Programare avansata pe obiecte - laborator 10 (231)

Butan Silvia
silvia.butan@endava.com
butan.silvia@gmail.com

Object oriented programming - lab 10

Thread Fundamentals

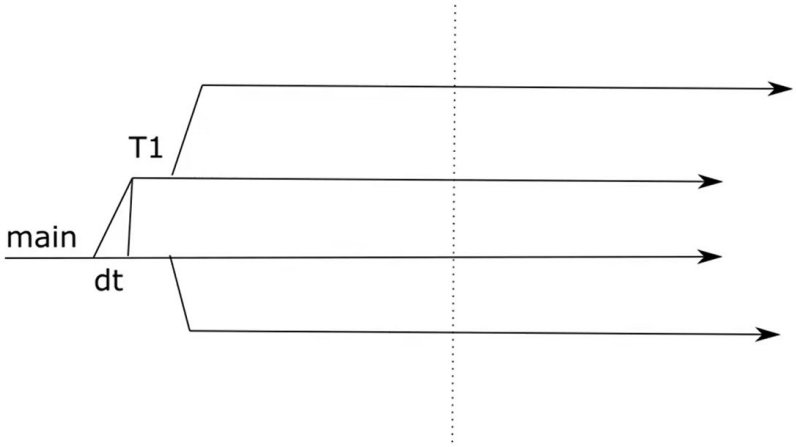
- Java provides built-in support for **multithreading**.
- A thread is actually a lightweight process. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a **thread** and each thread defines **a separate path of the execution**.
- Multithreading is a specialized form of multitasking.

Thread => set of instructions executed in a clear, defined order and independent and parallel of other sets of instructions of the same application program.

When we have only one thread in our program => we call it "***a single threaded program***". In Java the first thread, the one that starts with the main method is also called **main** thread.

When we say that we have a single threaded application in Java => it means that we have an application that starts from the main method and doesn't create any other threads. We discuss here only about the threads created by us in the application program. There are also other threads managed by the JVM, such as the thread on which runs the java garbage collector.

The Java run-time system depends on threads for many things. Threads reduce inefficiency by preventing the waste of CPU cycles.

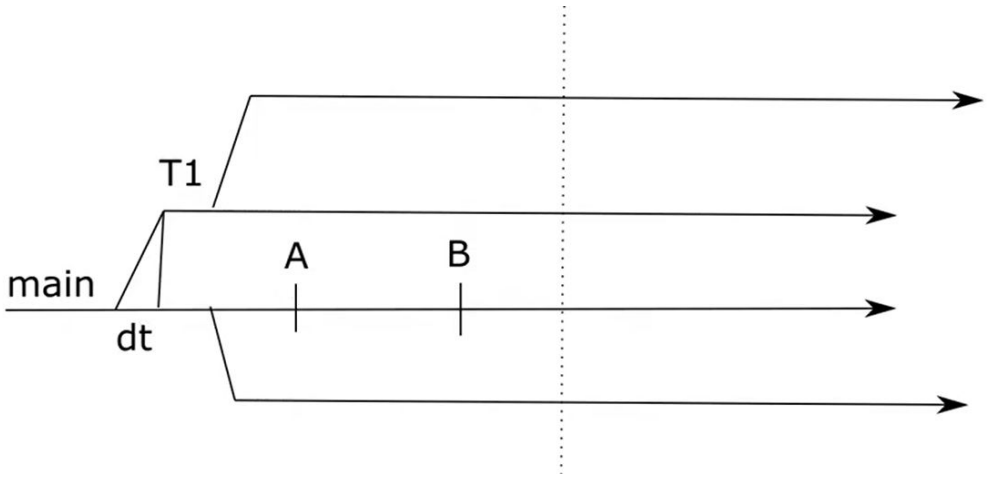


When starting new threads, there is a delay from the moment we start the thread and the moment it starts running (dt - usually milliseconds).

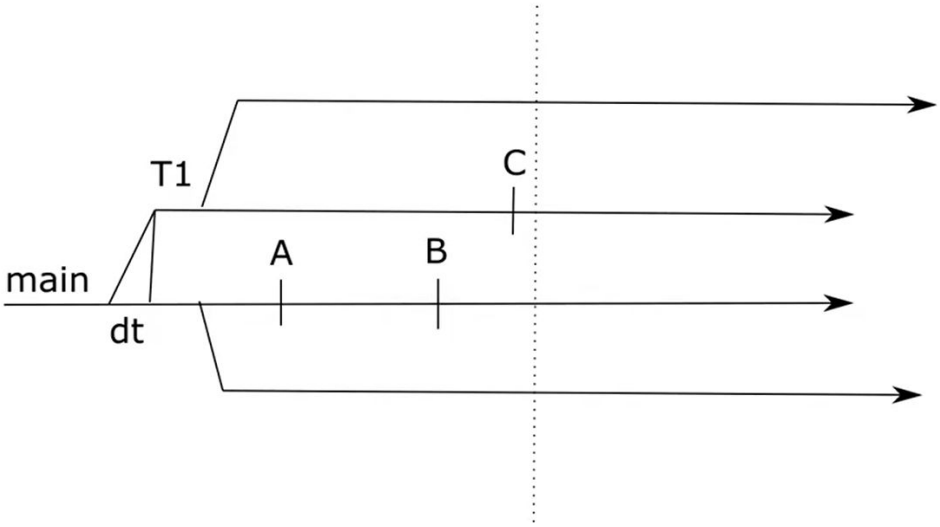
In the image above, there are 4 threads running in parallel => we might have 4 instructions executing at the same time.

The order of instructions:

- when we have one thread => the instructions are always in the order we execute them
    - Eg: On the thread **main** instruction **A** will always be first and instruction **B** second when executing.



- but if we have an instruction C on thread T1, we have no information on when it will be executed in relation to instructions A and C.



Defining and Starting a Thread

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

The **Thread** class defines a number of methods useful for thread management. These include static methods, which provide information about, or affect the status of, the thread invoking the method.

There are two ways to do this:

1. **Subclass Thread**

   The Thread class itself implements Runnable, though its run method does nothing. An application can subclass Thread, providing its own implementation of run, as in the HelloThread example:

```
package com.paolabs.lab10;

public class HelloThread extends Thread {

   // main method of a thread
   @Override
   public void run() {
       System.out.println("Hello from a thread!");
   }
}
```

```
package com.paolabs.lab10;

public class Main {

   // starting point of the main thread
   public static void main(String[] args) {
       HelloThread helloThread = new HelloThread();
       helloThread.start(); // you want your thread here to begin the
execution
   }
}
```

Possible outcomes:

```
End main!
Hello from a thread!
```

or:

```
Hello from a thread!
End main!
```

This approach is easier to use in simple applications, but is limited by the fact that your task class must be a descendant of Thread.

2. **Provide a Runnable object**

   The Runnable interface defines a single method, run, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor, as in the HelloRunnable example:

```
package com.paolabs.lab10.ex2;

public class HelloRunnable implements Runnable {

   @Override
   public void run() {
       System.out.println("Hello from another thread!");
   }
}
```

```
package com.paolabs.lab10.ex2;

public class Main {

   // starting point of the main thread
   public static void main(String[] args) {
       Thread helloThread = new Thread(new HelloRunnable());
       helloThread.start();

       System.out.println("End main!");
   }
}
```

Possible outcomes:

```
End main!
Hello from another thread!
```

or:

```
Hello from another thread!
End main!
```

This approach is more general, because the Runnable object can subclass a class other than Thread.

Threads exist in several states:

- **New** - When we create an instance of Thread class, a thread is in a new state.
- **Running** - The Java thread is in a running state.
- **Suspended** - A running thread can be suspended, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off.
- **Blocked** - A Java thread can be blocked when waiting for a resource.
- **Terminated** - A thread can be terminated, which halts its execution immediately at any given time. Once a thread is terminated, it cannot be resumed.

Pausing Execution with Sleep:

- Thread.sleep causes the current thread to suspend execution for a specified period.
- This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system.
- The sleep method can also be used for pacing;

```
package com.paolabs.lab10.ex3;

import java.util.Arrays;
import java.util.List;

public class SleepMessages {

   public static void main(String[] args) throws InterruptedException {
       List<String> info = Arrays.asList("Hello", "there", "Hi", "again!");

       for (String s : info) {
```

```
            //Pause for 4 seconds
            Thread.sleep(4000);
            //Print a message
            System.out.println(s);
        }
    }
}
```

Interrupts:

- An interrupt is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate.
- A thread sends an interrupt by invoking interrupt on the Thread object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.

Supporting Interruption:

- If the thread is frequently invoking methods that throw **InterruptedException**, it simply returns from the run method after it catches that exception. For example, suppose the central message loop in the SleepMessages example were in the run method of a thread's Runnable object. Then it might be modified as follows to support interrupts:

```
package com.paolabs.lab10.ex3;

import java.util.Arrays;
import java.util.List;

public class SleepMessagesWithInterrupts {

    public static void main(String[] args) throws InterruptedException {
        List<String> info = Arrays.asList("Hello", "there", "Hi", "again!");

        for (String s : info) {
            // Pause for 4 seconds
            try {
                Thread.sleep(4000);
            } catch (InterruptedException e) {
                // We've been interrupted: no more messages.
                return;
```

```
            }
            // Print a message
            System.out.println(s);
        }
    }
}
```

- Many methods that throw InterruptedException, such as sleep, are designed to cancel their current operation and return immediately when an interrupt is received.

- What if a thread goes a long time without invoking a method that throws InterruptedException? Then it must periodically invoke Thread.interrupted, which returns true if an interrupt has been received. For example:

```
package com.paolabs.lab10.ex5;

import java.util.Arrays;
import java.util.List;

public class InterruptEx {

    public static void main(String[] args) {
        List<String> info = Arrays.asList("Hello", "there", "Hi", "again!");

        for (String s : info) {
            // Print a message
            System.out.println(s);
            if (Thread.interrupted()) {
                // We've been interrupted: no more printing.
                return;
            }
        }
    }
}
```

Joins

- The join method allows one thread to wait for the completion of another. If t is a Thread object whose thread is currently executing, t.join(); causes the current thread to pause execution until t's thread terminates.

Synchronization

- Threads communicate primarily by sharing access to fields and the objects reference fields refer to. This form of communication is extremely efficient, but makes two kinds of errors possible: thread interference and memory consistency errors. The tool needed to prevent these errors is synchronization.

- However, synchronization can introduce thread contention, which occurs when two or more threads try to access the same resource simultaneously and cause the Java runtime to execute one or more threads more slowly, or even suspend their execution. Starvation and livelock are forms of thread contention.

Thread Interference

- Interference happens when two operations, running in different threads, but acting on the same data, interleave. This means that the two operations consist of multiple steps, and the sequences of steps overlap.

Synchronized Methods

- The Java programming language provides two basic synchronization idioms: synchronized methods and synchronized statements.

To make a method synchronized, simply add the synchronized keyword to its declaration:

```
package com.paolabs.lab10.ex6;

public class SynchronizedCounter {

   private int c = 0;

   public synchronized void increment() {
       c++;
   }

   public synchronized void decrement() {
       c--;
   }

   public synchronized int value() {
       return c;
   }
```

```
}
```

=> it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block suspend execution until the first thread is done with the object.
When a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Note that constructors cannot be synchronized - using the synchronized keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

Intrinsic Locks and Synchronization

Synchronization is built around an internal entity known as the intrinsic lock.

Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.

Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to acquire the object's intrinsic lock before accessing them, and then release the intrinsic lock when it's done with them. A thread is said to own the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquisition of the same lock.

Synchronized Statements

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

Deadlock

- Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.

```
public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s"
                + "  has bowed to me!%n",
                this.name, bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s"
                + " has bowed back to me!%n",
                this.name, bower.getName());
        }
    }

    public static void main(String[] args) {
        final Friend alphonse =
            new Friend("Alphonse");
        final Friend gaston =
            new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}
```