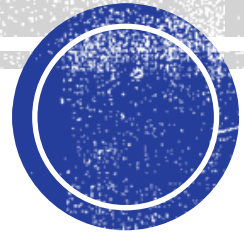
Multi-Threading

Thread- A dispatchable unit of work within a process

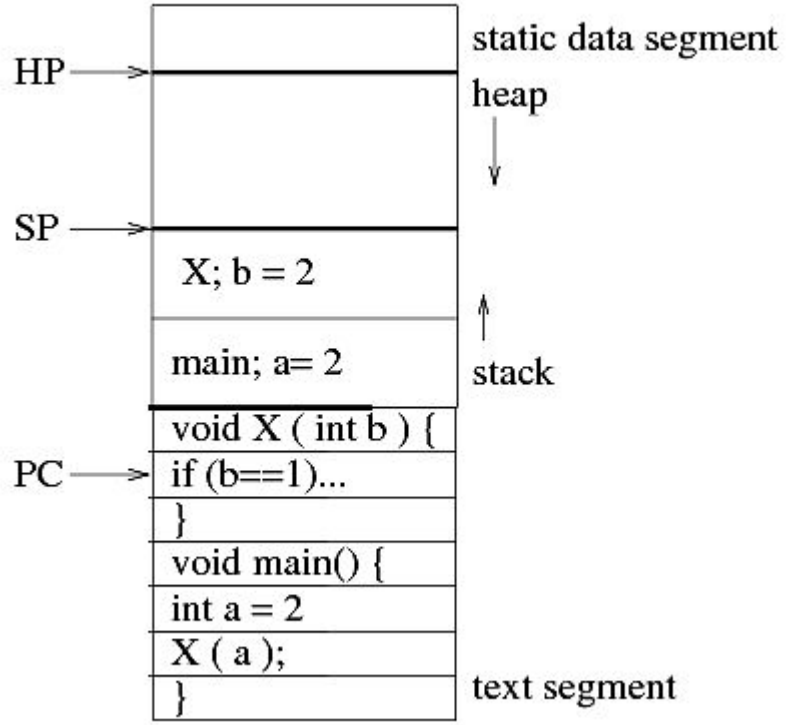


Basics- Program and Process

Program vs Process :

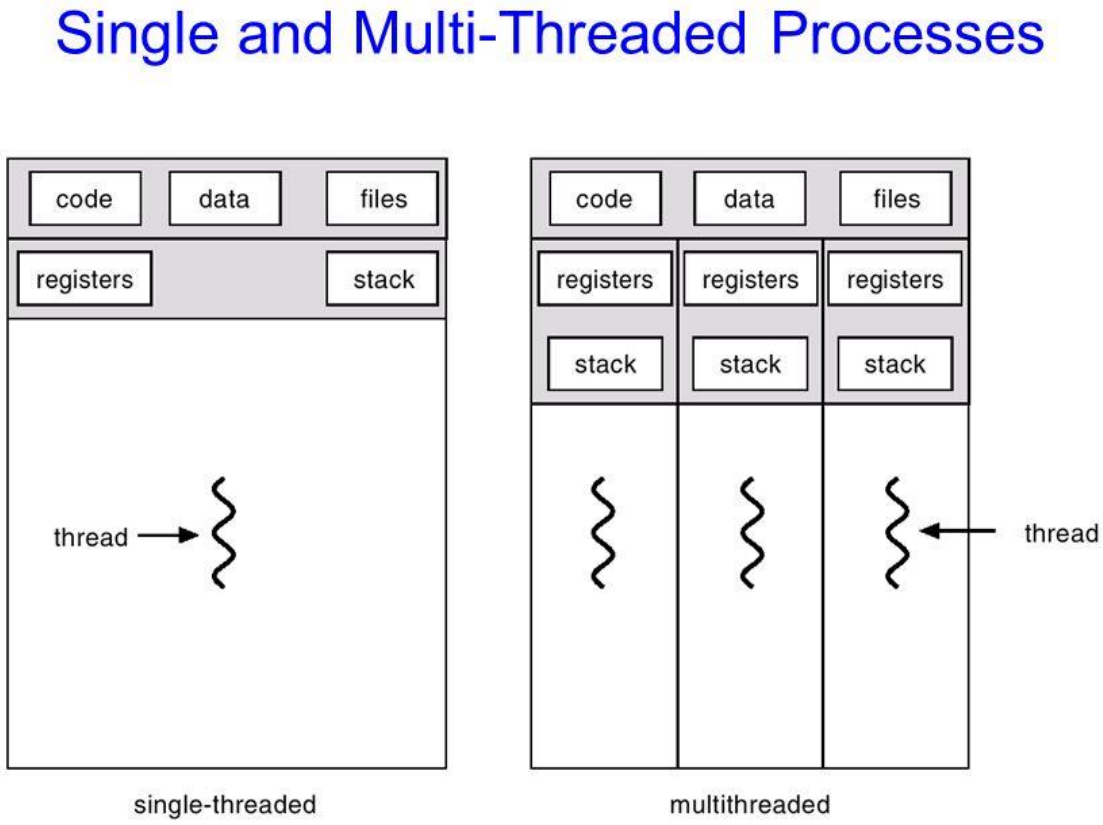
When we execute a program that was just compiled, the OS will generate a process to execute the program.

- A program is a passive entity as it resides in the secondary memory, such as the contents of a file stored on disk. One program can have several processes.
- The term process (Job) refers to program code that has been loaded into a computer's memory so that it can be executed by the central processing unit (CPU).
- A process can be described as an instance of a program running on a computer or as an entity that can be assigned to and executed on a processor. A program becomes a process when loaded into memory and thus is an active entity.
- Process is a dynamic entity and has a high resource requirement, it needs resources like CPU, memory address, I/O during its lifetime. Process has its own control block called Process Control Block.



Multitasking : can be achieved either by using multiprocessing or multithreading

- **Process-based multitasking**
  - is the feature that allows your computer to run two or more programs concurrently.
  - For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor.
  - In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.
- **Thread-based multitasking**
  - The thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously.
  - For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.



A multithreaded program contains two or more parts that can run concurrently

Multitasking Process/Threads

- Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Inter-process communication is expensive and limited. Context switching from one process to another is also costly.
- Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is low cost.
- While Java programs make use of process-based multitasking environments, process-based multitasking is not under the control of Java.
- However, multithreaded multitasking is under control of Java. Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. This is especially important for the interactive, networked environment in which Java operates, because idle time is common

A process is a program in execution.

A process can be divided into a number of small subprocess. Each small subprocess can be addressed as a single thread (a lightweight process).

A thread is a

- Facility to allow multiple activities within a single process
- Referred as lightweight process
- A thread is a series of executed statements
- Each thread has its own program counter, stack and local variables
- A thread is a nested sequence of method calls
- Its shares memory, files and per-process state

What are java threads?

Threads can be thought of as different strands of a program in execution. Each strand is a sequence of instructions to achieve a specific task. A thread scheduler, working at the behest of runtime system, provides the CPU for a quantum of time to each thread.

Thread in Java

In Java, the word thread means two different things.

1. An instance of Thread class.
2. or, A thread of execution.

- Multithreading reduces the CPU idle time that increase overall performance of the system.
- Thread is lightweight process hence it takes less memory and perform context switching
- It helps to share the memory and reduce time of switching between threads.

Why is multithreading relevant?

- Multithreading enables better use of CPU time thereby achieving parallelism and bringing down the time taken for the program execution.
- When a thread is waiting for an I/O task to complete, it is simple idling without using the CPU. Another thread could very well use the CPU during this duration and progress on its task.
- The thread scheduler allocates CPU to each thread for a specified duration so that every thread can make progress in a simultaneous manner.
- Java supports wide set of libraries to support multithreaded programming. Here we will address only the basics. For a more extensive coverage, please look into various resources on the web.

Why Multi-threading?

- Thread has many advantages over the process to perform multitasking.
- Process is heavy weight, takes more memory and occupy CPU for longer time that may lead to performance issue with the system.
- To overcome these issue process is broken into small unit of independent sub-process. These sub-process are called threads that can perform independent task efficiently.
- Hence, computer systems prefer to use thread over the process and use multithreading to perform multitasking.

The main thread

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

1. It is the thread from which other “child” threads will be spawned.
2. Often, it must be the last thread to finish execution because it performs various shutdown actions.

main( ) can be controlled through a Thread object. For that obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread. Its general form is shown here:

static Thread currentThread( )

```
// Controlling the main Thread.
class CurrentThreadDemo {
  public static void main(String args[]) {
     Thread t = Thread.currentThread();

    System.out.println("Current thread: " + t);

    // change the name of the thread
    t.setName("My Thread");
    System.out.println("After name change: " + t);

    try {
      for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
     } catch (InterruptedException e) {
       System.out.println("Main thread interrupted");
    }
  }
}
```

Code showing how to control main thread

Current thread: Thread[main,5,main]

After name change: Thread[My Thread,5,main]

5

4

3

2

1

static void sleep(long milliseconds) throws InterruptedException

- The number of milliseconds to suspend is specified in milliseconds. This method may throw an InterruptedException.
- The sleep( ) method causes the thread from which it is called to suspend execution for the specified period of milliseconds.

final void setName(String threadName)

final String getName( )

- Can set the name of a thread by using setName( ). You can obtain the name of a thread by calling getName( ) (but note that this is not shown in the program). These methods are members of the Thread class.
- In the setName( ) threadName specifies the name of the thread.

sleep( )
setName( )
getName( )

Thread Priorities

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others.
- Thread priorities are integers that specify the relative priority of one thread to another.
- A thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch.*

Synchronization

- Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it.
- For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it.

Messaging/Communication

- After you divide your program into separate threads, you need to define how they will communicate with each other.
- When programming with most other languages, you must depend on the operating system to establish communication between threads.
- Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have. Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out

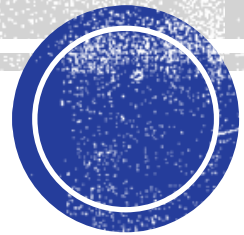
Creating threads

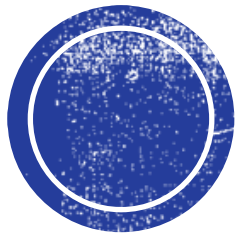
Thread implementation in java can be achieved in two ways:

1. Extending the java.lang.Thread class
2. Implementing the java.lang.Runnable Interface

Multi-Threading

Thread- The smallest until of dispatchable code





Thread creation

Implementing Runnable

Creating threads

Thread implementation in java can be achieved in two ways:

1. Extending the java.lang.Thread class
2. Implementing the java.lang.Runnable Interface

Implementing Runnable

- The easiest way to create a thread is to create a class that implements the Runnable interface.
- Runnable abstracts a unit of executable code.
- Can construct a thread on any object that implements Runnable.
- To implement Runnable, a class need only implement a single method called run( ).

public void run( )

- Inside run( ), you will define the code that constitutes the new thread. It is important to understand that run( ) can call other methods, use other classes, and declare variables, just like the main thread can.
- This thread will end when run( ) returns.

Implementing Runnable

- Create a class that implements the Runnable interface
- Instantiate an object of type Thread from within that class
- After the new thread is created, call its start( ) method, which is declared within Thread. start( ) executes a call to run( ).

Thread defines several constructors.

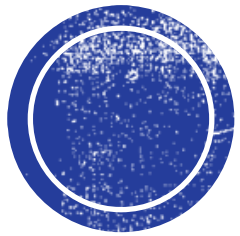
Thread(Runnable threadOb, String threadName)

- threadOb is an instance of a class that implements the Runnable interface
- The name of the new thread is specified by threadName.

```
    // Create a second thread.
    class NewThread implements Runnable {
      Thread t;

      NewThread() {
        // Create a new, second thread
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: " + t);
        t.start(); // Start the thread
      }

      // This is the entry point for the second thread.
      public void run() {
        try {
          for(int i = 5; i > 0; i--) {
            System.out.println("Child Thread: " + i);
            Thread.sleep(500);
          }
      } catch (InterruptedException e) {
        System.out.println("Child interrupted.");
      }
      System.out.println("Exiting child thread.");
    }
}
```

```
class ThreadDemo {
  public static void main(String args[ ] ) {
    new NewThread(); // create a new thread

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {

      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```



Thread creation

Extending Thread

Extending Thread class

- The second way to create a thread is to create a new class that extends Thread, 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.

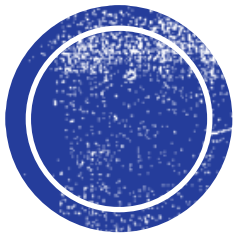
```
class NewThread extends Thread {

  NewThread() {
    // Create a new, second thread
    super("Demo Thread");
    System.out.println("Child thread: " + this);
    start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}
```

```
class ExtendThread {
  public static void main(String args[]) {
    new NewThread(); // create a new thread

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

```java
package threading;
public class Racer extends Thread { // or implements Runnable
private char id;
public Racer(char id) {
this.id = id;
}
public void run() {
for (int i=0; i<10; i++)
System.out.println(id + ":" + i);
}
}

```

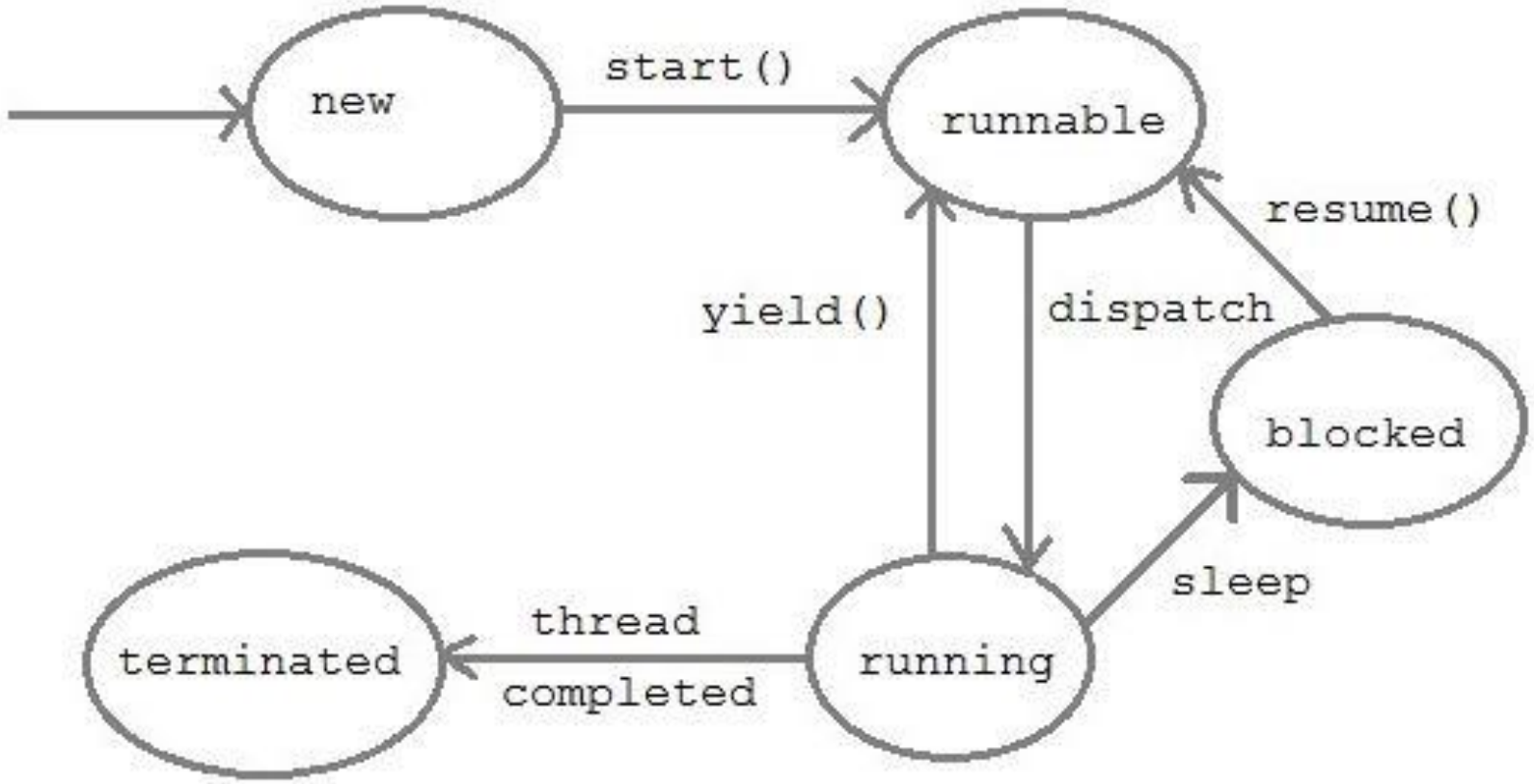
```java
package threading;
//Race.java

public class thread3 {
public static void main(String[] args) {
System.out.println("Main thread started");
Racer r1 = new Racer('A');
r1.start();
Racer r2 = new Racer('B');
r2.start();
System.out.println("Main thread ended");
}

}

```

```
Main thread started
Main thread ended
A:0
A:1
B:0
A:2
B:1
A:3
B:2
A:4
B:3
B:4
B:5
B:6
B:7
A:5
B:8
B:9
A:6
A:7
A:8
A:9
```



Thread life cycle

Thread life cycle



Thread life-cycle

1. New : A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.
2. Runnable : After invocation of start() method on new thread, the thread becomes runnable.
3. Running : A thread is in running state if the thread scheduler has selected it.
4. Waiting : A thread is in waiting state if it waits for another thread to perform a task. In this stage the thread is still alive.
5. Terminated : A thread enter the terminated state when it complete its task.

How can one thread know when another thread has ended?

USING ISALIVE( ) AND JOIN( )

Two ways exist to determine whether a thread has finished.

1. Call isAlive( ) on the thread
2. The method that is commonly use to wait for a thread to finish is called join( )

final boolean isAlive( )

The isAlive( ) method returns true if the thread upon which it is called is still running. It returns false otherwise.

final void join( ) throws InterruptedException

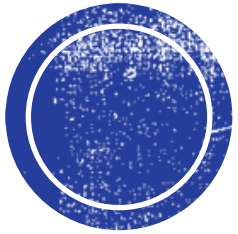
This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it. Additional forms of join( ) allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

```java
package threading;
//Using join() to wait for threads to finis
class NewThread implements Runnable {
String name; // name of thread
Thread t;
NewThread(String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start(); // Start the thread
}
//This is the entry point for thread.

public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println(name + " interrupted.");
}
System.out.println(name + " exiting.");
}
}
```

```java
public class thread1 {
public static void main(String args[]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
NewThread ob3 = new NewThread("Three");
System.out.println("Thread One is alive: "
+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "
+ ob2.t.isAlive());
System.out.println("Thread Three is alive: "
+ ob3.t.isAlive());
//wait for threads to finish
try {
System.out.println("Waiting for threads to finish.");
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Thread One is alive: "
+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "
+ ob2.t.isAlive());
System.out.println("Thread Three is alive: "
+ ob3.t.isAlive());
System.out.println("Main thread exiting.");
}
}
```

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Two: 5
One: 5
Three: 5
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
Two: 4
Three: 4
One: 4
Two: 3
Three: 3
One: 3
Two: 2
One: 2
Three: 2
Two: 1
Three: 1
One: 1
One exiting.
Three exiting.
Two exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

As you can see, after the calls to **join( )** return, the threads have stopped executing.



THREAD PRIORITIES

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.

Thread Priority

- The amount of CPU time that a thread gets often depends on several factors besides its priority.
- A higher-priority thread can also preempt a lower-priority one.
- To set a thread's priority, use the setPriority( ) method, which is a member of Thread.

final void setPriority(int level)

- *level* specifies the new priority setting for the calling thread.
  - The value of level must be within the range MIN_PRIORITY and MAX_PRIORITY. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify NORM_PRIORITY, which is currently 5. These priorities are defined as static final variables within Thread.

You can obtain the current priority setting by calling the **getPriority**( ) method of Thread.[final int getPriority( )]

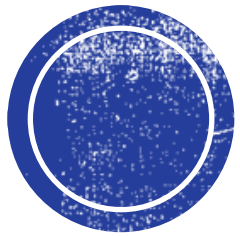
```java
package threading;
//Demonstrate thread priorities.
class clicker implements Runnable {
int click = 0;
Thread t;
private volatile boolean running = true;
public clicker(int p) {
t = new Thread(this);
t.setPriority(p);
}
public void run() {
while (running) {
click++;
}
}
public void stop() {
running = false;
}
public void start() {
t.start();
}
}
```

```java
class thread2 {
public static void main(String args[]) {
Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
lo.start();
hi.start();
try {
Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
lo.stop();
hi.stop();
// Wait for child threads to terminate.
try {
hi.t.join();
lo.t.join();
} catch (InterruptedException e) {
System.out.println("InterruptedException caught");
}
System.out.println("Low-priority thread: " + lo.click);
System.out.println("High-priority thread: " + hi.click);
}
}
```

```
Low-priority thread: 4408112
High-priority thread: 589626904
```

The output of this program, shown as follows indicates that the threads did context switch, even though neither voluntarily yielded the CPU nor blocked for I/O. The higher-priority thread got approximately 90 percent of the CPU time.

The exact output produced by this program depends on the speed of your CPU and the number of other tasks running in the system. When this same program is run under a non-preemptive system, different results will be obtained



Synchronization

```java
package threading;
public class Racer extends Thread { // or implements Runnable
private char id;
public Racer(char id) {
this.id = id;
}
public void run() {
for (int i=0; i<10; i++)
System.out.println(id + ":" + i);
}
}

```

```java
package threading;
//Race.java

public class thread3 {
public static void main(String[] args) {
System.out.println("Main thread started");
Racer r1 = new Racer('A');
r1.start();
Racer r2 = new Racer('B');
r2.start();
System.out.println("Main thread ended");
}

}

```

Why Sychronization?

- In the previous program, the two threads are independent.
- Also, the run() method does the trivial task of iteration count.
- In real scenarios, they have some form of interdependency. i.e. one thread cannot advance more than X number of steps in comparison to other.
- Threads are dependent on each other and make progress in a coordinated manner.
- Several issues arise if they don't coordinate well.

Race conditions in a multithreaded program can result in unpredictable behavior. Why?

- Since threads are part of the same program, the objects in the heap are accessible to all the threads.
- Two or more threads can attempt to modify the same shared object in a simultaneous fashion. This can result in unexpected behavior or incorrect output. Note that the scheduling is arbitrary and not in the programmer's control, such errors can be hard to detect.

A race condition is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.

- Consider the following program on updating the counter. The Incrementer thread increments the counter value by 1 while the Decrementer thread decrements the value by 1. There are four classes.

1. **Counter**: Implements methods increment() and decrement().
2. **Incrementer**: A thread that triggers incrementing by calling Counter's increment().
3. **Decrementer**: A thread that triggers decrementing by calling Counter's decrement().
4. **Main**: Creates an instance of Counter, Incrementer, Decrementer and starts them.

```java
// Incrementer.java
public class Incrementer extends
                              Thread {
  private Counter counter;

  public Incrementer(Counter c) {
    counter = c;
  }

  public void run() {
    while (true)  // run forever
      counter.increment();
  }
}
```

```java
// Decrementer.java
public class Decrementer extends
                              Thread {
  private Counter counter;

  public Decrementer(Counter c) {
    counter = c;
  }

  public void run() {
    while (true)  // run forever
      counter.decrement();
  }
}
```

```java
// Counter.java
public class Counter {
  private int val;

  public Counter()
  {  val = 0;  }

  public void increment()
  {  ++val; print();  }

  public void decrement()
  {  --val; print();  }

  public void print() {
   System.out.print(val);
  }
}
```

```java
// CounterTest.java
public class CounterTest {

  public static void main(String[] a) {

    System.out.println("Main started");

    Counter c = new Counter();
    Incrementer i = new Incrementer(c);
    Decrementer d = new Decrementer(c);

    i.start();
    d.start();

    System.out.println("Main ended");
  }
}
```

Safety Issue?

- The safety issue that can arise is when both the Incrementer (Thread i) and Decrementer (Thread d) call Counter's increment & decrement methods and simultaneously modify val. The following sequence of operations can lead to incorrect result.

1. **Thread i invokes increment().**
2. **Thread i reads val (say val = 10) and is about to increment.**
3. **The scheduler gives the control to Thread d in the meanwhile.**
4. **Thread d invokes decrement().**
5. **Thread d reads the same val. (i.e. 10)**
6. **Thread d decrements val from 10 to 9.**
7. **Thread d calls print() which prints 9.**
8. **The scheduler gives the control to Thread i.**
   (Thread i missed the latest val which is 9 since it has already read val in step 2.)
9. **Thread i increments val from 10 to 11.**
10. **Thread i calls print() which prints 11 (instead of the correct value 10).**

- This is not the only error that can happen. If Thread d does few more decrements before control shifts back to Thread i, then counter value will differ by more.

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.

Suppose we have two different threads T1 and T2, T1 starts execution and save certain values in a file temporary.txt which will be used to calculate some result when T1 returns. Meanwhile, T2 starts and before T1 returns, T2 change the values saved by T1 in the file temporary.txt (temporary.txt is the shared resource). Now obviously T1 will return wrong result.

To prevent such problems, synchronization was introduced. With synchronization in above case, once T1 starts using temporary.txt file, this file will be locked(LOCK mode), and no other thread will be able to access or modify it until T1 returns.

Synchronization

Synchronized Method

Synchronized Block

Using Synchronized Methods

- Key to synchronization is the concept of the monitor (also called a semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex.
- 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. 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 reenter the same monitor if it so desires.

- To enter an object's monitor, just call a method that has been modified with the synchronized keyword.
- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.

```
synchronized(objectidentifier) {
   // Access shared variables and other shared resources
}
```

- The objectidentifier is a reference to an object whose lock associates with the monitor that the synchronized statement represents.

- **Solution to problem of increment decrement:- Mutual exclusion using synchronized construct**
- The solution is to enforce mutual exclusion of threads. When Thread i is incrementing, Thread d should not be allowed to decrement and vice-versa. Java's way of enforcing mutual exclusion is to mark the interfering methods increment() and decrement() as **synchronized**. This will ensure that whenever increment() method is active, decrement() cannot be active and vice-versa.
- Essenstially, the **synchronized** keyword has the effect of locking the Counter object when a (synchronized) method is already active. So, even though control shifts from Thread i to Thread d with active increment(), Thread d cannot invoke decrement() but instead will go to wait state. The scheduler will have to give the control back to Thread i.

```java
// Counter.java with synchronized methods
public class Counter {
  private int val;

  public Counter()
  {  val = 0;  }

  public synchronized void increment()
  {  ++val; print();  }

  public synchronized void decrement()
  {  --val; print();  }

  public void print() {
    System.out.print(val);
  }
}
```

```
class PrintDemo {
   public void printCount() {
      try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Counter   ---   "  + i );
         }
      } catch (Exception e) {
         System.out.println("Thread  interrupted.");
      }
   }
}

class ThreadDemo extends Thread {
   private Thread t;
   private String threadName;
   PrintDemo  PD;

   ThreadDemo( String name,  PrintDemo pd) {
      threadName = name;
      PD = pd;
   }
   public void run() {
      synchronized(PD) {
         PD.printCount();
      }
      System.out.println("Thread " +  threadName + "
exiting.");
   }

   public void start () {
      System.out.println("Starting " +  threadName );
      if (t == null) {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
}
```

```
public class TestThread {

   public static void main(String args[]) {
      PrintDemo PD = new PrintDemo();

      ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
      ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );

      T1.start();
      T2.start();

      // wait for threads to end
      try {
         T1.join();
         T2.join();
      } catch ( Exception e) {
         System.out.println("Interrupted");
      }
   }
}
```

```
Starting Thread - 1
Starting Thread - 2
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 1  exiting.
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 2  exiting.
```

Namah Shivaya!