

# Java Fundamentals

Multithreading



# Threads und Runnables

- Multithreading ist fest in Java integriert
- Wird benötigt für
  - Parallelverarbeitung
  - Serveranwendungen



```
public class ThreadStarter {  
    public static void main(String[] args) {  
        Runnable runnable = () -> {  
            for (int i = 0; i < 10;i++) {  
                try {  
                    Thread.sleep(1000L);  
                    System.out.println("Thread " +  
                        Thread.currentThread().getName() +  
                        " counter:" + i);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        };  
  
        new Thread(runnable).start();  
        new Thread(runnable).start();  
        System.out.println("Main thread says good bye");  
    }  
}
```



Main thread says good bye  
Thread Thread-1 counter:0  
Thread Thread-0 counter:0  
Thread Thread-0 counter:1  
Thread Thread-1 counter:1  
Thread Thread-0 counter:2  
Thread Thread-1 counter:2  
Thread Thread-1 counter:3  
Thread Thread-0 counter:3  
Thread Thread-1 counter:4  
Thread Thread-0 counter:4  
Thread Thread-0 counter:5  
Thread Thread-1 counter:5  
Thread Thread-1 counter:6  
Thread Thread-0 counter:6  
Thread Thread-0 counter:7  
Thread Thread-1 counter:7  
Thread Thread-0 counter:8  
Thread Thread-1 counter:8  
Thread Thread-0 counter:9  
Thread Thread-1 counter:9



# Races Conditions

```
public class Stack {  
    int stacksize=0;  
    String[] stackArray = new String[10_000];  
  
    public void push(String data) {  
        stackArray[stacksize] = data;  
        stacksize++;  
    }  
  
    public String pop() {  
        stacksize--;  
        return stackArray[stacksize];  
    }  
}
```



# Race Conditions

```
Stack stack = new Stack();  
stack.push("Eins"); stack.push("Zwei"); stack.push("Drei");
```

Erwartung: `stacksize=3, stackArray=["Eins", "Zwei", "Drei"]`

Start	->	<code>stacksize=0, stackArray=[]</code>
Thread 1: <code>push("Eins"), stackArray[0]="Eins"</code>	->	<code>stacksize=0, stackArray=["Eins"]</code>
Thread 1: <code>push("Eins"), stacksize++</code>	->	<code>stacksize=1, stackArray=["Eins"]</code>
Thread 1: <code>push("Zwei"), stackArray[1]="Zwei"</code>	->	<code>stacksize=1, stackArray=["Eins", "Zwei"]</code>
Thread 2: <code>push("Drei"), stackArray[1]="Drei"</code>	->	<code>stacksize=1, stackArray=["Eins", "Drei"]</code>
Thread 1: <code>push("Zwei"), stacksize++</code>	->	<code>stacksize=2, stackArray=["Eins", "Drei"]</code>
Thread 2: <code>push("Drei"), stacksize++</code>	->	<code>stacksize=3, stackArray=["Eins", "Drei"]</code>

Ergebnis: `stacksize=3, stackArray=["Eins", "Drei"]`



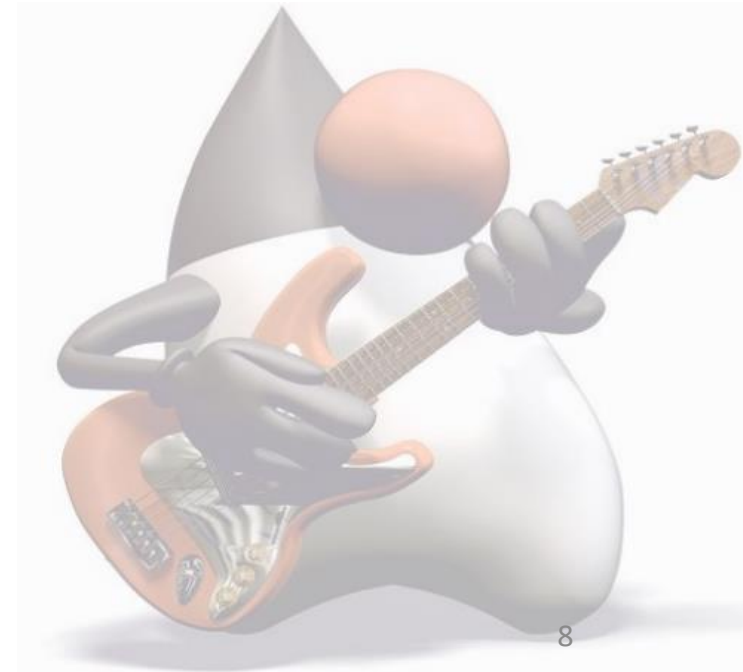
# Ein thread-sicherer Stack

```
public class Stack {  
    int stacksize=0;  
    String[] stackArray = new String[10_000];  
  
    synchronized public void push(String data) {  
        stackArray[stacksize] = data;  
        stacksize++;  
    }  
  
    synchronized public String pop() {  
        stacksize--;  
        return stackArray[stacksize];  
    }  
}
```



# Ein thread-sicherer Stack

```
public class Stack {  
    int stacksize=0;  
    String[] stackArray = new String[10_000];  
  
    Object ampel = new Object();  
  
    public void push(String data) {  
        synchronized(ampel) {  
            stackArray[stacksize] = data;  
            stacksize++;  
        }  
    }  
  
    public String pop() {  
        synchronized(ampel) {  
            stacksize--;  
            return stackArray[stacksize];  
        }  
    }  
}
```



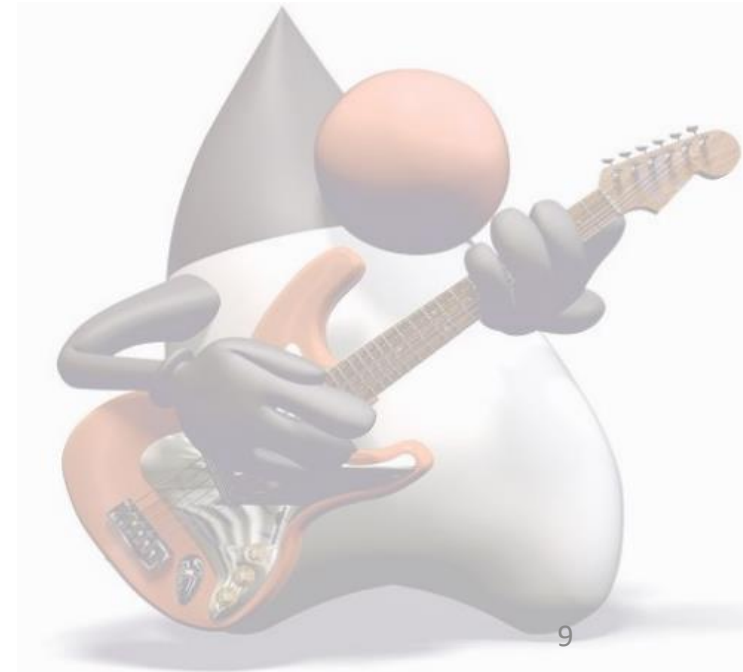


# Primitive und Multithreading

- Sind primitive Datentypen thread-sicher?

```
public class Counter {  
    private int a;  
  
    public int count() {  
        return a = a + 1;  
    }  
}
```

$a = a + 1$  ... ist keine atomare Operation auf der CPU



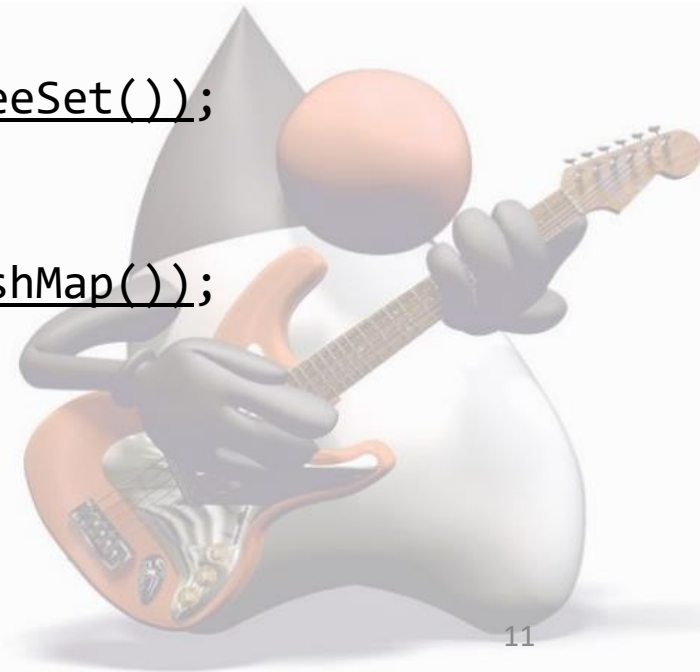
# Primitive und Multithreading

```
public class ThreadSafeCounter {  
    private AtomicInteger a = new AtomicInteger(0);  
  
    public int count() {  
        return a.incrementAndGet();  
    }  
}
```



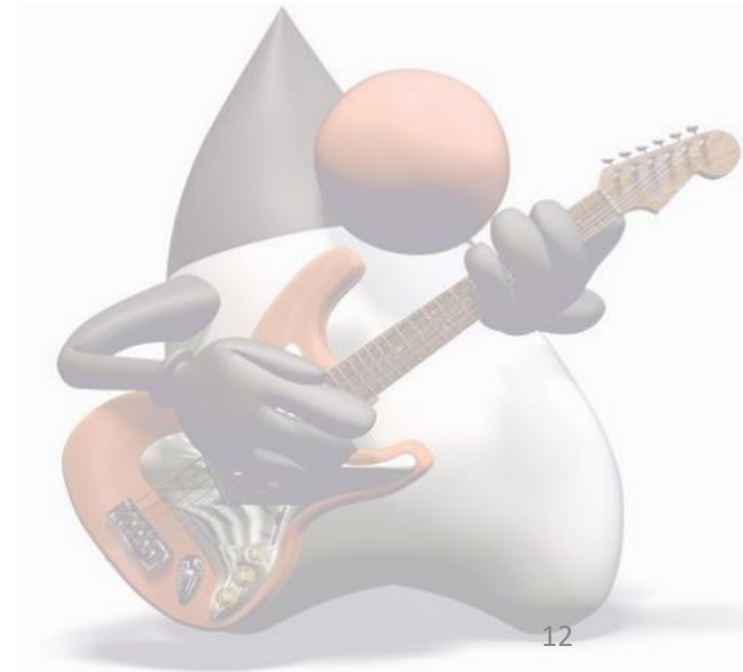
# thread-sichere Collections

```
public class TestThreadsafeCollections {  
    public static void main(String[] args) {  
        List<String> list =  
            Collections.synchronizedList(new ArrayList<>());  
  
        Set<String> set =  
            Collections.synchronizedSet(new TreeSet());  
  
        Map<Long, String> map =  
            Collections.synchronizedMap(new HashMap());  
    }  
}
```



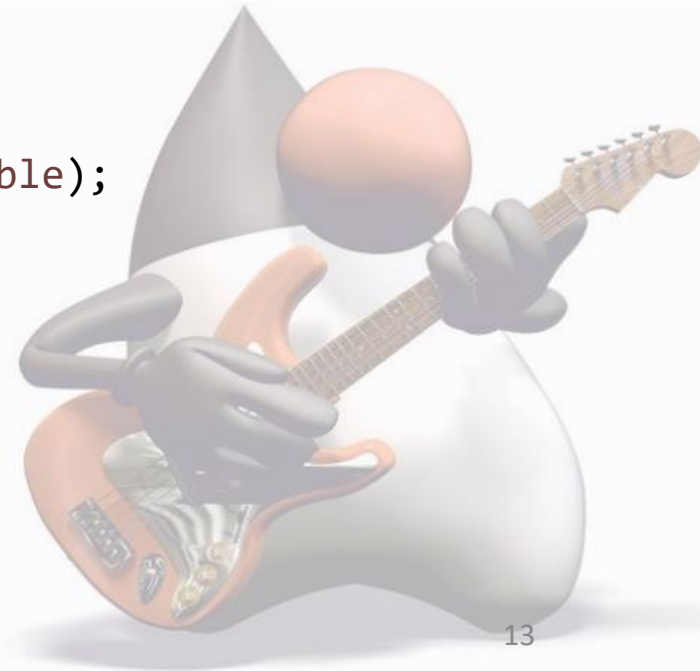
# Executor Service

- Die maximale Anzahl der Threads in einem Prozess muss limitiert sein, um die Verfügbarkeit des Servers zu garantieren
- Threads werden deshalb häufig in Pools gehalten
- es gibt vorgefertigte Threadpools



# Executor Service

```
public class ExecutorExample {  
    public static void main(String[] args) throws InterruptedException {  
        int availableProcessors = Runtime.getRuntime().availableProcessors();  
        ExecutorService service = Executors.newFixedThreadPool(availableProcessors);  
  
        Runnable runnable = () -> {System.out.print("a");};  
        Callable<Long> callable = () -> {return System.currentTimeMillis();};  
  
        for (int i=0;i<Integer.MAX_VALUE;i++) {  
            service.submit(runnable);  
            Future<Long> futureResult = service.submit(callable);  
        }  
  
        service.shutdown();  
        service.awaitTermination(Long.MAX_VALUE, TimeUnit.DAYS);  
    }  
}
```



# CompletableFuture

- CompletableFuture ist eine Erweiterung der Future Klasse
- Nachteil ist, dass ich warten muss, bis ein Ergebnis da ist
- CompletableFuture erlaubt es, einen Consumer oder eine Function anzugeben, die dann ausgeführt werden soll, wenn das Ergebnis bereitsteht.
- Ich muss nicht warten
- Es lassen sich reaktive Verarbeitungsketten damit aufbauen



# CompletableFuture

```
public class TestCompletableFuture {
    static String[] pages = { "https://www.vmware.com", "https://www.facebook.com",
                              "https://www.javatraining.at", "https://www.google.com" };

    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executor = Executors.newFixedThreadPool(10);

        for (String url : pages) {
            CompletableFuture<WebPageTestResult> cf = CompletableFuture.supplyAsync(() -> {
                long time = currentTimeMillis();
                String content = WebPageReader.readWebPage(url); // reads web page
                return new WebPageTestResult(url, content, currentTimeMillis() - time);
            }, executor);

            cf.thenApply((WebPageTestResult wbt) -> wbt.toString())
               .thenAccept((String s) -> System.out.println(s));
        }

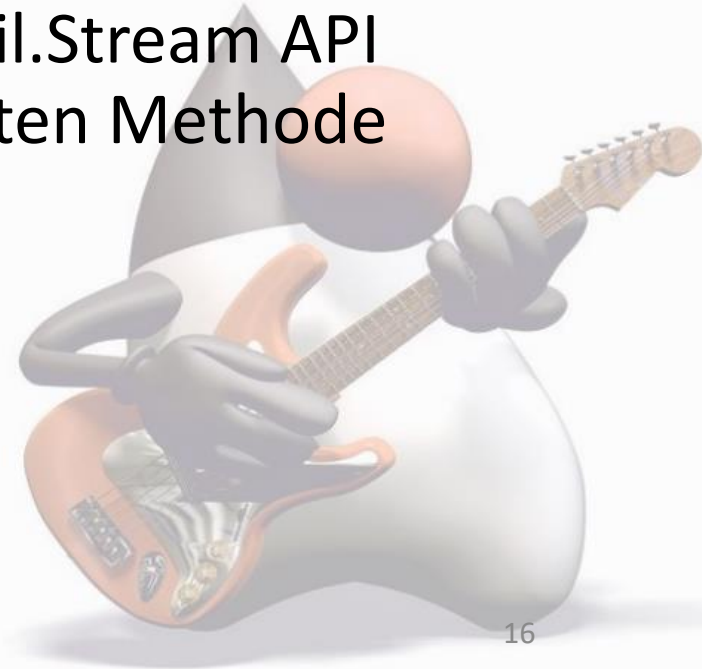
        System.out.println("Main ist zu Ende");

        executor.shutdown();
    }
}
```

```
Main ist zu Ende
WebPageTestResult [page=https://www.vmware.com, callDuration=902 ms, content size=101132]
WebPageTestResult [page=https://www.google.com, callDuration=907 ms, content size=12732]
WebPageTestResult [page=https://www.javatraining.at, callDuration=1156 ms, content size=146033]
WebPageTestResult [page=https://www.facebook.com, callDuration=1161 ms, content size=224694]
```

# Übung

- Schreiben Sie eine Klasse Averager mit einer Methode calcAverage, die einen Array von 100.000 Zufallszahlen nimmt und mit 100 parallelen Threads jeweils den Durchschnitt von 1.000 Zahlen errechnet.
- Schreiben Sie eine Methode, die mittels dem java.util.Stream API parallel den Durchschnitt der Ergebniszahlen der ersten Methode errechnet.





# Java Fundamentals

Multithreading

