Multithreading

- About Concurrency
- Thread and Runnable objects
- Object sharing by multiple threads
- Race Condition
- Synchronization
- Deadlock
- Inter-thread communication
- Thread Life Cycle
- java.util.concurrent package

Concurrency

Concurrency is the ability to run several programs or several parts of a program concurrently (simultaneously).

Multi-tasking operating systems can run multiple applications concurrently where each application runs in a separate window.

Even a **single application** is often expected to do more than one thing at a time *(multithreading).*

For ex. a streaming audio application must simultaneously read the digital audio off the network, decompress it, manage playback, and update its display.

Task Scheduling

Modern operating systems use preemptive multitasking to allocate CPU time to applications.

There are *two terms* we need to know:

1. Process:

- A process is an area of memory that contains both code and data.
- A process has a thread of execution that is scheduled to receive CPU time slices.
- Processes are often seen as synonymous with programs or applications
- Most implementations of the *Java virtual machine* run as a single process.

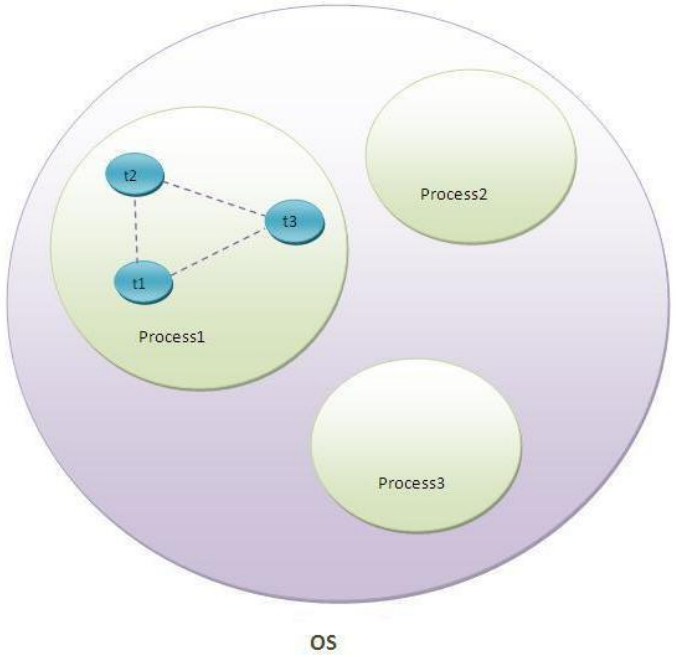
2. Thread:

- A thread is ***a path of execution*** within a process. It is a scheduled execution of a process.
- Concurrent threads are possible.
- All threads for a process share the same data memory but may be following different paths through a code section.
- Threads are sometimes called *lightweight processes*.

Processes & Threads

Threads exist within a process — every process has at least one thread.

Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic.



Multithreaded execution is an essential feature of the Java platform.

Every application has at least one thread , called the *main thread*. This thread has the ability to create additional threads.

Implementing multithreaded Programming

An application that creates an thread instance must provide the code that will run in that thread.

There are two ways to do this:

- *Provide a **Runnable** object to Thread class constructor*
- *Extending Thread class*

Implementing Runnable Interface

1.Provide a Runnable object to Thread class constructor

- A Runnable object is an instance of a class that implements *Runnable* interface
- The **Runnable** interface declares a single method, ***run().*** The implementation class should define this method, *run().*
- Then the runnable object is passed to the Thread class constructor and then invoke the *start()* method of thread instance, as shown below:

```
Thread thread = new Thread(runnable object);
 thread.start();   // implicitly invokes run() method
```

Extending Thread class

2. Extending Thread class

- The **Thread** class itself implements **Runnable**, though its run method does nothing.
- An application can subclass Thread, providing its own implementation of run() method

Thread class constructors:

1. Thread t1 = new Thread();
2. Thread t2 = new Thread(*runnableObject*);
3. Thread t3 = new Thread(object,"thread-name");
4. Thread t4 = new Thread ("thread-name");

Implementation through both the methods

```
public class HelloRunnable implements Runnable{
public void run() {
      System.out.println("Hello from a thread!");
 }
}

Class HelloRunnableDemo {
public static void main(String args[]) {
      Runnable  runnableObject  = new HelloRunnable();
      Thread t = new Thread(runnableObject);
       t.start();
}
 }
```

The start() method of thread object implicitly invokes run() method.

```
public class HelloThread extends Thread{

public HelloThread() {
super();
this.start();
}

@Override
public void run() {
System.out.println("Thread Name: "+ this.getName());
}}
```

```
public class HelloThreadDemo {
public static void main(String[] args) {
System.out.println("Main Thread Name: "+
Thread.currentThread().getName());
new HelloThread();
}
}
```

```
Main Thread Name: main
Thread Name: Thread-0
```

Current Thread Details

```
public class CurrentThreadDetails{
public static void main(String [] args){
        Thread thread = Thread.currentThread();
        System.out.println(thread);
        System.out.println(thread.getName());
}
}
```

// The first println() output:

Thread[main,5,main]

- The **first main** indicates name of the thread
- **5** indicates priority number
- The **second main** indicates name of the **thread group**

// The second println() output:

main

Thread class methods

1. Thread thread = Thread.currentThread() : *returns current thread object*
2. String s = thread.getName() : *returns thread name*
3. thread.setName("thread-name") : To set a name to the user-thread
4. thread.start() : *invokes run() method*
5. Thread.sleep(milliseconds) : The current thread will into sleep state for specified number of milliseconds, *1000 ms is 1 sec*
6. int priorityNumber= thread.getPriority() : *returns priority of the thread,*

   Note: priority range: 0 (Min) to 10(Max). We can provide any number from 0 to 10 or use constants, MIN_PRIORITY whose value is 0 , MAX_PRIORITY is 10 and NORM_PRIORITY value is 5
7. thread.setPriority(*priorityNumber*) : *sets the priority of a thread*
8. Boolean thread.isAlive() : *returns true if thread is alive else returns false.*
9. thread.join() : The caller thread waits till the *thread* dies i.e. caller thread joins after *thread* completes its execution

join() method

The join method allows one thread to wait for the completion of another thread.
If any executing thread t1 calls join() on t2 i.e; t2.join() then t1 will immediately enter into waiting state until t2 completes its execution.

```
public class JoinDemo implements Runnable{
 public void run() {
      System.out.println(Thread.currentThread().getName()+" is alive: "+Thread.currentThread().isAlive());
      try{
      Thread.sleep(1000);
      }
      catch(InterruptedException ie){}
 }
   public static void main(String args[]) throws Exception {
      Thread thread = new Thread(new JoinDemo(),"MyThread");
      thread.start();
      thread.join();
      System.out.println(thread.getName()+" is alive: "+thread.isAlive());
   }
}
```

Without thread.join() statement:
MyThread is alive: true
MyThread is alive: true

With thread.join() statement:
MyThread is alive: true
MyThread is alive: false

Compute-intensive task allocated to a separate thread

A Thread class for sorting a List in the background

```
public class Worker
implements Runnable{

List<Integer> list;

public Worker(List<Integer> list) {
super();
this.list = list;
}
@Override
public void run() {
Collections.sort(list);
}
}
```

```
public class WorkerTester {
public static void main(String[] args) throws InterruptedException {
List<Integer> myList = new ArrayList<>();

for(int i=0;i<10000;i++){
myList.add(new Random().nextInt());
}
System.out.println(myList);
//Collections.sort(myList);
long start=System.currentTimeMillis();
Worker worker =new Worker(myList);
Thread t = new Thread(worker);
t.start();
t.join();
System.out.println("Time taken to sort: "+(System.currentTimeMillis()-
start)+ " milliseconds");
//System.out.println(myList);}}
```

main thread waits until worker thread completes its run() method

Multiple threads working on shared object

```
public class SharedObject implements Runnable{
private  static StringBuilder message=new StringBuilder("Welcome");
@Override
public void run() {
      int size=message.length();
      for(int i=0;i<size;i++){
      System.out.println(Thread.currentThread().getName()
      +":"+ message.append('x'));
      }
}
}
```

```
public class SharedObjectTester {
public static void main(String[] args) throws
InterruptedException {
SharedObject sharedObject=new SharedObject();
Thread t1=new Thread(sharedObject);
Thread t2 = new Thread(sharedObject);
Thread t3 = new Thread(sharedObject);
t1.start();
t2.start();
t3.start();
}
}
```

The above program leads to ***race condition*** as all the 3 threads are trying to access the shared object at a time.

Race Condition

- The situation where more than one thread competes for the same resource, where the sequence in which the resource is accessed is significant, is called race condition.
- Without synchronization, it is possible for one thread to modify a shared resource while another thread is in the process of using or updating that resource. This is called as ***thread interference***.
- The section of the code that leads to race conditions is called a **critical section**.
- Race conditions can lead to unpredictable results and subtle program bugs.
- Race conditions can be avoided by ***proper thread synchronization in critical sections.***

Synchronization

The synchronization is the capability to control the access of multiple threads to shared resources.

The mechanism that Java uses to support synchronization is the *monitor* (also called a semaphore).

Each object in Java is associated with a monitor, which a thread can lock or unlock.

A monitor is used as a mutually exclusive lock(**mutex)** which enables multiple threads to independently work on shared data without interfering with each other.

Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor.

Synchronized Method & Synchronized Block

There two synchronization syntaxes in Java Language.

- Synchronized method
- Synchronized block

- The practical differences are in **controlling scope and the monitor.**
- With a synchronized method, the lock is obtained for the duration of the entire method.
- With synchronized blocks you can specify exactly when the lock is needed.

```
class Program {
public synchronized void f() {
.....
}
}
```

```
class Program {
public void f() {
synchronized(this){
...
}
}
}
```

Multiple threads working on shared object

```
public class SharedObject implements Runnable{
private  static StringBuilder message=new StringBuilder("Welcome");
@Override
public void run() {
      int size=message.length();
      synchronized(this){
            for(int i=0;i<size;i++){
            System.out.println(message.append('x'));
            }
      }
 }
}
```

After modifying the **SharedObject** class, re-run **SharedObjectTester** class and check the output.

volatile

In a multithreaded program , when two or more threads are sharing the same instance variable, for efficiency considerations, each thread keeps its own private copy of such a shared variable.

The real(or master) copy of the variable is updated at various times. So, when a thread is updating the value, it is actually updating in the local cache but not the main copy.

The other thread which is using the same variable but local copy does not know anything about the value changed by the another thread in its local copy.

While this approach works fine, it may be inefficient in cases where the master copy should always reflects its current state.

If a variable is declared with the *volatile* keyword then it is guaranteed that any thread that reads the field will see the most recently written value.

volatile datatype variablename;

Inter Thread Communication

Inter thread communication is about making synchronized threads communicate with each other.

It is implemented by the following methods of **Object** class:

- wait()
- notify()
- notifyAll()

1. wait() : Causes the **current thread to release the lock** and wait until *either* another thread invokes the notify() method or the notifyAll() method for this object, or specified amount of time has elapsed. The current thread must own this object's monitor.

Syntax:

public final wait() throws InterruptedException
public final wait(long timeout) throws InterruptedException

Note: The *wait()* method can be invoked only within synchronized method or block. If used in normal methods, j*ava.lang.IllegalMonitorStateException* exception is thrown.

Inter Thread Communication

2. notify() : Wakes up a single thread that is waiting on this object's monitor. If many threads are waiting on this object, one of them is chosen to be awaken or resurrected.

Syntax :

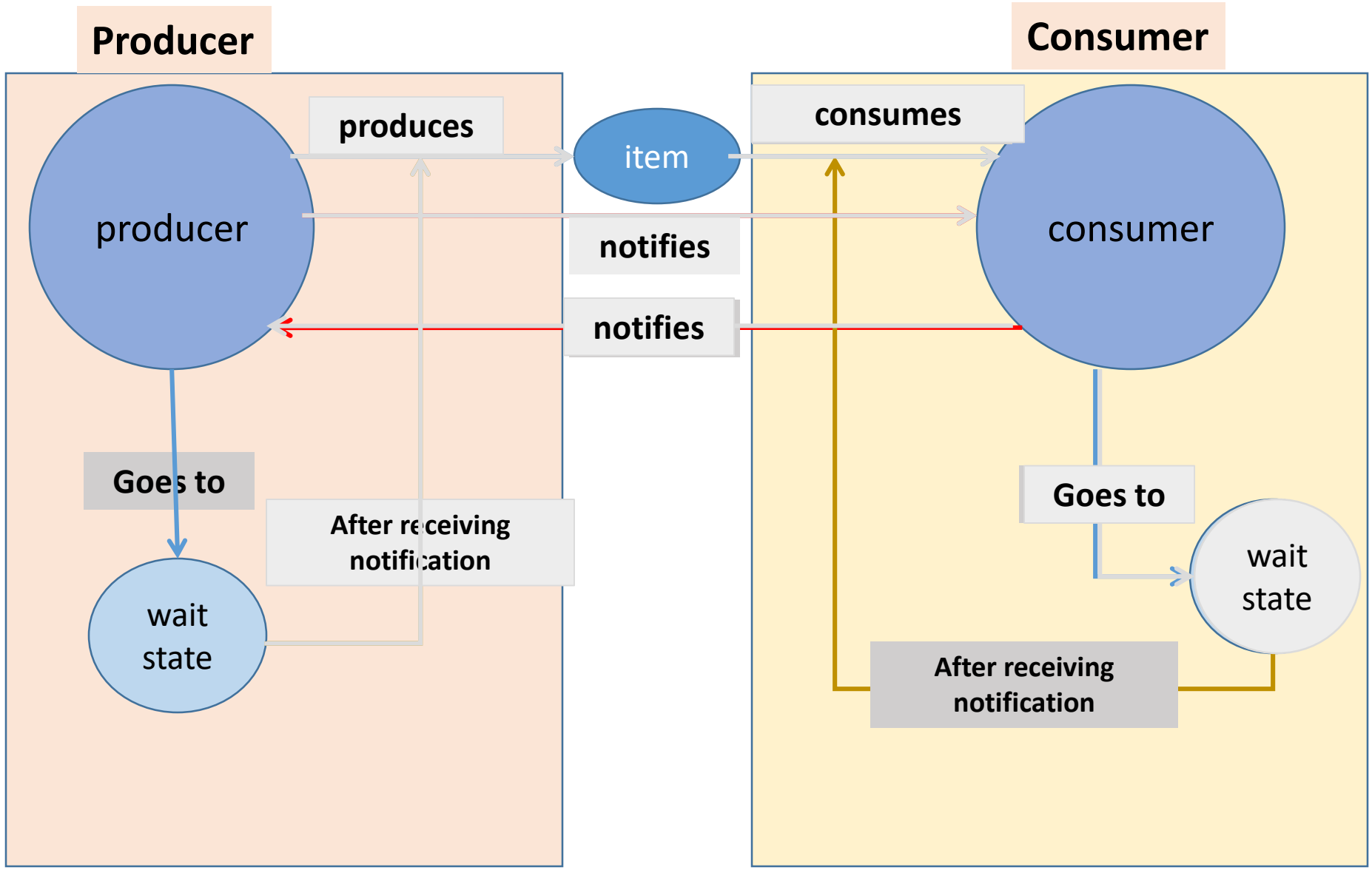
public void notify()

3. notifyAll() : Wakes up all threads that are waiting on this object's monitor

Syntax :

public final notifyAll()

Inter Thread Communication Example



Inter Thread Communication Example

Item
Producer
Consumer
ProducerConsumerTester

Deadlock

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. To analyse a deadlock, we need to look at the java thread dump of the application

Java VisualVM is a tool that provides a visual interface for viewing detailed information about Java applications while they are running on a Java Virtual Machine

To start VisualVM on Windows, run the ***visualvm.exe*** program that is in the \bin folder of your JDK folder

Starvation

Livelock

Daemon and User Threads

Daemon threads in Java are those threads which run in background and mostly created by JVM for performing background tasks like garbage collection and other house keeping tasks.

A call to *setDaemon()* method, with the argument true , makes the thread that is created a daemon thread.

Daemon thread dies if the thread that created it dies.

Non-daemon threads are called as **user threads**.

A user thread has a life of its own that is not dependent on the thread that creates it. It can continue execution after the thread that created it has ended.

Note: main thread is a non-daemon thread.

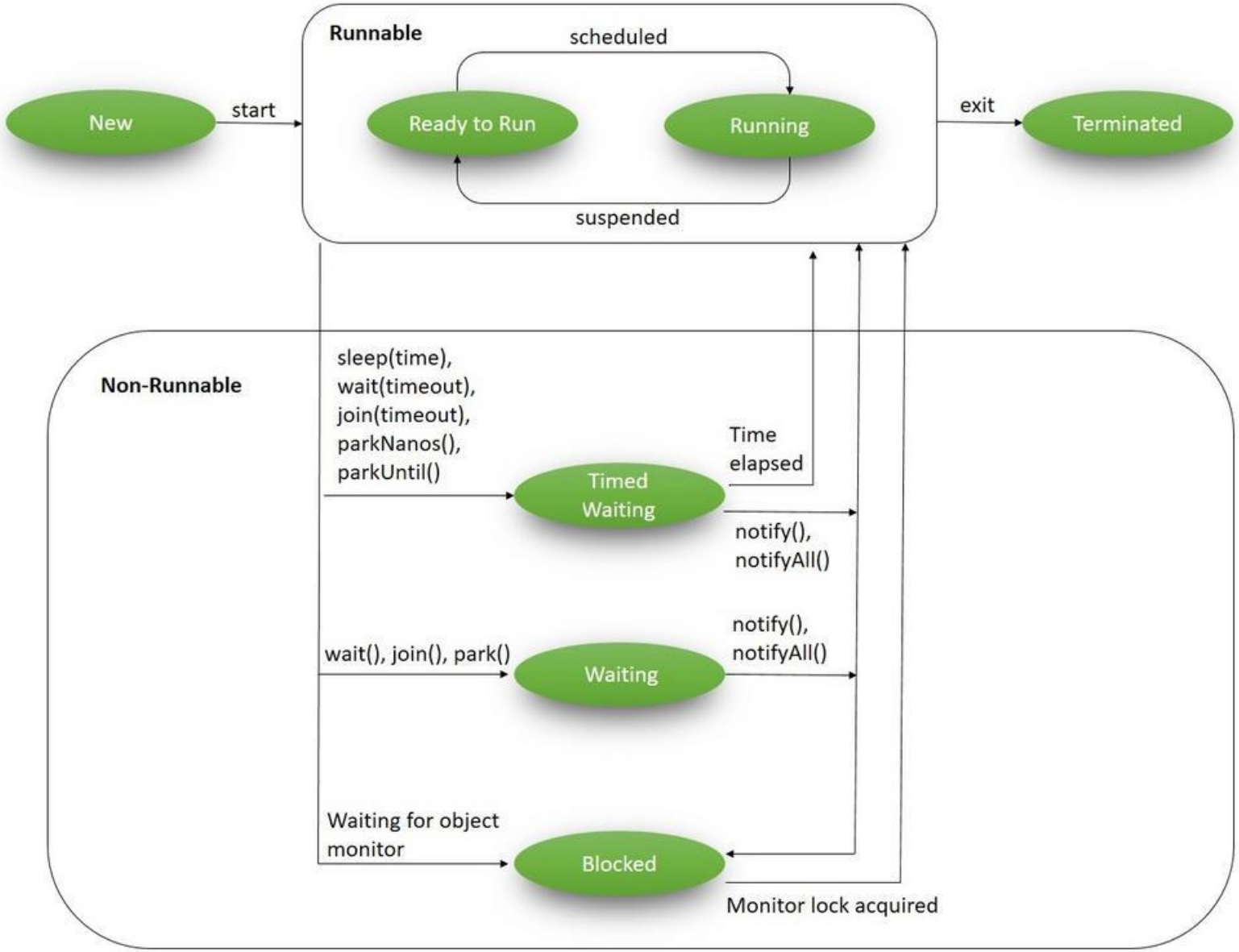
Daemon Thread Example

```
public class DaemonThreadDemo implements Runnable{
public void run(){
while(true){
System.out.println(Thread.currentThread().getName());}
}
public static void main(String[] args) {
System.out.println(Thread.currentThread().getName());
Thread t1 = new Thread(new DaemonThreadDemo());
Thread t2 = new Thread(new DaemonThreadDemo());

//t1.setDaemon(true); t2.setDaemon(true);

t1.start(); t2.start();
try{
Thread.sleep(100);
}catch(InterruptedException ioe){}
System.out.println("End of main thread");
}
}
```

We cannot make a thread daemon if it has already started.
For example we cannot make *main t*hread a daemon thread.
The following code ***throws exception at runtime***.
(java.lang.IllegalThreadStateException)

```
public static void main(String[] args) throws InterruptedException{
Thread t = Thread.currentThread();
t.setDaemon(true);
}
```

Multithread Lifecycle



Concurrency

java.util.concurrent package

The multithreading APIs covered so far are adequate for very basic tasks, but higher-level building blocks are needed for more advanced tasks.

This is especially true for massively concurrent applications that fully exploit today's multiprocessor and multi-core systems.

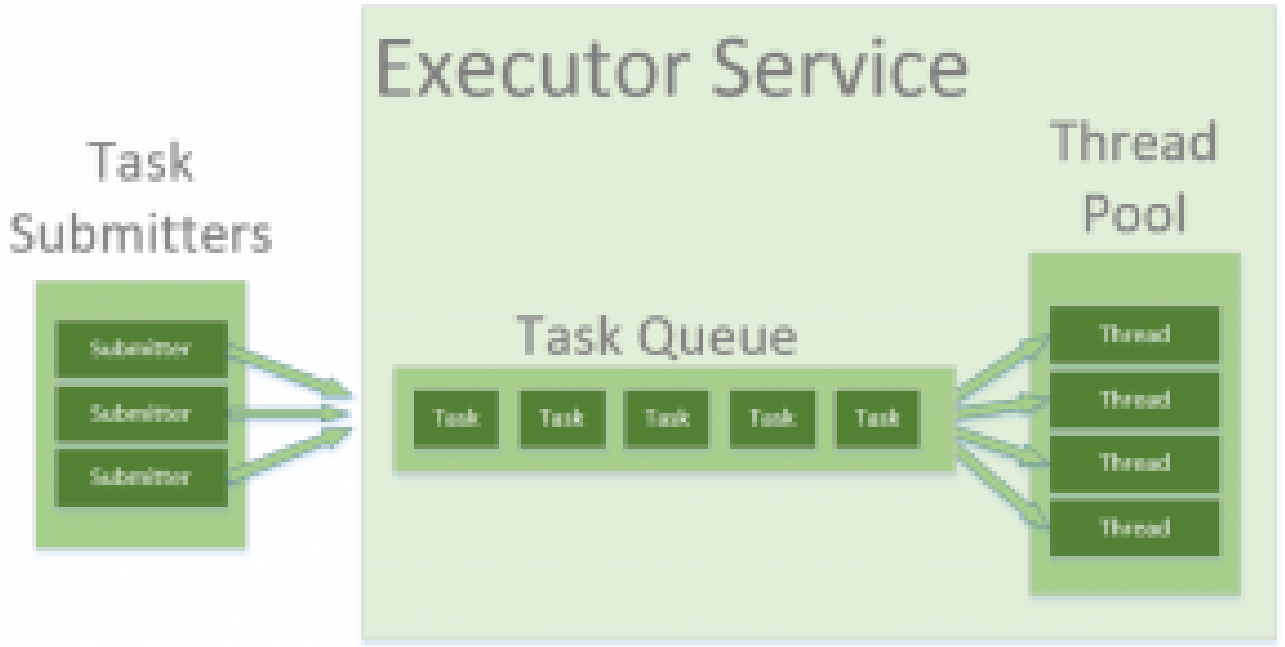
Most of the **high-level concurrency features** are implemented in the new **java.util.concurrent** packages.

The Thread Pool

In Java, threads are mapped to system-level threads which are operating system's resources. If you create threads uncontrollably, you may run out of these resources quickly.

The context switching between threads is done by the operating system as well – in order to emulate parallelism. A simplistic view is that – the more threads you spawn, the less time each thread spends doing actual work.

The Thread Pool pattern helps to save resources in a multithreaded application, and also to contain the parallelism in certain predefined limits.



Executor Interfaces

Executor Interfaces

The java.util.concurrent package defines three executor interfaces:

- **Executor**, a simple interface that supports launching new tasks.
- **ExecutorService**, a sub interface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.
- **ScheduledExecutorService**, a sub interface of ExecutorService, supports future and/or periodic execution of tasks.

The Executor Interface

The Executor interface provides a single method, **execute,** designed to be a drop-in replacement for a common thread-creation idiom.

If r is a Runnable object, and e is an Executor object you can replace
(new Thread(r)).start(); *with* **e.execute(r);**

```
Executor executor = Executors.newSingleThreadExecutor();
executor.execute(() -> System.out.println("Hello World"));
```

Without Executor Interfaces

Without Executor Interfaces

To run a task in a separate thread-of-execution, three things are required:

- ***Thread*** class object
- ***Runnable*** interface implementation or 'The Task'
- Invocation of ***start()*** method on the Thread object

Running Threads Using Executors API

Only one way!

- Create task definition class
- Provide task object to an Executor Service

Preferred way of running tasks...

- Uses thread-pools
- Allocates heavy-weight threads upfront
- Decouples task-submission from thread-creation-and-management
- Each thread in the pool executes multiple tasks one-by-one
- A tasks-queue holds the tasks
- Threads are stopped when the Executor Service is stopped

Important Classes /Interfaces of Executors framework

- Executor (I)
- ExecutorService (I)
- Executors (C)
- Future (I)

```
void execute (Runnable task);
```

```
public class Invoker implements Executor {
    @Override
    public void execute(Runnable r) {
        r.run();
    }
}
```

```
public void execute() {
    Executor executor = new Invoker();
    executor.execute( () -> {
        // task to be performed
    });
}
```

ExecutorService Interface

ExecutorService which extends Executor interface is a complete solution for asynchronous processing. It manages an in-memory queue and schedules submitted tasks based on thread availability.
To use *ExecutorService,* we need to create one *Runnable* class.

- Executor (I)
- ExecutorService (I)
- Executors (C)
- Future (I)

```
<T> Future<T> submit (Callable<T> task);
Future<?> submit (Runnable task);
void shutdown ();
List<Runnable> shutdownNow ();
boolean isShutdown ();
...
```

shutdown() waits till the all submitted task finish executing. The other method is ***shutdownNow()*** which immediately terminates all the pending/executing tasks.
There is also another method ***awaitTermination(long timeout, TimeUnit unit)*** which forcefully blocks until all tasks have completed execution after a shutdown event triggered or execution-timeout occurred, or the execution thread itself is interrupted,

ExecutorService-based Approach

Java provides the *ScheduledExecutorService* interface, which is a more robust and precise solution. This interface can schedule code to run once after a specified delay or at fixed time intervals.

To run a piece of code once after a delay, we can use the ***schedule*** method:

```
ScheduledExecutorService executorService = Executors.newSingleThreadScheduledExecutor();

executorService.schedule( Classname::someTask, delayInSeconds, TimeUnit.SECONDS);
```

To run a task at fixed time intervals, we can use the ***scheduleAtFixedRate*** method:

```
ScheduledExecutorService executorService = Executors.newSingleThreadScheduledExecutor();

executorService.scheduleAtFixedRate( Classname::someTask, 0, delayInSeconds, TimeUnit.SECONDS);
```

Executors Class

- Executor (I)
- ExecutorService (I)
- Executors (C)
- Future (I)

```
public static ExecutorService newFixedThreadPool(int nThreads);
public static ExecutorService newCachedThreadPool();
public static ExecutorService newSingleThreadExecutor();
public static ExecutorService newSingleThreadScheduledExecutor();
...
```

Available thread pools

Most of the executor implementations in java.util.concurrent use *thread pools*, which consist of *worker threads*.

Using worker threads minimizes the overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead.

One common type of thread pool is the *fixed thread pool*. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread.

Tasks are submitted to the pool via an internal queue.

Available Thread pools

newFixedThreadPool(int nThreads) – n threads will process tasks at the time, when the pool is saturated, new tasks will get added to a queue without a limit on size. Good for CPU intensive tasks.

newSingleThreadExecutor() – creates an newFixedThreadPool(1), only one thread will process everything. Good when you really need predictability and sequential tasks completion.

newCachedThreadPool() – doesn't put tasks into a queue. So this pool doesn't have task queue. For a submitted task, if there no free thread, new thread is spawned. When all current threads are busy, it creates another thread to run the task. Sometimes it can reuse threads.

So for almost all purposes, **Executors::newFixedThreadPool(int nThreads)** should be your choice when you need a thread pool.

Threads pools with the Executor Framework

```
// MyRunnable is the task to be performed
public class MyRunnable implements Runnable {
	private final long countUntil;
	public MyRunnable(long countUntil) {
		this.countUntil = countUntil;
	}
	@Override
	public void run() {
		long sum = 0;
		for (long i = 1; i < countUntil; i++) {
			sum += i;
		}
		System.out.println(sum);
	}
}
```

```
// run the runnables with the executor framework.
public class Main {
private static final int NTHREADS = 10;
public static void main(String[] args) throws InterruptedException{
	ExecutorService executor =
	Executors.newFixedThreadPool(NTHREADS);
	for (int i = 0; i < 500; i++) {
		Runnable task = new MyRunnable(10000000L + i);
		executor.execute(task);
	}

	// This will make the executor accept no new runnable objects
	// and finish all existing runnable objects already placed in the queue
	executor.shutdown();

	// Wait until all threads are finish
	executor.awaitTermination(5, TimeUnit.MINUTES);
	System.out.println("Finished all threads");
}
}
```

Future

The *Future* class represents a future result of an asynchronous computation – a result that will eventually appear in the *Future* after the processing is complete.

Long running methods are good candidates for asynchronous processing and the *Future* interface. This enables us to execute some other process while we are waiting for the task encapsulated in *Future* to complete.

Some examples of operations that would leverage the async nature of Future are:

- computational intensive processes (mathematical and scientific calculations)
- manipulating large data structures (big data)
- remote method calls (downloading files, HTML scrapping, web services).

```
public class SquareCalculator {

    private ExecutorService executor
      = Executors.newSingleThreadExecutor();

    public Future<Integer> calculate(Integer input) {
        return executor.submit(() -> {
            Thread.sleep(1000);
            return input * input;
        });
    }
}
```

Consuming *Futures*

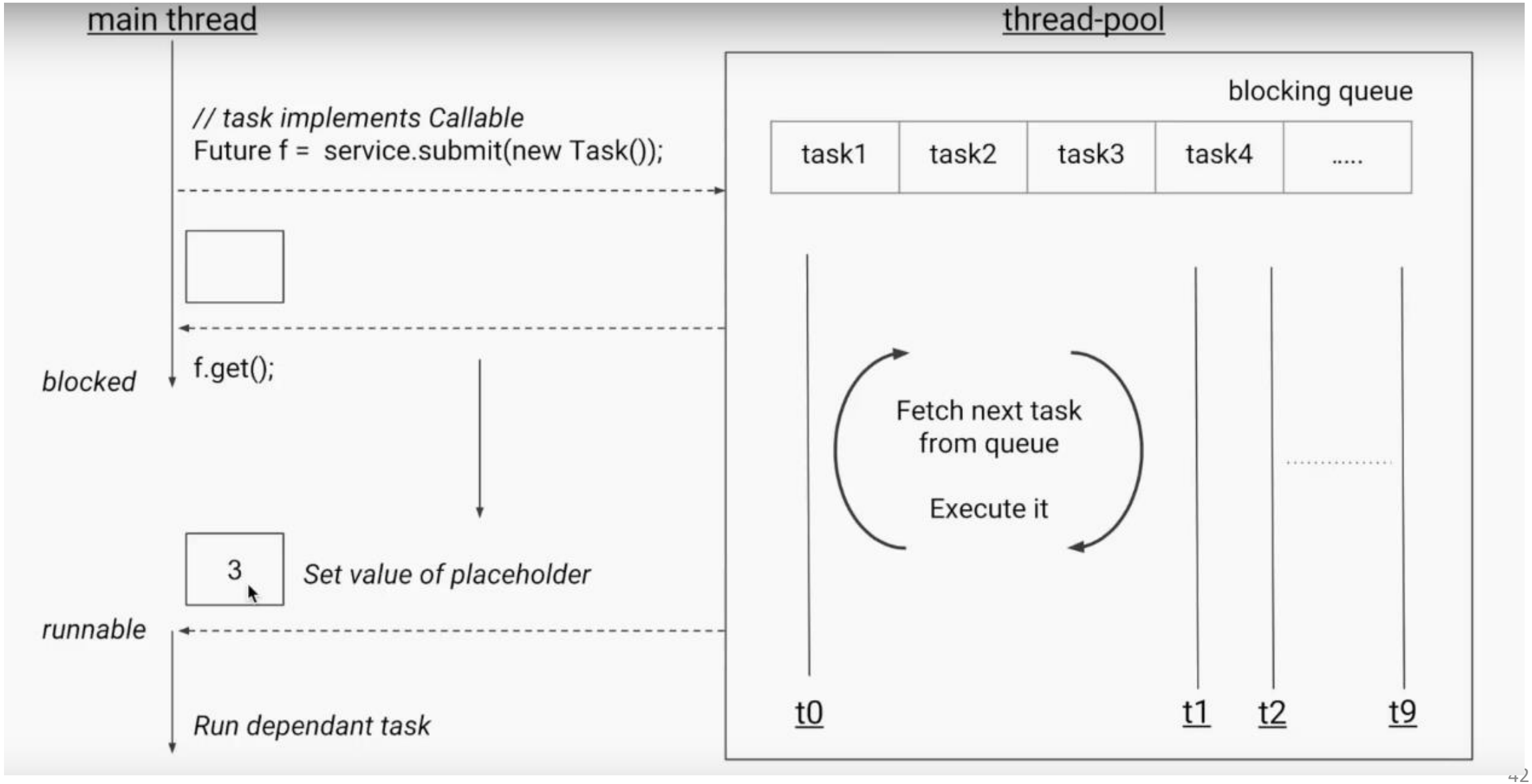
```
Future<Integer> future = new SquareCalculator().calculate(10);

while(!future.isDone()) {
    System.out.println("Calculating...");
    Thread.sleep(300);
}

Integer result = future.get();
```

Future

```
public class Task implements Callable<Integer>{

@Override
public Integer call() throws Exception {
return new Random().nextInt();
}
}
```

```
public class CallableTester {

public static void main(String[] args) {
ExecutorService service= Executors.newFixedThreadPool(10);
//submit task and accept the placeholder object for return value
Future<Integer> future = service.submit(new Task());
try {
//get the task return value
Integer result=future.get();//blocking
System.out.println("Result from task is "+result);
}catch(InterruptedException | ExecutionException e) {
e.printStackTrace();
}
}
}
```

Blocked get()


main thread
thread-pool
// task implements Callable
Future f = service.submit(new Task());
blocking queue
task1
task2
task3
task4
.....
blocked
f.get();
Fetch next task
from queue
Execute it
3
Set value of placeholder
runnable
Run dependant task
t0
t1
t2
t9

Runnable Vs Callable

Both interfaces are designed to represent a task that can be executed by multiple threads. *Runnable* tasks can be run using the *Thread* class or *ExecutorService* whereas *Callables* can be run only using the latter.

The *Runnable* interface is a functional interface and has a single *run()* method which doesn't accept any parameters and does not return any values.
This is suitable for situations where we are not looking for a result of the thread execution,

```
public interface Runnable {
    public void run();
}
```

```
public void executeTask() {
    executorService = Executors.newSingleThreadExecutor();
    Future future = executorService.submit(Runnable object);
    executorService.shutdown();
}
```

Future object will not hold any value

Runnable Vs Callable

The *Callable* interface is a generic interface containing a single *call()* method – which returns a generic value *V*:

```
public interface Callable<V> {
    V call() throws Exception;
}
```

```
public class FactorialTask implements Callable<Integer> {
    int number;

    // standard constructors

    public Integer call() throws InvalidParamaterException {
        int fact = 1;
        // ...
        for(int count = number; count > 1; count--) {
            fact = fact * count;
        }

        return fact;
    }
}
```

The result of *call()* method is returned within a *Future* object:

Thank You!