

Working with Threads

Creating new threads

- A process (=a running program) has one main thread that starts at the main() method
- But a process can also create new threads that run in parallel

```
public static void main(String[] args) {
```

```
    Thread t = new Thread(() -> {  
        System.out.println("Hello world");  
        String name = "This is "+Thread.currentThread().getName();  
        System.out.println(name);  
    }, "My new thread");
```

```
    t.start();
```

```
    int name = "This is "+Thread.currentThread().getName();  
    System.out.println(name);
```

```
}
```

The Thread constructor takes a Runnable interface with a single method

The name of the new thread (optional argument)


Program execution by two threads

Main thread

```
Thread t = new Thread(() -> { ... }, "My new thread");
```

```
t.start();
```


```
int name = "This is "+Thread.currentThread().getName();  
System.out.println(name);
```



Main thread
ends here

My new thread

```
System.out.println("Hello w  
int name = "This is "+Threa  
System.out.println(name);
```



Second thread
ends here

- A process ends if all its threads have ended

Creating threads (different style)

- You often see programmers putting the code for a new thread in a separate class to make it more readable. This is useful if each thread needs its own data. In this example, each thread has a MyThread object:

```
class MyThread implements Runnable {
    private String text;

    public MyThread(String text) {
        this.text = text;
    }
    @Override
    public void run() {
        System.out.println(text);
    }
}
```

```
Thread t1=new Thread(new MyThread("Hello"));
t1.start();
```

```
Thread t2=new Thread(new MyThread("World"));
t2.start();
```

Waiting for threads to finish

- Sometimes, you want that a thread waits until another thread has finished

```
public static void main(String[] args) {
    Thread t = new Thread(() -> {
        try {
            Thread.sleep(5000);
        }
        catch (InterruptedException e) {
            throw new RuntimeException("Unexpected interrupt", e);
        }
        System.out.println("New thread has finished its work");
    });
    t.start();

    System.out.println("This is the main thread");

    try {
        t.join();
    }
    catch (InterruptedException e) {
        throw new RuntimeException("Unexpected interrupt", e);
    }

    System.out.println("All threads have finished");
}
```

Doing nothing interesting. Just sleeping 5000ms

The main thread waits until the new thread has finished

sleep() and join() can throw an InterruptedException. This happens if the thread is interrupted by something

Waiting for threads to finish (2)

- You can specify how long you want to wait for a thread to finish:

```
t.join(10000); // wait 10 seconds maximum
// check whether the thread has finished:
if(t.isAlive()) {
    // the other thread is still running (or not yet started)
}
```

Why using threads?

Threads can be useful in two situations:

1. You want to do something that will probably take a lot of time and you don't want to block the rest of the program
2. You want to speed up computation

Use threads for non-blocking operations

■ Bad:

```
 JButton button=new JButton("Press me!");  
 button.addActionListener(  
     (ActionEvent e) ->  
         ...do something that takes a lot of time (for example reading a file)...  
 );  
 frame.add(button);
```

The entire window will freeze until the function has completed!

■ Good:

```
 JButton button=new JButton("Press me!");  
 button.addActionListener(  
     (ActionEvent e) ->  
         ...create a thread to do something that takes a lot of time...  
 );  
 frame.add(button);
```


Use threads to speed-up computation

- Example: Sum of all natural numbers from 1 to 1000:

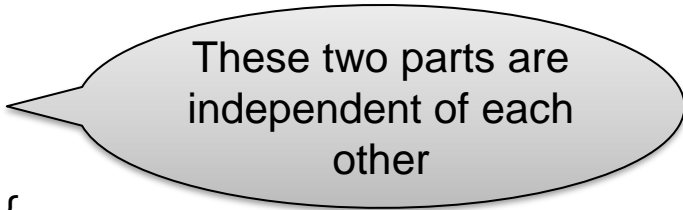
```
int sum = 0;
for(int i=1; i<=1000; i++) {
    sum+=i;
}
```

- The calculation can be divided into two parts:

```
int sum1 = 0;
for(int i=1; i<=500; i++) {
    sum1+=i;
}

int sum2 = 0;
for(int i=501; i<=1000; i++) {
    sum2+=i;
}

int sum = sum1+sum2;
```



These two parts are independent of each other

Adding numbers with threads

```
class Sum implements Runnable {
    final int a,b;
    int sum;

    public Sum(int a, int b) { this.a = a; this.b = b; }

    public void run() {
        for(int i=a;i<=b;i++) {
            sum += i;
        }
    }
}

public static void main(String[] args) throws InterruptedException {
    Sum s1 = new Sum(1,500);
    Sum s2 = new Sum(501,1000);
    Thread t1 = new Thread(s1); // create two threads
    Thread t2 = new Thread(s2);

    t1.start(); t2.start(); // start both threads
    t1.join(); t2.join(); // wait until both threads have finished
    int sum = s1.sum + s2.sum;
}
```

How many threads do we need?

- We could use 100 threads instead of 2 to calculate the sum from 1 to 1000. Does that mean our program becomes 100 times faster?
- No. On a modern computer, creating a simple thread (without any extra objects) takes around 0.05-0.1 ms. That's approximately the time to calculate the sum from 1 to 100 000.
- Conclusion: Threads only improve the speed of a program if the tasks for the threads are longer than the overhead to create and manage them

Futures and thread pools

Futures

- Our Sum class is an example for a Runnable that gives a result: the sum of the natural numbers from a to b

```
class Sum implements Runnable {
    final int a,b;
    int sum;                // <- the result of the Runnable

    public Sum(int a, int b) { this.a = a; this.b = b; }

    public void run() {
        for(int i=a;i<=b;i++) {
            sum += i;
        }
    }
}
```

- This is a very typical pattern. In Java, a Runnable that calculates a result is called a *Future*
- Of course, there are already some packages that help you working with Futures 😊

Working with `java.util.concurrent.Future`

```
public static int calculate(int a, int b) {  
    int sum = 0;  
    for(int i=a;i<=b;i++) {  
        sum += i;  
    }  
    return sum;  
}
```

`Future.get()` throws an `InterruptedException` if interrupted while waiting or an `ExecutionException` if there was a problem in the calculation.

```
public static void main(String[] args)  
    throws ExecutionException, InterruptedException {
```

This is a thread pool. It has two threads that are waiting for work.

```
ExecutorService executor = Executors.newFixedThreadPool(2);
```

```
Future<Integer> f1 = executor.submit(() -> calculate(1, 500));
```

```
Future<Integer> f2 = executor.submit(() -> calculate(501, 1000));
```

```
int sum = f1.get() + f2.get();  
executor.shutdown();
```

`Future.get()` waits until the thread has finished.

We give the threadpool two tasks to do.

```
}
```

How do thread pools work?

- A thread pool is a group of threads that are ready to work. In our example, we have created a threadpool with two threads:

```
ExecutorService executor = Executors.newFixedThreadPool(2);
```

- Threads in threadpool are like chefs in the kitchen of a restaurant waiting for orders. If you submit a task to the pool, one of the threads will take the task and it will immediately start working on it:

```
Future<Integer> f1 = executor.submit(...); // executed by thread 1
Future<Integer> f2 = executor.submit(...); // executed by thread 2
```

- You can submit more tasks, but they will wait until one of the previous tasks has finished:

```
Future<Integer> f1 = executor.submit(...); // executed by thread 1.
Future<Integer> f2 = executor.submit(...); // executed by thread 2.
Future<Integer> f3 = executor.submit(...);
// the submit method returns immediate, but the execution of f3 will
// wait until f1 or f2 is finished
```

How do thread pools work? (2)

- You can get the result of a future with `get()`:

```
f1.get()
```

If the task is not yet finished, the method `get()` will wait.

- When you don't need the thread pool anymore, you have to shut it down to stop all its threads:

```
executor.shutdown();
```