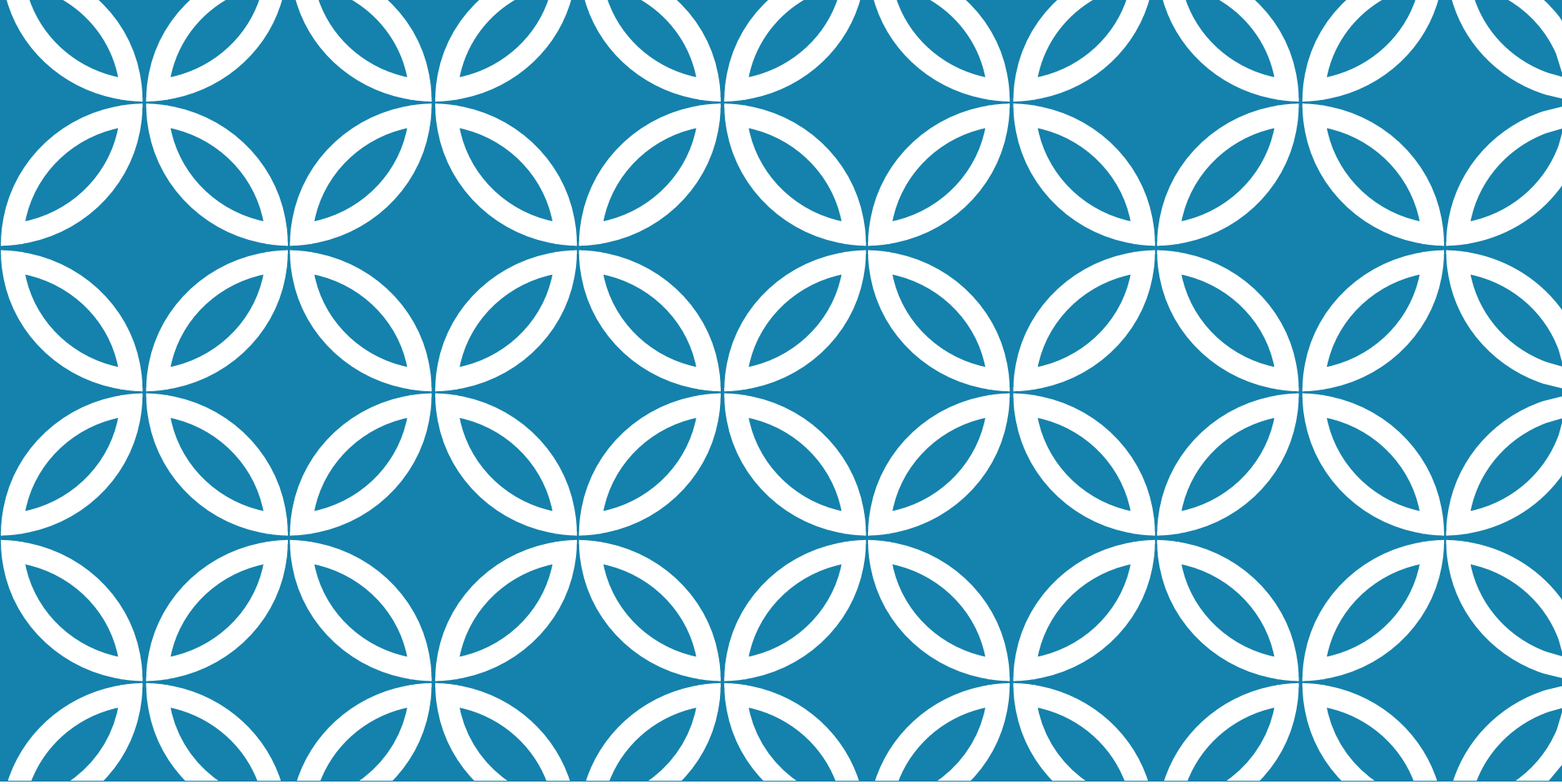


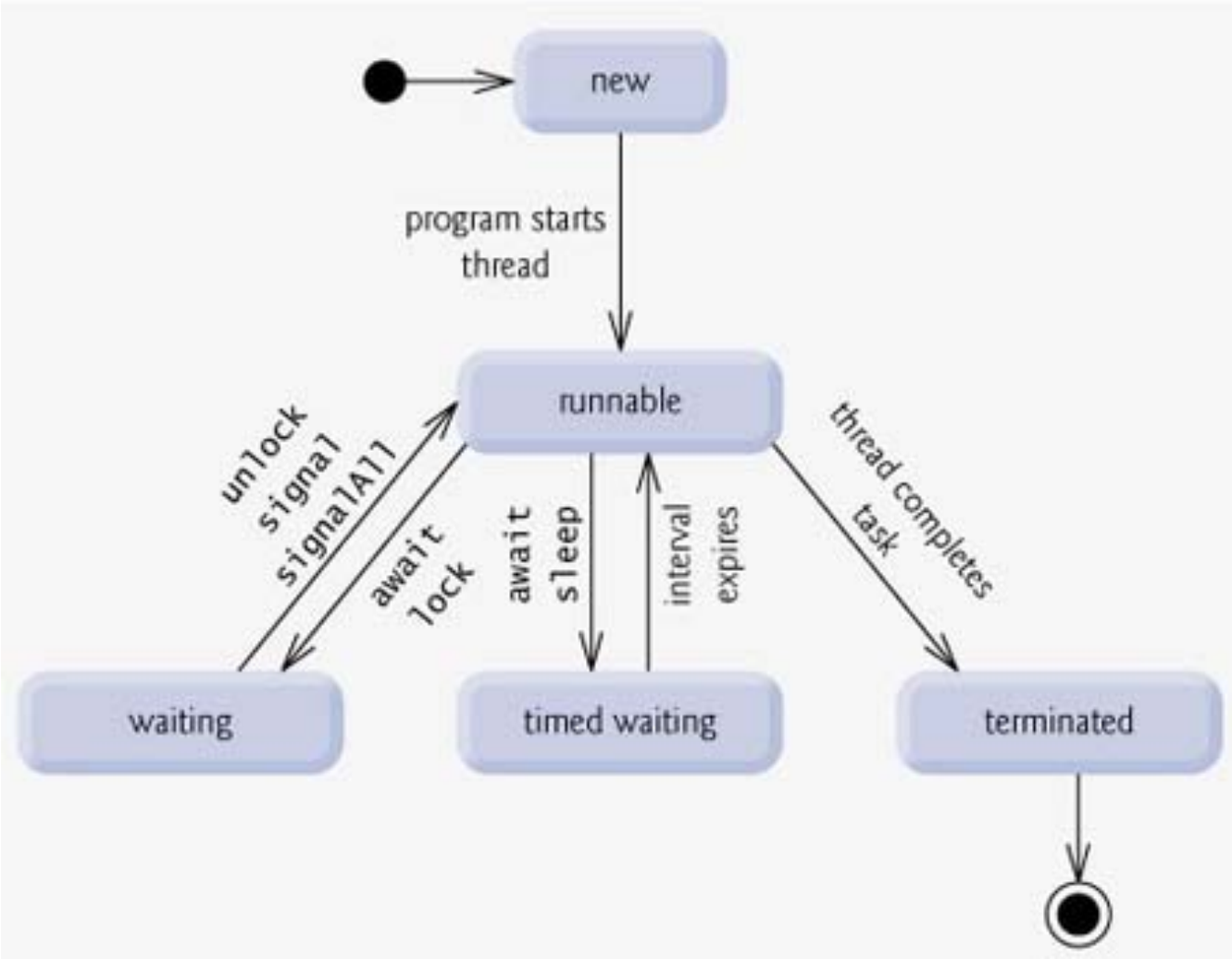
MULTITHREADING

INTRODUCTION

- Java provides built-in support for *multithreaded programming*.
- A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.
- A multithreading is a specialized form of multitasking.
- Multithreading requires less overhead than multitasking processing.

- **Process:** A process consists of the memory space allocated by the operating system that can contain one or more threads.
- A thread cannot exist on its own; it must be a part of a process.
- A process remains running until all of the non-daemon threads are done executing.
- Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

LIFE CYCLE OF A THREAD



- **New:** A new thread begins its life cycle in the new state.
  - It remains in this state until the program starts the thread.
  - It is also referred to as a born state.
- **Runnable:** After a newly born thread is started, the thread becomes runnable.
  - A thread in this state is considered to be executing its task.

- **Waiting:** Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task.
  - A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time.
- A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

THREAD PRIORITIES

- Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.
- Java priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10).
- By default, every thread is given priority NORM_PRIORITY (a constant of 5).
- Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads.
- However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

CREATING A THREAD

- Java defines two ways in which this can be accomplished:
  - By implementing the Runnable interface
  - By extending the Thread class itself.

BY EXTENDING THREAD

- A new class that extends **Thread** is required to be created and then to create an instance of that class.
- The extending class must override the **run( )** method, which is the entry point for the new thread.
- It must also call **start( )** to begin execution of the new thread.

BY IMPLEMENTING RUNNABLE

- The easiest way to create a thread is to create a class that implements the **Runnable** interface.
- To implement Runnable, a class needs to only implement a single method called **run( )** as follows

```
public void run( )
```

- run() can call other methods, use other classes, and declare variables, just like the main thread can.
- After creating a class that implements Runnable, an object of type Thread is to be created from within that class. Thread defines several constructors.

```
Thread(Runnable threadOb, String threadName);
```

- Here, *threadOb* is an instance of a class that implements the Runnable interface and the name of the new thread is specified by *threadName*.
- After the new thread is created, it will not start running until you call its **start( )** method, which is declared within Thread.

```
void start( );
```

METHODS OF THREAD CLASS

#	Methods with Description
1	public void start() Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	public void run() If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	public final void setName(String name) Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	public final void setPriority(int priority) Sets the priority of this Thread object. The possible values are between 1 and 10.

#	Methods with Description
5	public final void setDaemon(boolean on) A parameter of true denotes this Thread as a daemon thread.
6	public final void join(long millisec) The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	public void interrupt() Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	public final boolean isAlive() Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

- The previous methods are invoked on a particular Thread object.
- The following methods in the Thread class are static.
- Invoking one of the static methods performs the operation on the currently running thread.

#	
1	public static void yield() Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
2	public static void sleep(long millisec) Causes the currently running thread to block for at least the specified number of milliseconds.
3	public static boolean holdsLock(Object x) Returns true if the current thread holds the lock on the given Object.
4	public static Thread currentThread() Returns a reference to the currently running thread, which is the thread that invokes this method.
5	public static void dumpStack() Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

THREAD SYNCHRONIZATION

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
- The process by which this synchronization is achieved is called *thread synchronization*.
- The synchronized keyword in Java creates a block of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object.
- To enter a critical section, a thread needs to obtain the corresponding object's lock.

- The general form of the synchronized statement is as follows:

```
synchronized(object) { // statements to be synchronized }
```

- Here, object is a reference to the object being synchronized.
- A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor