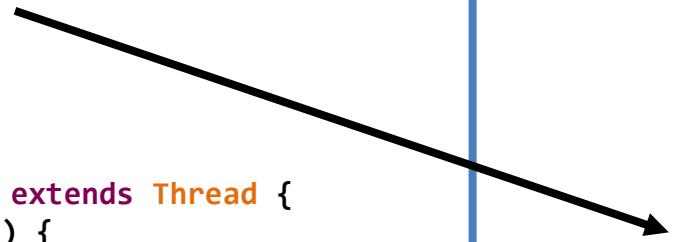
Java multithreading

```
public class MyThread extends Thread {
	public void run() {
		System.out.println("Hello world!");
	}
}
```

```
public class Main {
	public static void main(String[] args) {
		int N = 4;
		MyThread threads[] = new MyThread[N];

		for (int i = 0; i < N; i++) {
			threads[i] = new MyThread();
		}
		for (int i = 0; i < N; i++) {
			threads[i].start();
		}

		System.out.println("Hello from main thread");

		for (int i = 0; i < N; i++) {
			try {
				threads[i].join();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}
```

start method

Threads are started by calling .**start**() method on a Thread object.



```
public class MyThread extends Thread {
      public void run() {
            System.out.println("Hello world!");
      }
}
```

```
public class Main {
      public static void main(String[] args) {
            int N = 4;
            MyThread threads[] = new MyThread[N];

            for (int i = 0; i < N; i++) {
                  threads[i] = new MyThread();
            }
            for (int i = 0; i < N; i++) {
                  threads[i].start();
            }

            System.out.println("Hello from main thread");

            for (int i = 0; i < N; i++) {
                  try {
                        threads[i].join();
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            }
      }
}
```

start method

```
public class MyThread extends Thread {
      public void run() {
            System.out.println("Hello world!");
      }
}
```

```
void start() {
      create_stack_for_new_thread();
      inform_JVM/OS_of_new_thread();
      tell_new_thread_to_execute_run();
}
```

```
public class Main {
      public static void main(String[] args) {
            int N = 4;
            MyThread threads[] = new MyThread[N];

            for (int i = 0; i < N; i++) {
                  threads[i] = new MyThread();
            }
            for (int i = 0; i < N; i++) {
                  threads[i].start();
            }

            System.out.println("Hello from main thread");

            for (int i = 0; i < N; i++) {
                  try {
                        threads[i].join();
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            }
      }
}
```

join method

```
public class MyThread extends Thread {
      public void run() {
            System.out.println("Hello world!");
      }
}
```

```
public class Main {
      public static void main(String[] args) {
            int N = 4;
            MyThread threads[] = new MyThread[N];

            for (int i = 0; i < N; i++) {
                  threads[i] = new MyThread();
            }
            for (int i = 0; i < N; i++) {
                  threads[i].start();
            }

            System.out.println("Hello from main thread");

            for (int i = 0; i < N; i++) {
                  try {
                        threads[i].join();
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            }
      }
}
```

Main thread waits for the other threads to finish by calling .**join**().

It needs to be surrounded by try/catch in case the thread finished because of interrupt.

join method

```
public class MyThread extends Thread {
      public void run() {
            System.out.println("Hello world!");
      }
}
```

```
void join() {
       wait_for_thread_to_signal_it_finished();
}
```

```
public class Main {
      public static void main(String[] args) {
            int N = 4;
            MyThread threads[] = new MyThread[N];

            for (int i = 0; i < N; i++) {
                  threads[i] = new MyThread();
            }
            for (int i = 0; i < N; i++) {
                  threads[i].start();
            }

            System.out.println("Hello from main thread");

            for (int i = 0; i < N; i++) {
                  try {
                        threads[i].join();
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            }
      }
}
```

thread execution

```
public class MyThread extends Thread {
      public void run() {
            System.out.println("Hello world!");
      }
}
```


```
public class Main {
      public static void main(String[] args) {
            int N = 4;
            MyThread threads[] = new MyThread[N];

            for (int i = 0; i < N; i++) {
                  threads[i] = new MyThread();
            }
            for (int i = 0; i < N; i++) {
                  threads[i].start();
            }
            // Main thread continues execution
            System.out.println("Hello from main thread");

            for (int i = 0; i < N; i++) {
                  try {
                        threads[i].join();
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            }
      }
}
```

implements instead of extends

```
public class MyThread extends Thread {
	public void run() {
		System.out.println("Hello world!");
	}
}
```

Why? Because Java accepts that a class can only extend one other. You might need to extend a different class.

```
public class Main {
	public static void main(String[] args) {
		int N = 4;
		MyThread threads[] = new MyThread[N];

		for (int i = 0; i < N; i++) {
			threads[i] = new MyThread();
		}
		for (int i = 0; i < N; i++) {
			threads[i].start();
		}

		System.out.println("Hello from main thread");

		for (int i = 0; i < N; i++) {
			try {
				threads[i].join();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}
```

implements instead of extends

```
public class MyRunnable implements Runnable {
	public void run() {
		System.out.println("Hello world!");
	}
}
```

```
public class Main {
	public static void main(String[] args) {
		int N = 4;
		Thread threads[] = new Thread[N];

		for (int i = 0; i < N; i++) {
			Runnable aux = new MyRunnable();
			threads[i] = new Thread(aux);
		}
		for (int i = 0; i < N; i++) {
			threads[i].start();
		}

		System.out.println("Hello from main thread");

		for (int i = 0; i < N; i++) {
			try {
				threads[i].join();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}
```

Java private/shared

- In Java there is **no** concept of private and shared variables.

- If a thread has the reference to an object it can modify it (All Java Objects are on the heap – the stack is for primitives and method calls). The result is visible by the other threads (with limitations: see volatile).

(equivalent to shared in OpenMP)

- All primitive variables that are on the stack can only be accessed by the thread which owns that stack.

(equivalent to private in OpenMP)

Concurrency problems

```
i = 0;
```

thread 1

```
i = i + 2;
```

thread 2

```
i = i + 2;
```

```
i is 2 or 4
```

Concurrency problems

```
i = 0;
```

thread 1

```
mov eax i;
add eax 2;
mov i eax;
```

thread 2

```
mov eax i;
add eax 2;
mov i eax;
```

```
i is 2 or 4
```

Concurrency problems

```
i = 0;
```

thread 1

```
mov eax i;

add eax 2;

mov i eax;
```

thread 2

```
mov eax i;

add eax 2;
mov i eax;
```

i is **2** or 4

Concurrency problems

```
i = 0;
```

thread 1

```
mov eax i;
add eax 2;
mov i eax;
```

thread 2

```
mov eax i;
add eax 2;
mov i eax;
```

i is 2 or **4**

Solution: synchronized

```
i = 0;
```

thread 1

```
synchronized (object) {
        i = i + 2;
}
```

thread 2

```
synchronized (object) {
        i = i + 2;
}
```

```
i is 4
```

synchronized with object

```
synchronized (object) {
    i = i + 2;
}
```

equivalent to:

```
lock (object)
    i = i + 2;
unlock (object)
```

```
object can be any object
object can be this
object can be a.class
```

synchronized method

Critical section between all threads that call the method from **the same** object

```
increment() {
    synchronized (this) {
        i = i + 2;
    }
}
```

```
synchronized increment() {
        i = i + 2;
}
```

Critical section between all threads that call the method from class Name

```
increment() {
    synchronized (Name.class) {
        i = i + 2;
    }
}
```

```
static synchronized increment() {
        i = i + 2;
}
```

volatile

```
volatile int i;
```

All writes to i must be seen by all threads

It is slower

```
int i;
```

i can be cached by a thread (maybe even in CPU registers)

All operations that involve i are faster