# Java

## *Thread*

https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

# Multitasking & Multithreading

- Multitasking allows several activities to occur concurrently on the computer
- A multithreaded program contains two or more parts that can run concurrently
  - Each part of such a program is called a thread
  - Each thread defines a separate path of execution
- Multithreading is a specialized form of multitasking

# Process-based multitasking

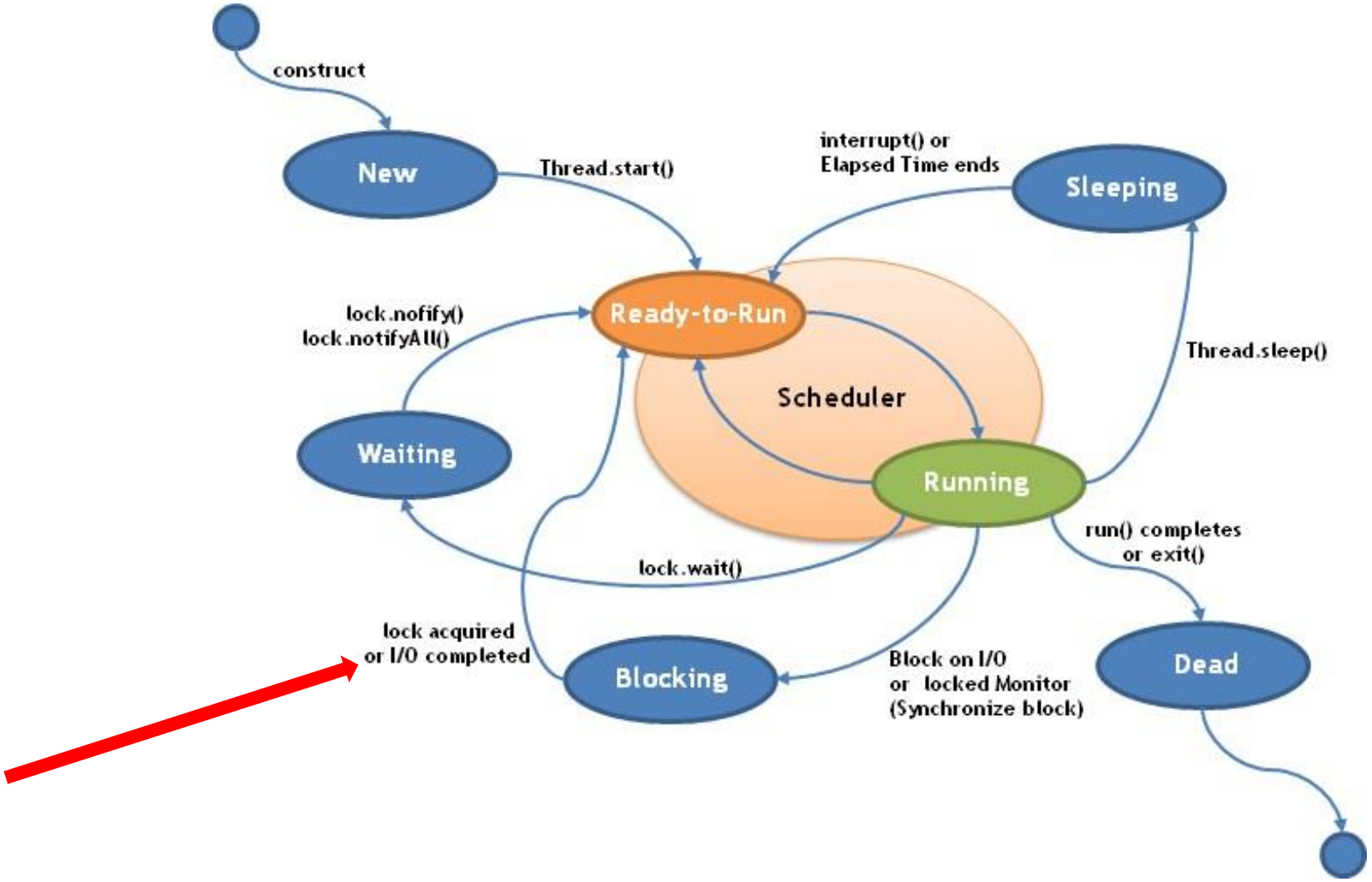
- Allows your computer to run two or more programs (processes) concurrently
  - Enables to run the Java compiler at the same time that you are using a text editor or visiting a web site
- Process is the smallest unit of code that can be dispatched by the scheduler.
- Java makes use of process-based multitasking environments but has no direct control over it

# Thread-based multitasking

- Allows parts of the same process (threads) to run concurrently
  - Thread is the smallest unit of dispatchable code
- A single program can perform two or more tasks simultaneously
  - A text editor can format text at the same time that it is printing (if performed by two separate threads)
- Java supports thread-based multitasking and provides high-level facilities for multithreaded programming

# Thread States

Source: https://avaldes.com/java-thread-states-life-cycle-of-java-threads/



# Main Thread

- When a Java program starts up, one thread begins running immediately
- This is called the main thread of the program
- It is the thread from which the child threads will be spawned
- Often, it must be the last thread to finish execution

# Main Thread

```
1  public class MainThread {
2      public static void main(String[] args) {
3          Thread t = Thread.currentThread();
4          System.out.println("Current thread: " + t);
5          // change the name of the thread
6          t.setName("My Thread");
7          System.out.println("After name change: " + t);
8          try {
9              for(int n = 5; n > 0; n--) {
10                 System.out.println(n);
11                 Thread.sleep( millis: 1000);
12             }
13         }catch (InterruptedException e) {
14             System.out.println("Main thread interrupted");
15         }
16     }
17 }
```

This reference is stored in the local variable ***t***.

Change the internal name of the thread

Specifies the delay period in milliseconds.

This would happen if some other thread wanted to interrupt this sleeping one

Example: MainThread.java

# sleep() method

- Thread pause is accomplished by the sleep() method
  - The argument to sleep() specifies the delay period in milliseconds
- The sleep() method might throw an InterruptedException
  - It would happen if some other thread wanted to interrupt this sleeping one
- The sleep() method causes the thread from which it is called to suspend execution for the specified period of milliseconds

# How to create Thread

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

1. By extending the **Thread** class
2. By implementing **Runnable** Interface

- Extending Thread
  - Need to override the public void run() method
- Implementing Runnable
  - Need to implement the public void run() method
- Which one is better?
  - Implementing Runnable

# Extending Thread

```
1  class NewThread2 extends Thread {
2      NewThread2() {
3          super( name: "Extends Thread");
4          start();
5      }
6      // This is the entry point for the thread.
7      public void run() {
8          try {
9              for(int i = 5; i > 0; i--) {
10                 System.out.println("Child Thread: " + i);
11                 Thread.sleep( millis: 500);
12             }
13         } catch (InterruptedException e) {
14             System.out.println("Child interrupted.");
15         }
16         System.out.println("Exiting child thread.");
17     }
18 }
19
20 public class ExtendsThread {
21     public static void main(String[] args) {
22         new NewThread2();
23     }
24 }
```

Example: ExtendsThread.java

# Implementing Runnable

```
1  class NewThread1 implements Runnable {
2      Thread t;
3      NewThread1() {
4          t = new Thread( target: this);
5          t.start();
6      }
7      // This is the entry point for the thread.
8      public void run() {
9          try {
10             for(int i = 5; i > 0; i--) {
11                 System.out.println("Child Thread: " + i);
12                 Thread.sleep( millis: 500);
13             }
14         } catch (InterruptedException e) {
15             System.out.println("Child interrupted.");
16         }
17         System.out.println("Exiting child thread.");
18     }
19 }
20
21 public class ImplementsThread {
22     public static void main(String[] args) {
23         new NewThread1();
24     }
25 }
```

Example: ImplementsThread.java

# Other ways

```
class NewThread3 implements Runnable {
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep( millis: 500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

public class ImplementsThread2 {
    public static void main(String[] args) {
        Runnable r = new NewThread3();
        Thread t = new Thread(r);
        t.start();
    }
}
```

```
public class CreateThread {
    public static void main(String[] args) {
        CreateThread ct = new CreateThread();
        new Thread(ct::f1, name: "T1").start();
    }

    public void f1() {
        for(int i = 5; i > 0; i--) {
            System.out.println(i);
            try {
                Thread.sleep( millis: 500);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}
```

Example: ImplementsThread2.java

# Multithreading

- Advantages of multithreading
  - Threads share the same address space
  - Context switching and communication between threads is usually inexpensive
- Java works in an interactive, networked environment
  - Data transmission over networks, read/write from local file system, user input - all slower than computer processing
  - In a single-threaded environment, the program has to wait for a task to finish before proceeding to the next
  - Multithreading helps reduce the idle time because another thread can run when one is waiting

# Multithreading in Multicore

- Java's multithreading work in both single-core and multi-core systems
- In single-core systems
  - Concurrently executing threads share the CPU, with each thread receiving a slice of CPU time
  - Two or more threads do not run at the same time, but idle CPU time is utilized
- In multi-core systems
  - Two or more threads do execute simultaneously
  - It can further improve program efficiency and increase the speed of certain operations

# Multiple Threads

- It is possible to create more than one thread inside the main
- In multiple threads, often you will want the main thread to finish last. This is accomplished by
  - using a large delay in the main thread
  - using the **join()** method, this method waits until the thread on which it is called terminates
- Whether a thread has finished or not can be known using **isAlive()** method
- ***Example**: MultipleThreads.java, JoinAliveThreads.java*

# Thread Methods

- start()
- run()
- sleep()
- join()
- getID()
- getName()
- setName()
- getPriority()
- setPriority()
- isAlive()

# Synchronization

# 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 is achieved is called synchronization
- Key to synchronization is the concept of the monitor( also called as semaphore)
- A monitor is an object that is used as a mutually exclusive lock (also known as mutex)
  - Only one thread can own a monitor at a given time

# Synchronization

- When a thread acquires a lock state, it is said to have entered the monitor
- All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor
- These other threads are said to be waiting for the monitor
- A thread that owns a monitor can re-enter the same monitor if it so desires

# Synchronization

- Two ways to achieve synchronization
- Synchronized method

  ***synchronized void call(String msg) { }***
- Synchronized block

  ***public void run() {***

  ***synchronized(target) { target.call(msg); } }***
- ***Example:***

  *Synch.java*

  *Synch1.java*

# Synchronized Method

- All objects have an implicit monitor with them
  - To enter an object's monitor, call a synchronized method
  - All other threads that try to call it (or any other synchronized method) on the same instance have to wait
  - To exit the monitor, the owner returns from the method
- A thread enters any synchronized method on an instance
  - No other thread can enter any other synchronized method on the same instance
  - Non-synchronized methods on that instance will continue to be callable

# Synchronized Statement

- Synchronized methods will not work in all cases
  - To synchronize access to objects of a class not designed for multithreading (class doesn't use synchronized method)
  - No access to the source code, so not possible to synchronized appropriate methods within the class
- How can access to an object of this class be synchronized?
  - Put calls to the methods defined by this class inside a synchronized block

# Inter Thread Communication

- One way is to use polling
  - Loop to check some condition repeatedly, wastes CPU time
  - Once the condition is true, appropriate action is taken
- Java includes an elegant inter-thread communication mechanism via the **wait()**, **notify()** and **notifyAll()** methods
- These methods are implemented as final methods in Object.
- All three methods can be called only from within a synchronized method

# Inter Thread Communication

- Producer-Consumer Problem

Producer → Produces item-1

Consumer → consumes item-1

Producer → Produces item-2

Consumer → consumes item-2

```
// An incorrect implementation of a producer and consumer.
class  Q
{
            int   n;
            synchronized   int   get()
            {
                        System.out.println("Got: " + n);
                        return  n;
            }

           synchronized   void   put(int   n)
           {
                        this.n = n;
                        System.out.println("Put: " + n);
           }
}
```

```
class Producer implements
Runnable
{
        Q    q;
        Producer(Q    q)
         {
          this.q = q;
          new  Thread(this,
"Producer").start();
        }

        public  void  run()
        {
                int  i = 0;
                while(true)
                 {

        q.put(i++);
                }
        }
```

```
class Consumer  implements  Runnable
{
  Q    q;
  Consumer(Q   q)
  {
   this.q = q;
   new  Thread(this, "Consumer").start();
  }
  public  void  run()
  {
          while(true)
          {
                  q.get();
          }
  }
}
```

```
class  PC
{
        public  static  void  main(String   args[])
        {
                Q    q = new  Q();
                new   Producer(q);
                new   Consumer(q);
                System.out.println("Press Control-C to stop.");
        }
}
```

Output

```
Put:  1
        Got:  1
        Got:  1
        Got:  1
        Got:  1
        Got:  1
        Put:  2
        Put:  3
        Put:  4
        Put:  5
        Put:  6
        Put:  7
Got: 7
```

# Correct implementation using wait() and notify()

```
// A correct implementation .
class  Q
 {    int  n;
                boolean   valueSet = false;
                synchronized  int  get()
                {          if(!valueSet)
                              try
                              {    wait();    }


catch(InterruptedException   e)
                              {
System.out.println("Exception ");     }

                System.out.println("Got: " + n);
                              valueSet = false;
                              notify();
                              return  n;
                 }
```

```
synchronized  void  put(int   n)
{     if(valueSet)
          try
          {       wait();     }

          catch(InterruptedException   e)
           {       System.out.println("Exception");    }

         this.n = n;
         valueSet = true;
         System.out.println("Put: " + n);
         notify();
} }
```

```
class Producer implements
Runnable
{
        Q  q;
        Producer(Q q)
        {
         this.q = q;
         new Thread(this,
"Producer").start();
        }
        public void run()
        {
                int  i = 0;
                while(true)
                 {
        q.put(i++);
                }
        }
}
```

```
class Consumer implements Runnable
{
   Q  q;
   Consumer(Q  q)
   {
     this.q = q;
     new Thread(this, "Consumer").start();
   }
   public  void  run()
   {
           while(true)
           {
                   q.get();
           }
   }
}
```

```
class   PCFixed
{          public  static  void  main(String  args[])
                   {
                            Q   q = new   Q();
                            new   Producer(q);
                            new   Consumer(q);
                            System.out.println("Press Control-C to stop.");
                   }
 }
```

# Inter Thread Communication

- ***wait()***
  - tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ) or notifyAll( )
- ***notify()***
  - wakes up a thread that called wait( ) on the same object
- ***notifyAll()***
  - wakes up all the threads that called wait( ) on the same object. One of the threads will be granted access first

# Wait within Loop

- wait() waits until notify() or notifyAll() is called.
- In very rare cases the waiting thread could be awakened due to a spurious wakeup
  - A waiting thread resumes without notify( ) or notifyAll() having been called
  - The thread resumes for no apparent reason
  - Java API documentation recommends that calls to wait() should take place within a loop that checks the condition on which the thread is waiting
  - *Best practice is to use wait() within loop and notifyAll()*