1 to 10[but not 0 to 10] where 1 is the least priority and 10 is highest priority.

•Thread class defines the following constants to represent some standard priorities.

1. Thread. MIN_PRIORITY----------1
2. Thread. MAX_PRIORITY----------10
3. Thread. NORM_PRIORITY--------5

THREAD PREVENTION METHODS

- **We can prevent(stop) a Thread execution by using the following methods.**

1. **yield();**
2. **join();**
3. **sleep();**

THREAD PREVENTION METHODS : yield()

- **To pause current executing Thread for giving the chance of remaining waiting Threads of same priority**

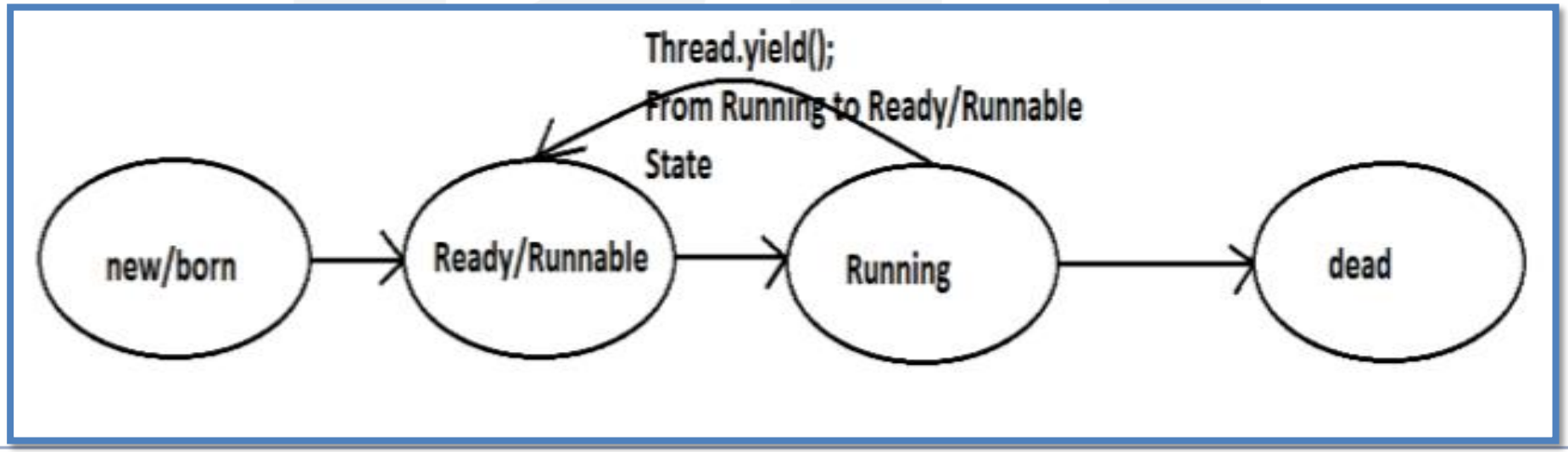


Image source : Google

THREAD PREVENTION METHODS : yield()

- **EXAMPLE**

```
class MyThread extends Thread {
        public void run()  {
                for(int i=0;i<5;i++)   {
                        Thread.yield();
                        System.out.println("child thread");
                        }
                }
}
```

THREAD PREVENTION METHODS : yield()

```
class ThreadYieldDemo {
        public static void main(String[] args)  {
                MyThread t=new MyThread();
                t.start();
                for(int i=0;i<5;i++)   {
                        System.out.println("main thread");
                }
            }
}
```

Image source : Google

THREAD PREVENTION METHODS : JOIN()

- **If a Thread wants to wait until completing some other Thread then we should go for join() method.**

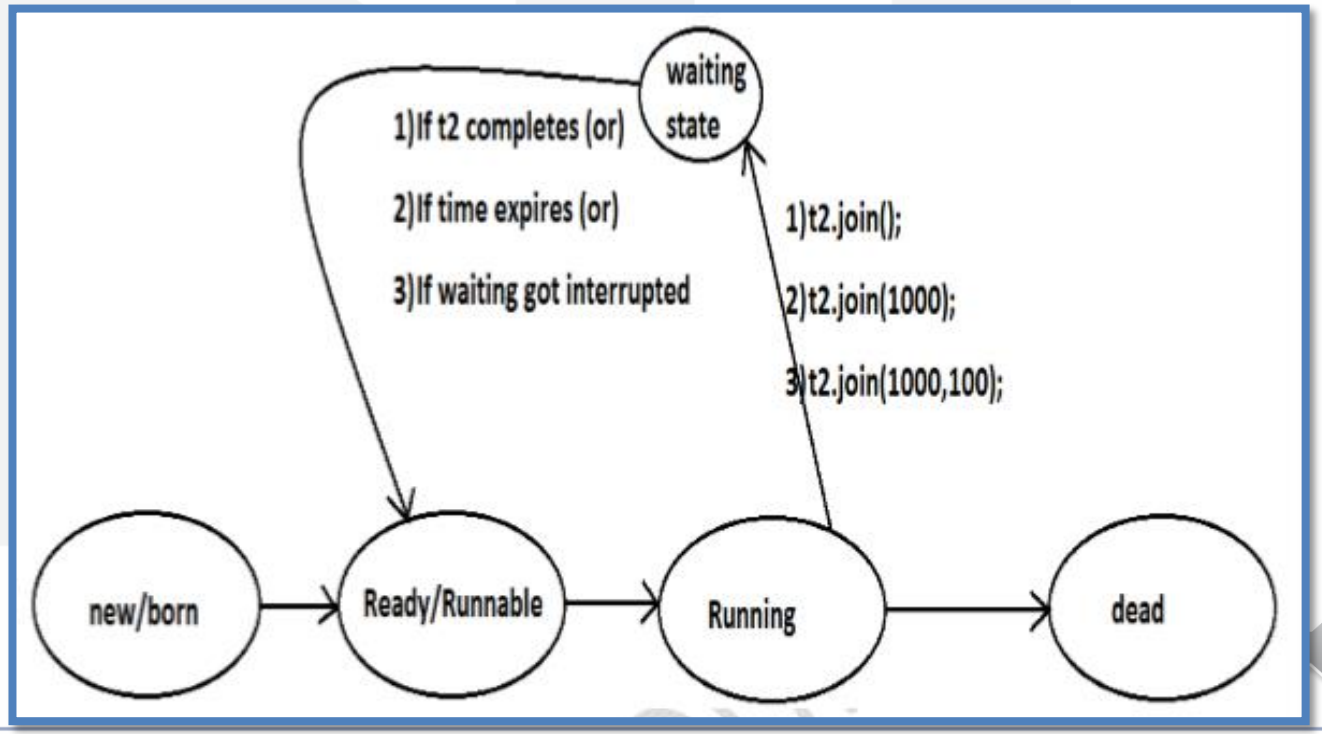


Image source : Google

THREAD PREVENTION METHODS : SLEEP()

- **If a Thread don't want to perform any operation for a particular amount of time then we should go for sleep() method.**

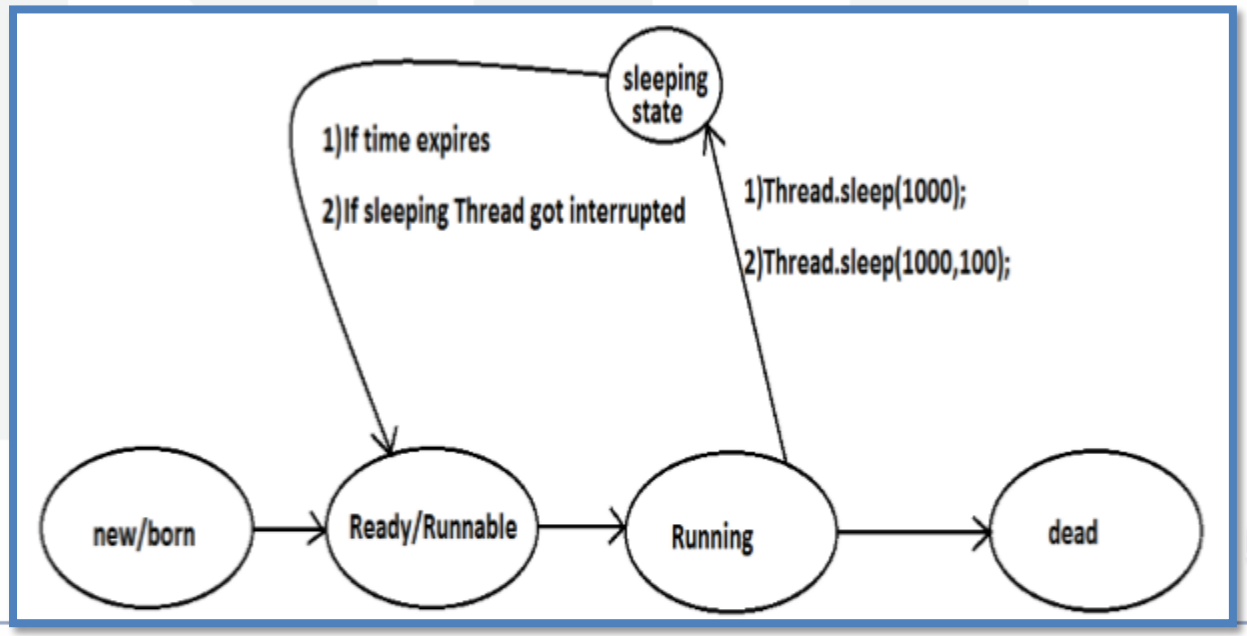


Image source : Google

INTERRUPTING THREAD

- If a Thread can interrupt a sleeping or waiting Thread by using interrupt() (break off) method of Thread class.

```
public void interrupt();
```

- Whenever we are calling interrupt() method we may not see the effect immediately, if the target Thread is in sleeping or waiting state it will be interrupted immediately.

INTERRUPTING THREAD

- If the target Thread is not in sleeping or waiting state then interrupt call will wait until target Thread will enter into sleeping or waiting state. Once target Thread entered into sleeping or waiting state it will effect immediately.

- In its lifetime if the target Thread never entered into sleeping or waiting state then there is no impact of interrupt call simply interrupt call will be wasted.

THREAD SYNCHRONIZATION

- Synchronized is the keyword applicable for methods and blocks but not for classes and variables.
- If a method or block declared as the synchronized then at a time only one Thread is allow to execute that method or block on the given object.
- The main advantage of synchronized keyword is we can resolve date inconsistency problems.
- But the main disadvantage of synchronized keyword is it increases waiting time of the Thread and effects performance of the system.

THREAD SYNCHRONIZATION

- Hence if there is no specific requirement then never recommended to use synchronized keyword.

- Internally synchronization concept is implemented by using lock concept.

- Every object in java has a unique lock. Whenever we are using synchronized keyword then only lock concept will come into the picture.

THREAD SYNCHRONIZATION

- If a Thread wants to execute any synchronized method on the given object 1st it has to get the lock of that object. Once a Thread got the lock of that object then it's allow to execute any synchronized method on that object. If the synchronized method execution completes then automatically Thread releases lock.
- While a Thread executing any synchronized method the remaining Threads are not allowed execute any synchronized method on that object simultaneously. But remaining Threads are allowed to execute any non-synchronized method simultaneously.

Thread safe collection

- If multiple threads concurrently modify a data structure such as a hash table, then it is easily possible to damage the data structure.

- You can protect a shared data structure by supplying a lock, but it is usually easier to choose a thread-safe implementation instead.

Thread safe collection

[1] Efficient Maps, Sets, and Queues

- **The java.util.concurrent package supplies efficient implementations for maps, sorted sets, and queues: Concurrent Hash Map, Concurrent Skip List Map, Concurrent Skip List Set, and Concurrent Linked Queue.**
- **minimize contention by allowing concurrent access to different parts of the data structure.**
- **The collections return weakly consistent iterators.**

Thread safe collection

- To ensure that only one thread adds an item into the cache :

  cache.putIfAbsent(key, value);

- The opposite operation is remove:

  cache.remove(key, value)

- Atomically removes the key and value if they are present in the map. Finally,

  cache.replace(key, oldValue, newValue)

Thread safe collection

[2] Copy on write Arrays

- **The CopyOnWriteArrayList and CopyOnWriteArraySet are thread-safe collections in which all mutators make a copy of the underlying array.**
- **When you construct an iterator, it contains a reference to the current array.**
- **If the array is later mutated, the iterator still has the old array, but the collection's array is replaced. As a consequence, the older iterator has a consistent (but potentially outdated) view that it can access without any synchronization expense.**

EXECUTOR

- Constructing a new thread is somewhat expensive because it involves interaction with the operating system.
- If your program creates a large number of short-lived threads, then it should instead use a thread pool.
- A thread pool contains a number of idle threads that are ready to run.
- Another reason to use a thread pool is to throttle the number of concurrent threads

Executors Factory Methods

Method	Description
newCachedThreadPool	New threads are created as needed; idle threads are kept for 60 seconds.
newFixedThreadPool	The pool contains a fixed set of threads; idle threads are kept indefinitely.
newSingleThreadExecutor	A "pool" with a single thread that executes the submitted tasks sequentially (similar to the Swing event dispatch thread).
newScheduledThreadPool	A fixed-thread pool for scheduled execution; a replacement for java.util.Timer.
newSingleThreadScheduledExecutor	A single-thread "pool" for scheduled execution.

Synchronizer

- The java.util.concurrent package contains several classes that help manage a set of collaborating threads
- These mechanisms have “canned functionality” for common rendezvous patterns between threads.
- If you have a set of collaborating threads that follows one of these behavior patterns, you should simply reuse the appropriate library class instead of trying to come up with a handcrafted collection of locks and conditions.

Synchronizer Factory Methods

Class	What It Does	When To Use
CyclicBarrier	Allows a set of threads to wait until a predefined count of them has reached a common barrier, and then optionally executes a barrier action.	When a number of threads need to complete before their results can be used.
CountDownLatch	Allows a set of threads to wait until a count has been decre-mented to 0.	When one or more threads need to wait until a specified number of events have occurred.
Exchanger	Allows two threads to exchange objects when both are ready for the exchange.	When two threads work on two instances of the same data structure, one by filling an instance and the other by emptying the other.

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in