PROG2

Tooling

Gradle

```
gradle-demo-project
├── .gitattributes          - configure git to gracefully handle line-endings
├── .gitignore              - configure git to ignore the transient directories build and .gradle
├── .gradle                 - gradle internal cache, log, locks, … (transient; do not commit to git)
├── app                     - module containing the Application
│   ├── build/…               - directory to store the generated results, e.g. class files (transient; do not commit to git)
│   ├── build.gradle        - build script of the application module (explained later)
│   └── src                 - any source files of application (incl. tests & resources)
│       ├── main.java.ch.zhaw.it.prog2.demo.gradle.App.java
│       └── test.java.ch.zhaw.it.prog2.demo.gradle.AppTest.java
│           └── resources
├── gradle                  - gradle support files (can be commited to git)
│   └── wrapper             - gradle wrapper → use gradle without installing it on the system
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew                 - gradle wrapper script for Unix-Shell (Linux, macOS, WSL, git-bash,…)
├── gradlew.bat             - gradle wrapper script for Windows (CMD/PowerShell)
└── settings.gradle         - define project name and included modules
```

```
plugins {
  id 'java'
  id 'application'
}

repositories {
  mavenCentral()
}

dependencies {
```

```
  implementation 'com.google.guava:guava:28.0-jre'

  testImplementation 'org.junit.jupiter:junit-jupiter-api:5.7.1'
  testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.7.1'
}

application {
  mainClassName = 'ch.zhaw.it.prog2.demo.gradle.App'
}

test {
  useJUnitPlatform()
}
```

Concurrency

Program consists of ***two or more flows of control*** which are processed in parallel (multi-core) or quasi/ pseudo parallel (single-core)

→ handle multiple user requests at the same time

Types of concurrency

True concurrency

On computers with multiple CPU cores, each core can run a flow independently in parallel.

→ 1 core = 1 flow

Interleaving concurrency

- time slicing = the cores switch between flows in rapid succession, which gives the impression of parallel flows
- **scheduler** is assigning the CPU cores to the available flows of control at each time slot using a specific strategy

Scheduling

> 💡 Which flow is getting the CPU core?

- Assigns processing time (CPU) to the available ready to run (runnable) threads

Non-preemptive (cooperative)

- Process releases core voluntarily
- first come, first served
- shortest process next

Preemptive

- Scheduler can interrupt a process (time-slice based)
- interactive processes

Real Time

- Very tight timing requirements

Terms

A program with multiple flows of control can be implemented in several ways:
- Program consisting of **multiple processes** (each with a single thread)
- Program consisting of one process with **multiple threads**
- Program consisting of **multiple processes with multiple threads**

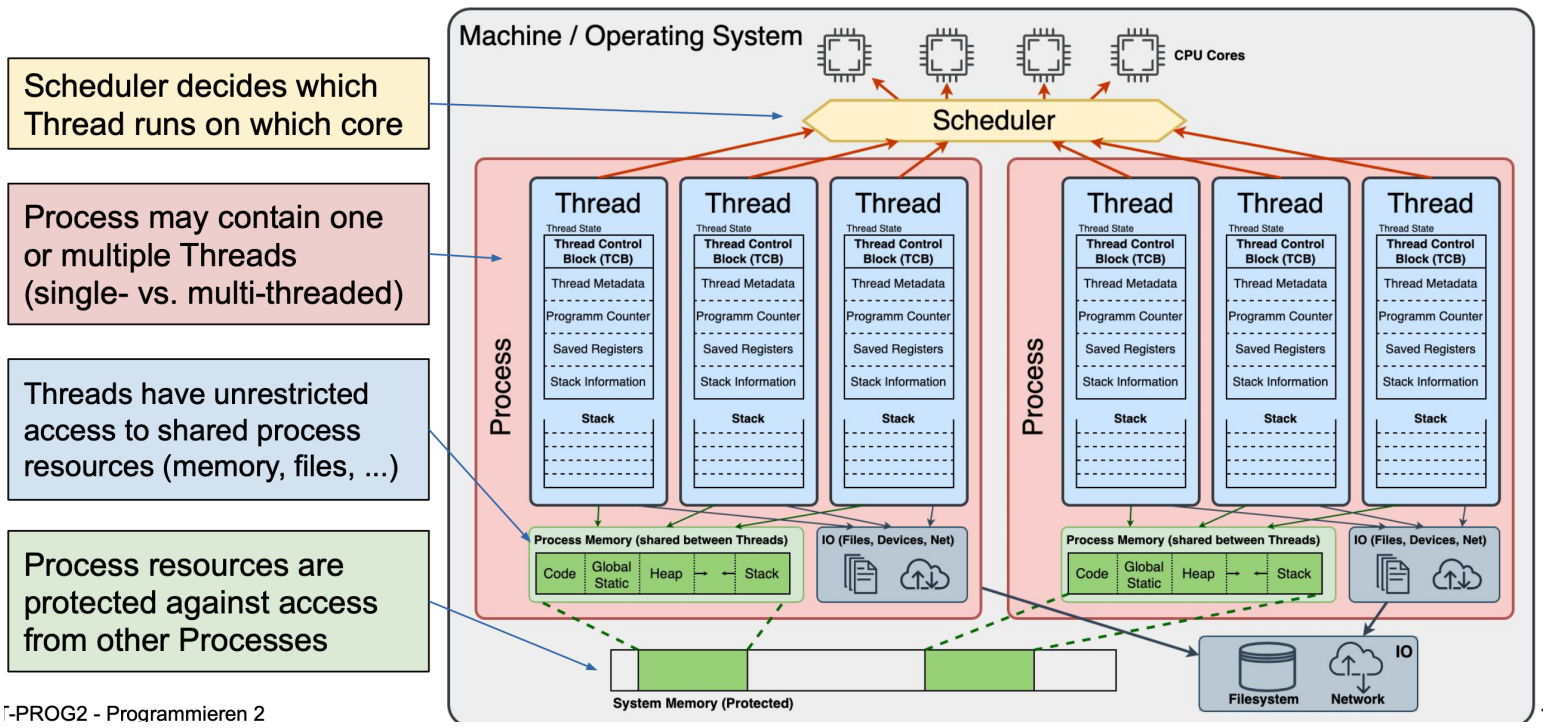
Process vs. Thread

Process

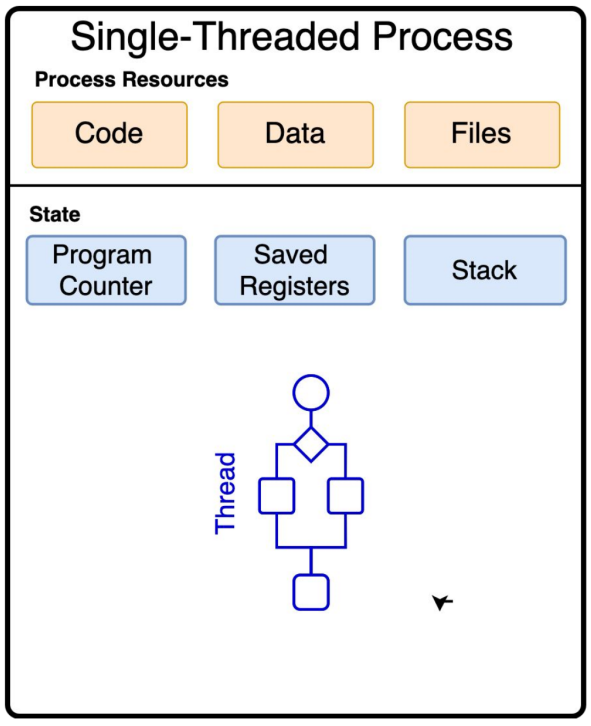
- processes run in a separate memory area (→ access to memory area of another process is not possible)
- process switching is expensive!

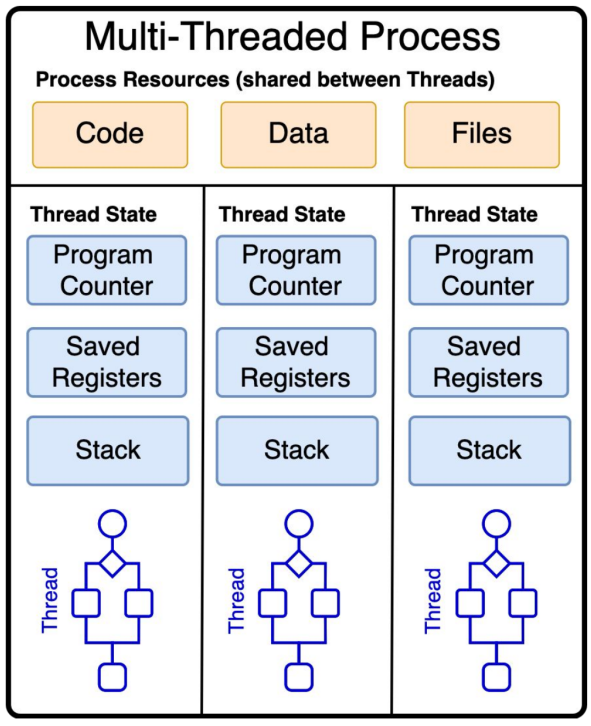
Threads

- Running in shared process memory (→ data is accessible by all threads)
- thread switching is cheap



Single vs. multi-threaded





Program Counter (PC):
Current position in program sequence

Saved Registers:
Content of CPU register to save/restore on thread switch

Stack:
Local Variables, method pointers, return values, …

Concurrency in Java

- JVM is a single process system
- JVM supports multiple threads

JVM runtime

- JVM starts one non-daemon thread, which calls the static method `main()`

> 💡 JVM runs until all non-daemon threads terminate

Daemon thread

- automatically abandoned thread once the process or JVM halts

> 💡 non-daemon threads block termination of the process/JVM until the last has ended

Create a thread

#1 Extend the `Thread` class and override the run() method

```
Thread myThread = new MyThread();
myThread.start(); // calls the run() method

class MyThread extends Thread {
  public void run() {}
}
new MyThread().start();
```

▼ Why not call the `run()` method directly?

If you call the run method directly , it won't create a new thread and it will be in same stack as main.

#2 Implementation of the `Runnable` interface in a separate or an anonymous class

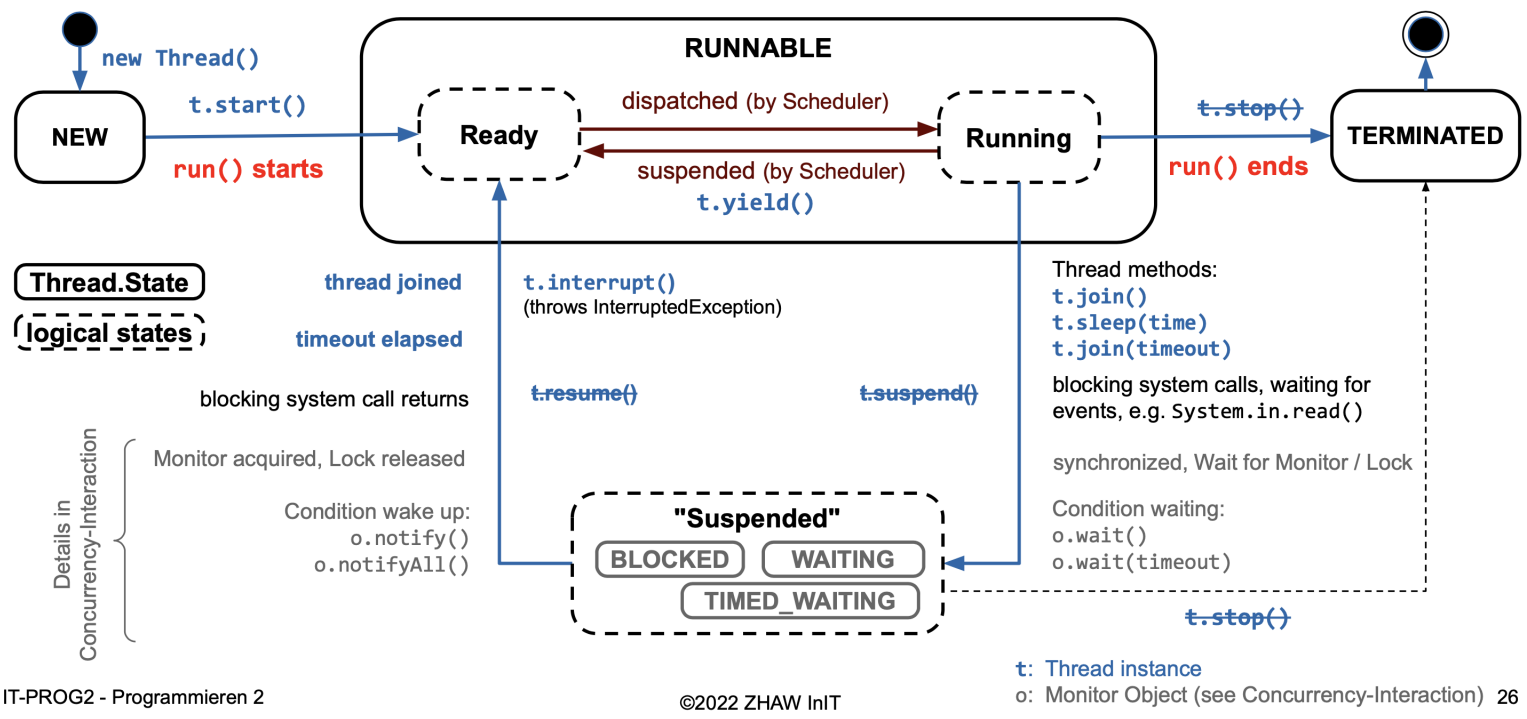
- Advantage: Java allows only single-inheritance

```
class MyRunnable implements Runnable extends ... {
  public void run() {
        // code to run in thread
  }
}
new Thread(new MyRunnable()).start();
```

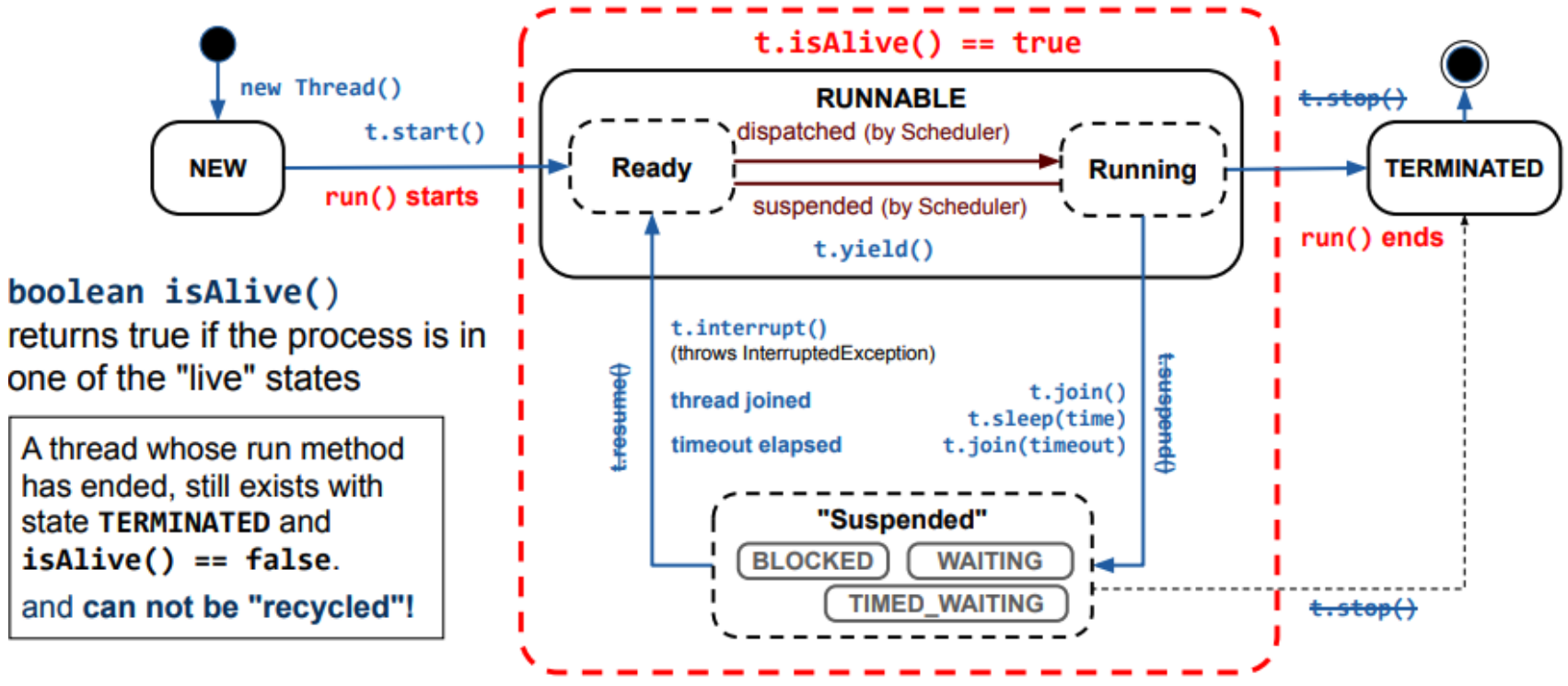
Scheduling

- implements preemptive Multithreading → Scheduler can withdraw the CPU from a thread
- A running thread can release the CPU explicitly using `yield()`
  - sends an advice, but ultimately the scheduler decides what to do

Thread lifecycle



Thread liveness



Terminate

▼ How to terminate a thread?

wait until run() ends

Step 1: let run() method terminate

- According to the state-diagram, a thread is terminating when its run method terminates
  Problem: What if the run method has to repeat its job (run-loop), until somebody from the outside decides to stop doing so?
  To call `stop()` on the `Thread` object is prohibited (as we saw before) …

Step 2: using a flag variable to end the run-loop

- Let outside process/thread set a variable, based on which the run-loop is terminated
- To ensure that the thread is evaluating the current value we have to declare the flag as data type enforcing consistency between threads (e.g. Atomic-Types).

```
AtomicBoolean doContinue = new AtomicBoolean(true);
public void run() {
    while(doContinue.get()) {   // run-loop
      // do work
    }
}
```

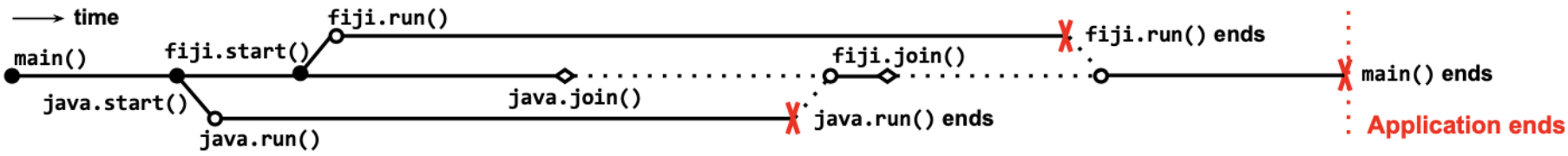
AtomicBoolean provides a consistent value for all Threads

```
class SimpleThread extends Thread {
  AtomicBoolean doContinue = new AtomicBoolean(true);

  public void run() {
    while (doContinue.get() { ... }
  }

  public void terminate() {
    doContinue.set(false);
    this.interrupt(); // wake up threads
  }
}
```

`join()` - Waiting for a thread to terminate

→ blocks the thread until the given thread stops

- if a thread calls **otherThread.join()**, itself will be blocked until **otherThread** is terminated.



`Executor` Framework

→ decouple creation and management of threads from the application logic

- Thread creation → pool of threads
- Thread management → life cycle
- Thread submission and execution → scheduling

Executor

- interface

```
public void execute(Runnable task)
```

```
class DirectExecutor implements Executor {
    // run in the current thread (not concurrent)
    public void execute(Runnable task) {
        task.run();
    }
}
class ThreadPerTaskExecutor implements Executor {
    // run each task in its own thread
    public void execute(Runnable task) {
        new Thread(task).start();
    }
}
```

ExecutorService

- adds functionality to manage the lifecycle of the tasks → provides methods to manage termination
- `submit` method allows tracking progress of tasks

ScheduledExecutorService

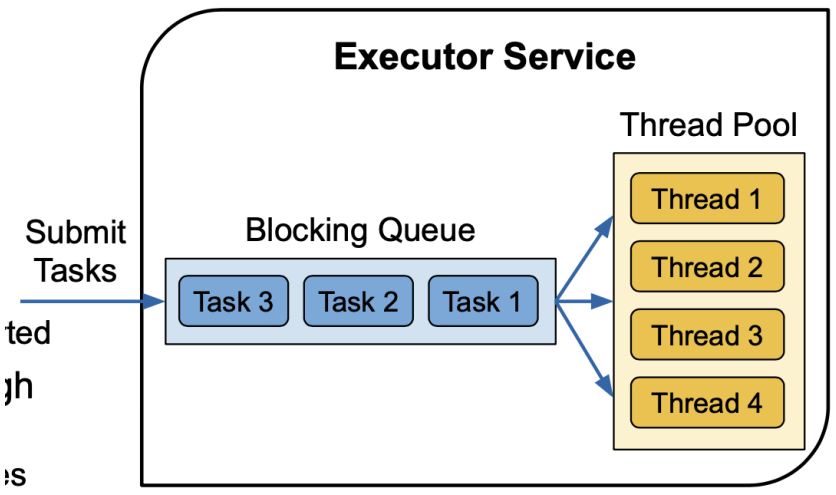
- adds functionality to schedule the execution

Termination

- keeps the threads running until it is shut down → save resources

Thread Pools

- creating threads is an expensive operation
- set of worker threads → managed independently of the tasks (Runnable & Callables)



if the queue is full, new tasks are rejected

Types of Thread Pools

- `java.util.concurrent.Executors` → provides ExecutorServices

FixedThreadPool: CPU intensiv tasks

- SingleThreadExecutor
  - "ThreadPool" with only one thread which will execute the submitted tasks sequentially
- FixedThreadPool
  - Creates and reuses a fixed number of threads
- CachedThreadPool
  - creates new threads as needed, but reuses previously created threads when they are available.
  - idle threads will be terminated and removed after 60s (shrinking the pool)
  - ideal for programs using many short-lived asynchronous tasks

maximum number of threads should match the number of cores

```
int cores = Runtime.getRuntime().availableProcessors();
```

Scheduled Executor Service

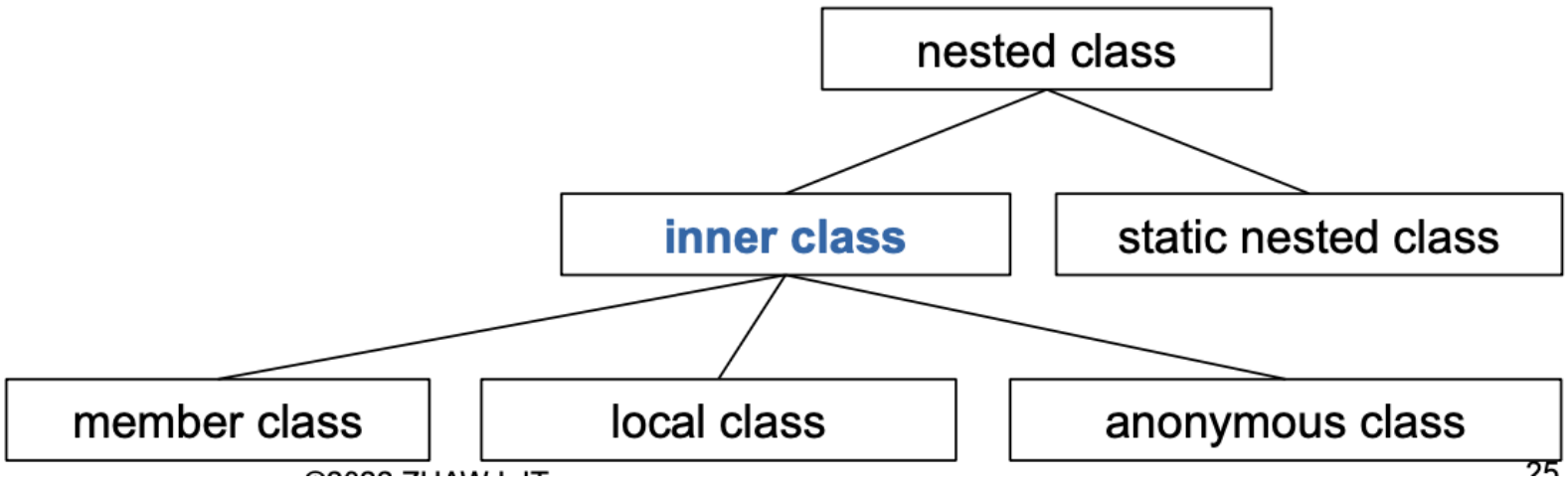
- execute tasks periodically or with a delay

- **`schedule(Runnable task, long delay, TimeUnit unit)`**
  - submits a one-shot task that will be executed after the given delay of the given time unit.
- **`scheduleAtFixedRate(Runnable task, long initialDelay, long period, TimeUnit unit)`**
  - submits a periodic action, that is executed the first time after an initial delay and subsequently with the given period
  - executes at `initialDelay, initialDelay+period, initialDelay+2*period, ...`
- **`scheduleWithFixedDelay(Runnable task, long initialDelay, long delay, TimeUnit unit)`**
  - submits a periodic action, that is executed the first time after an initial delay and subsequently with the given delay after termination of the last execution.
  - executes at `initialDelay, initialDelay+runtime+delay, initialDelay+2*(runtime+delay), ...`

Two Types of ScheduledExecutorServices are provided by the Executors class:

- ScheduledThreadPool: uses ScheduledThreadPool of given size
- SingleThreadScheduledExecutor: uses ScheduledThreadPool of size 1 (active tasks are run sequentially)

Nested / Inner classes



- **member class**
  - class definition inside another class (→ member of the outer class)
  - has access to members (variables, methods) of the outer class
- **local class**
  - class definition inside a method body (like local variables)
  - also has access to local variables of the method
- **anonymous class**
  - local class definition without a class name
  - only a single object instance is created

Callables & Futures

`Callable`

💡 return results from concurrent tasks

- can return a result, on success or throw a checked Exception

```
public interface Callable<V> {
  V call() throws Exception; // to be executed in a thread
}
```

`Future`

💡 collect the results from callables

- ExecutorService returns an object of type Future
- similar to a Promise
- placeholder representing the result of a computation → available later

```
Future<String> future = executorService.submit(callable);

Future.isDone()
Future.get() // blocks until the task is completed
```

```
public class CallableAndFutureIsDoneExample {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        System.out.println("Submitting Callable");
        Future<String> future = executorService.submit(new Callable<String>() {
            @Override
            public String call() throws Exception {
                TimeUnit.SECONDS.sleep(2);
                return "Hello from Callable";
            }
        });
        while(!future.isDone()) {
            System.out.println("Task is still not done...");
            TimeUnit.MILLISECONDS.sleep(400);
        }
        System.out.println("Task completed! \nRetrieving the result");
        String result = future.get();
        System.out.println("Result: " + result);
        executorService.shutdown();
    }
}
```

submitting Callable as anonymous inner class

Wait for result!

Not blocking!

```
Output:
Submitting Callable
Task is still not done...
Task is still not done...
Task is still not done...
Task completed!
Retrieving the result
Result: Hello from Callable
```

Waiting until all tasks have completed

1. Use List<Future>
2. `ExecutorService.invokeAll()`

```
List<Callable<String>> taskList = Arrays.asList(task1, task2, task3);
long startTime = System.currentTimeMillis();
List<Future<String>> futureList = executorService.invokeAll(taskList);
```

fastest task

```
long startTime = System.currentTimeMillis();
String result = executorService.invokeAny(taskList);
```

Wait for completion of **fastest Callable**, which does not throw an Exception

Cooperation

Synchronization is required for:

- accessing shared resources
- cooperation or coordination of threads

Threads & shared resources

- Threads use the same memory address space

→ Threads can access the same shared objects

- Java operations are generally not atomic!

- On parallel execution of multiple threads the flow can **run interleaving**
  → Results depend on scheduling (not predictable, different in each run)!
  → **Race Conditions → occasional Errors → not Thread safe**
- **Solutions**
  - Avoid shared resources (often not possible)
  - For simple cases use **Atomic Types**
  - Allow **only one thread** at a time **access to the shared resource**
    → **Mutual Exclusion**

Atomic types

- single values: **AtomicBoolean**, **AtomicInteger**, **AtomicLong** and **AtomicReference**
- array types: **AtomicIntegerArray**, **AtomicLongArray**, **AtomicReferenceArray**

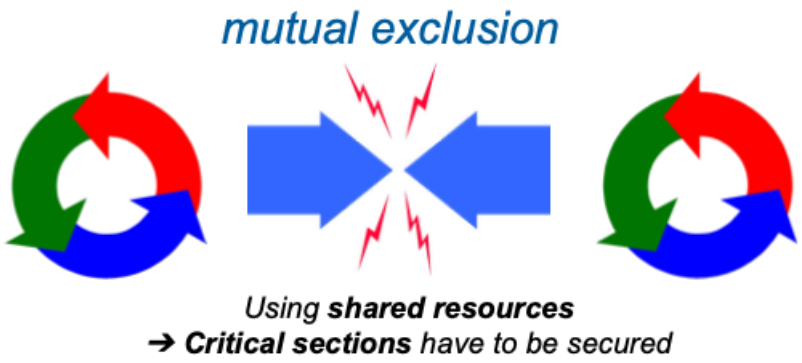
- They provide a set of atomic function combinations:
  - **t addAndGet(t delta)** vs. **t getAndAdd(t delta) (returns updated/old val)**
  - **t incrementAndGet()** vs. **t getAndIncrement()**
  - **t decrementAndGet()** vs. **t getAndDecrement()**
  - **boolean compareAndSet(t expectedValue, t newValue)**
    → set `newValue` if `expectedValue == currentValue`
  - setter & getter methods, ...

Types of synchronization

1. Mutual exclusion
2. Condition synchronization

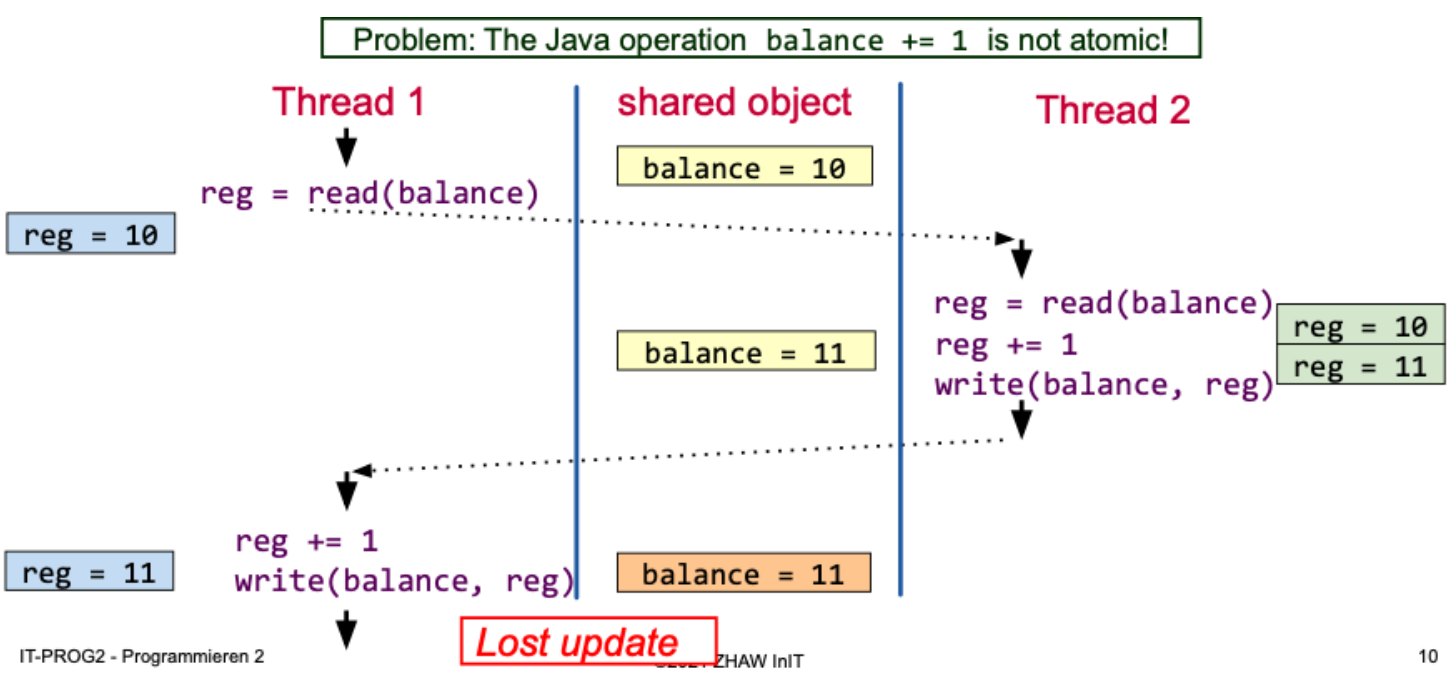
Mutual exclusion

💡 Allow only one thread at a time access to the shared resource



*Using **shared resources***
*➔ **Critical sections** have to be secured*

▼ Example



Critical sections

Code blocks that access shared objects must never be executed at the same time by multiple threads

```
this.balance += amount; // critical section (not atomic)
```

Busy waiting

```
class Account {
     private int balance = 0;
     private volatile boolean locked = false;
     public void transferAmount(int amount) {
          while (locked) {;} //busy waiting
          locked = true;   //enter critical section
          this.balance += amount;
          locked = false;  //leave critical section
     }
}
```

critical section

→ inefficient! ⇒ prefer **synchronized statement**

synchronized statement

- marks a critical section within the code

→ executed as mutually excluded

Synchronized Method

```
class Account {
  private int balance = 0;
  ...
  public synchronized void transferAmount(int amount) {
    this.balance += amount;
    System.out.println(amount);
  }
}
```

critical section

Synchronized Block

```
class Account {
  private int balance = 0;
  ...
  public void transferAmount(int amount) {
    synchronized (this) {
      this.balance += amount;
    }
    System.out.println(amount);
  }
```

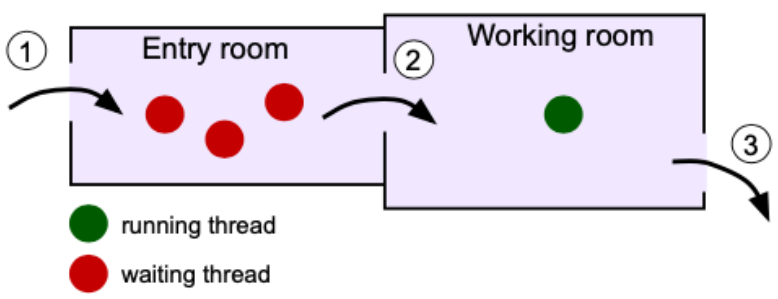
Monitor-Object (see later)

critical section

- based on the `Monitor` concept

Monitor object

- lock for exclusive access right
- maximally one thread can own the monitor



1. Acquire monitor: Enter entry room
   → `synchronized` is called
2. Monitor successfully acquired
   Got the lock, enter working room
   → enter `synchronized` block
3. Release monitor: Leave working room
   → leave `synchronized` block

- Start of **`synchronized`** block: **acquire monitor**
  - Already occupied → wait in entry room
  - otherwise → enter working room
- Within **`synchronized`** block: Mutual exclusion guaranteed
  - Thread "owns" the monitor lock
  - No other thread can enter the working room
- End of **`synchronized`** block: **release monitor**
  - If other threads are in the entry room, one is chosen to get the lock

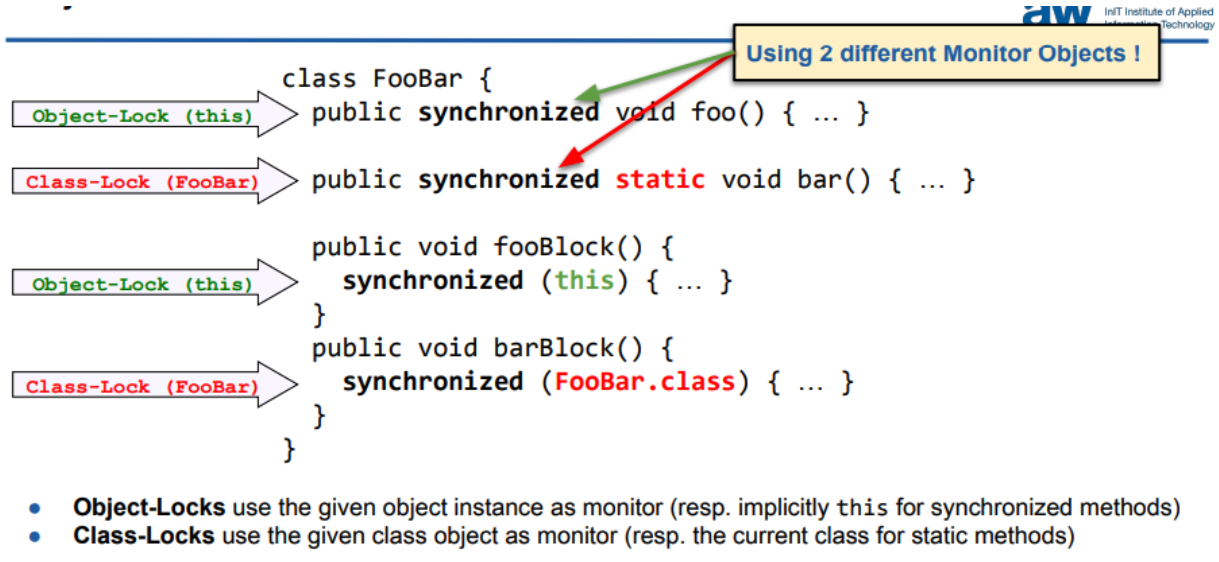
recursive

```
class MyClass {
    public synchronized void methodA() {
        ...
    }
    public synchronized void methodB(){
        ...
        this.methodA();
    }
}
```

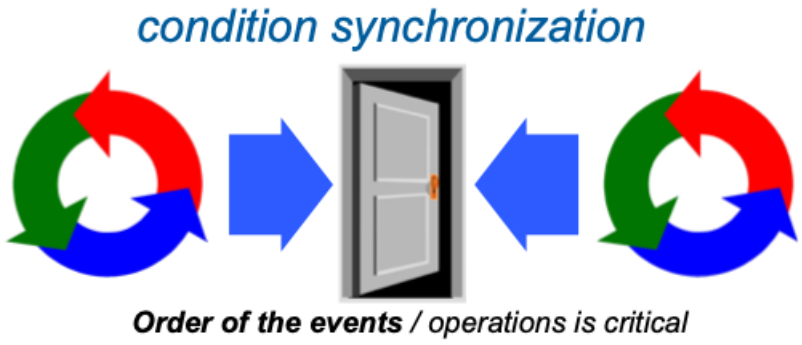
Thread does not block because both methods use the same monitor (`this`) and the Thread already owns it.

→ possible deadlock if the methods use different `Monitor` objects

Object-Lock vs. Class-Lock



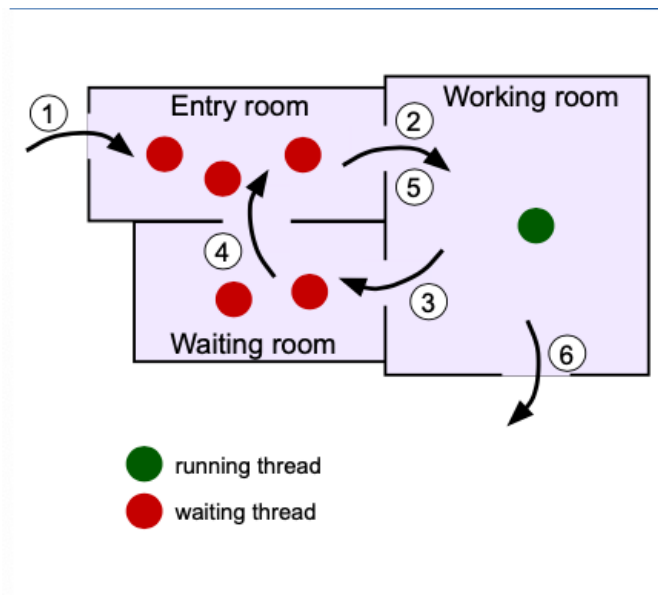
Condition synchronization



Extended Monitor concept

→ waiting room

- Temporary **release the "lock" and wait**, to allow other threads to enter the synchronized-blocks, and modify the state. Continue on event notification.
- **Passive-Wait**: The sleeping Thread is not using CPU-time



1. Try to acquire monitor lock: Enter entry room
   → `synchronized` is called
2. Monitor successfully acquired:
   Got the lock, enter working room
   → enter `synchronized` block
3. Release monitor and wait for event:
   Go to waiting room, release the lock
   → memorize position in synchronized block
4. Get event and try reacquiring monitor:
   Re-enter entry room
5. Monitor successfully reacquired:
   Got the lock, enter working room
   → continue at position in synchronized block
6. Release monitor lock: Leave working room
   → leave `synchronized` block

Each object provides the following methods:

```
wait()
notify()
notifyAll()
```

💡 calling those functions is only allowed if owning the monitor → current thread is within the synchronized block

wait

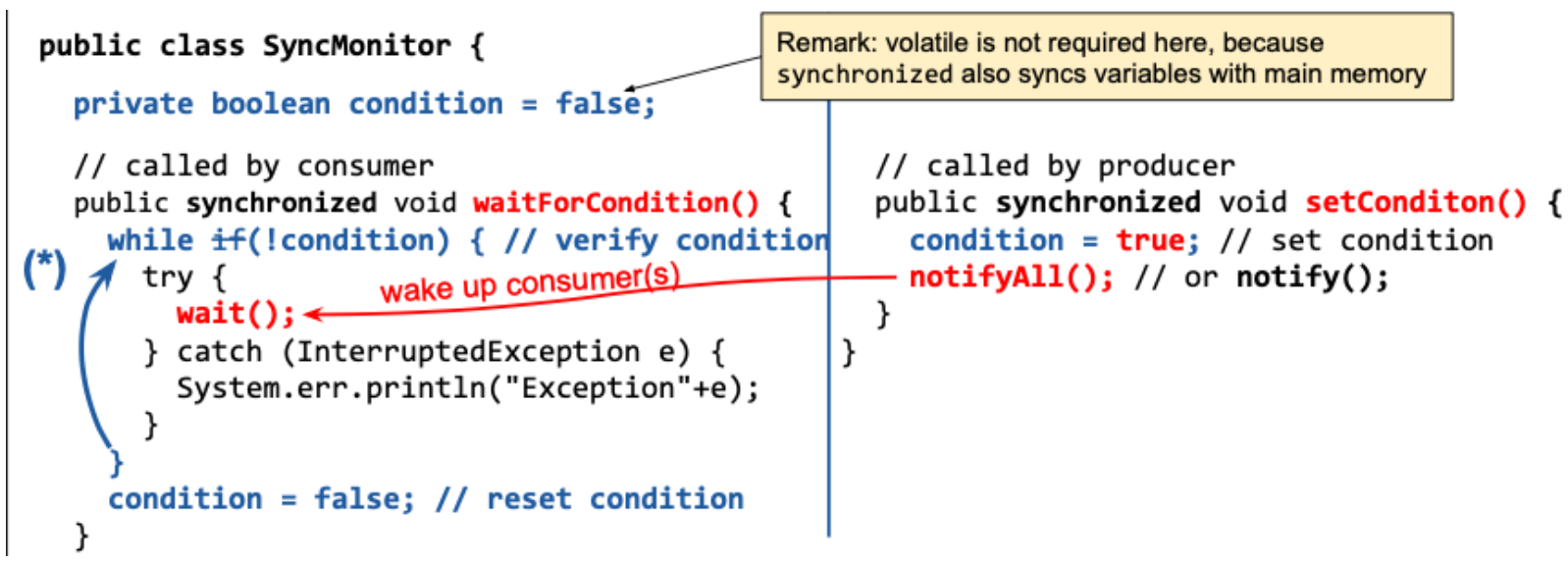
- calling thread is suspended → waiting room
- monitor lock is released → next thread from the entry room can enter the working room

notify

- awakes one random thread in the waiting room → entry room

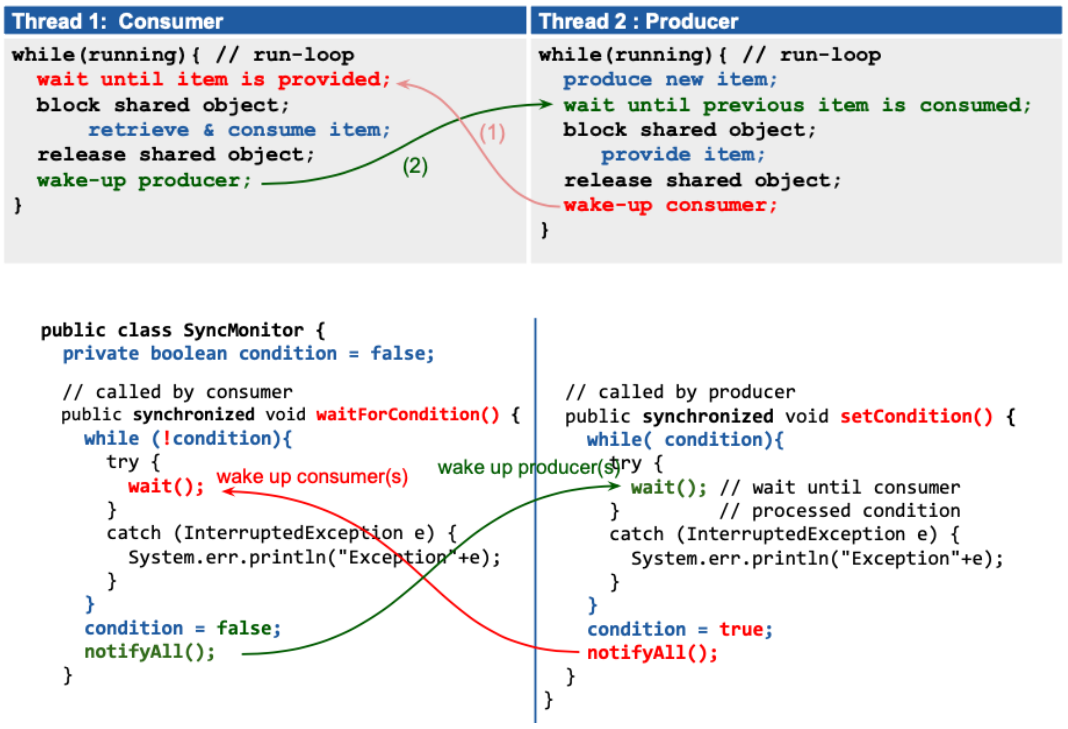
Sync Monitor object

Implementation

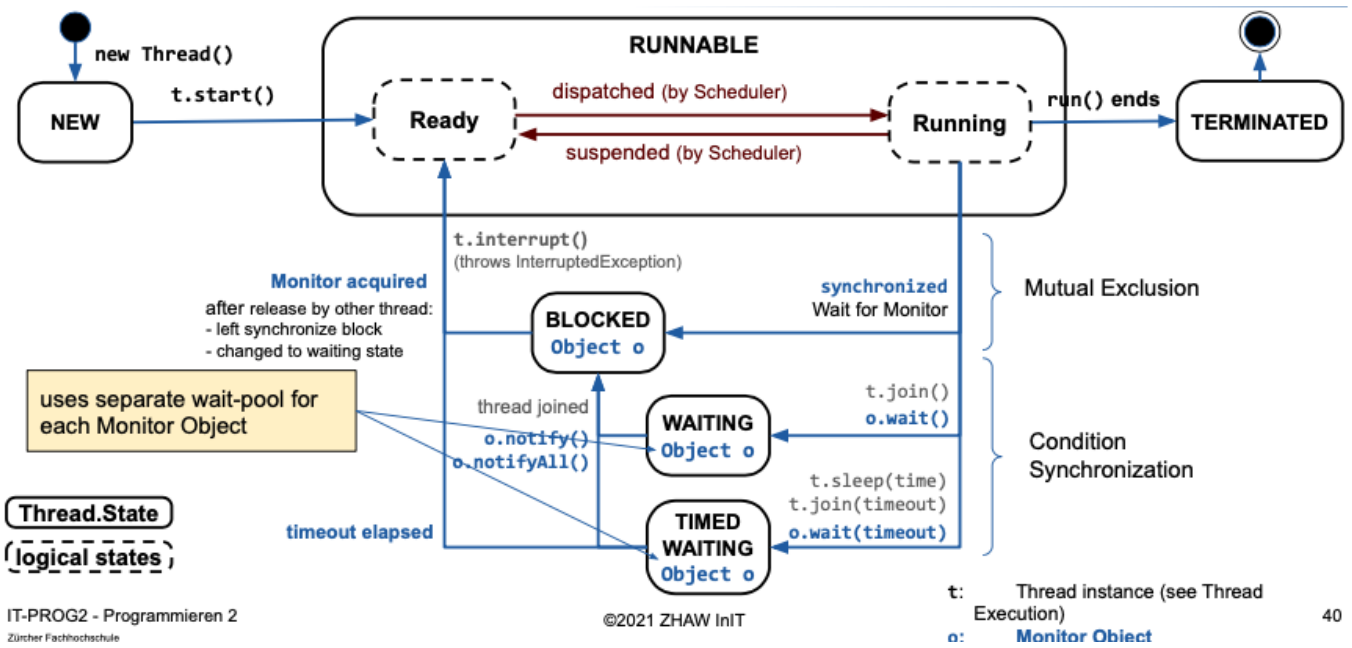


Two way condition synchronization

→ Producer should wait until consumer has processed the request (feedback)



Thread lifecycle

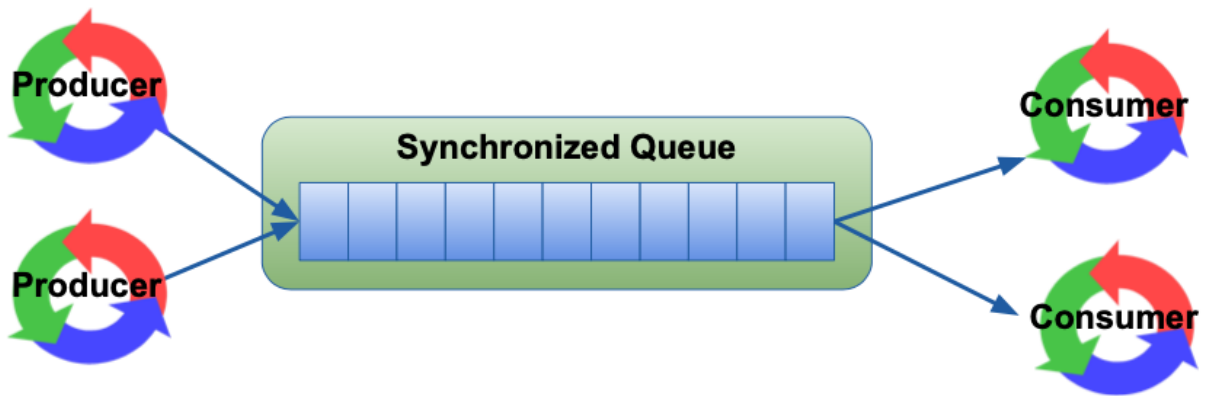


Advanced synchronization mechanisms

Synchronized queues

→ Producer-Consumer problem with queue

- Producer and consumer threads share a queue of data objects



Thread 1: Producer

```
while(running){ // run-loop
  produce item;
  wait if queue is full;
  block shared object (queue);
      provide item to queue;
  release shared object (queue);
  wake-up consumer;
}
```

Thread 2 : Consumer

```
while(running){ // run-loop

  wait if queue is empty;
  block shared object (queue);
      fetch item from queue;
  release shared object (queue);
  wake-up producer;
  consume item;
}
```

```
public class SyncQueue<E> {
  private LinkedList<E> queueList;
  private int capacity;

  public SyncQueue(int capacity) {
    this.capacity = capacity;
    queueList = new LinkedList<E>();
  }

  // called by producer
  public synchronized void add(E item)
  throws InterruptedException
  { // condition: queue not full
    while (queueList.size() >= capacity) {
      wait();  // block if queue is full
    }
    queueList.addLast(item); // append to list
    notifyAll(); // wake-up consumers
  }
```

Using Java Generics allows Consumer Producer scenarios using a generic queue

Java Generics → see BlueJ book Ch. 4 & 5

```
// called by consumer
public synchronized E remove ()
throws InterruptedException
{  // condition: queue not empty
   while (queueList.size() == 0) {
     wait(); // block if queue is empty
   }
   E item = queueList.removeFirst();
   notifyAll(); // wake-up producers
   return item;
}
```

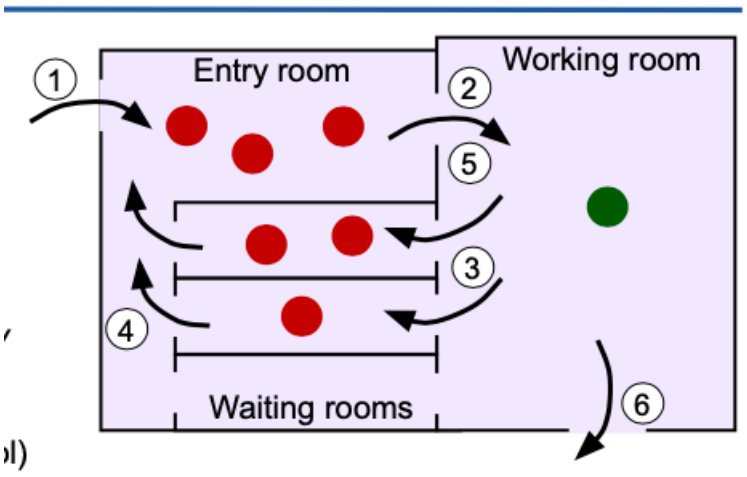
Lock & Conditions

Problem: Monitor-Objects have only one Wait-Set (waiting room)

→ Goal: Mechanism using a separate Wait-Set per condition

- Lock object for mutual exclusion
- Condition object user for condition synchronization

→ Each Condition object belongs to exactly one Lock object



1.	try to acquire lock:	`mutex.lock()`
2.	lock acquired:	`lock()` successful
3.	wait for cond X:	`condX.await()`
4.	thread in WR called	`condX.signal()`
5.	lock reacquired:	continue after await()
6.	release lock:	`mutex.unlock()`

```java
public class ConditionSync {
  private Lock mutex = new ReentrantLock();
  private Condition change = mutex.newCondition();
  private boolean condition = false;

  public void waitForCondition() throws InterruptedException {
      mutex.lock();
      try {
          while(!condition) { // verify condition
              change.await();
          }
          condition = false;  // reset condition
      } finally {
          mutex.unlock();
      }
  }
  public void setCondition() {
      mutex.lock();
      try {
          condition = true;   // set condition
          change.signalAll(); // or change.signal();
      } finally {
          mutex.unlock();
      }
  }
```

try-finally required to ensure release of mutex lock

```java
public class MonitorSync {


private boolean condition = false;

  public synchronized void waitForCondition()
                              throws InterruptedException {

    while(!condition) { // verify condition
        wait();
    }
    condition = false;  // reset condition
  }



  public synchronized void setCondition() {
    condition = true;   // set condition
    notifyAll();        // or notify();
  }
}
```

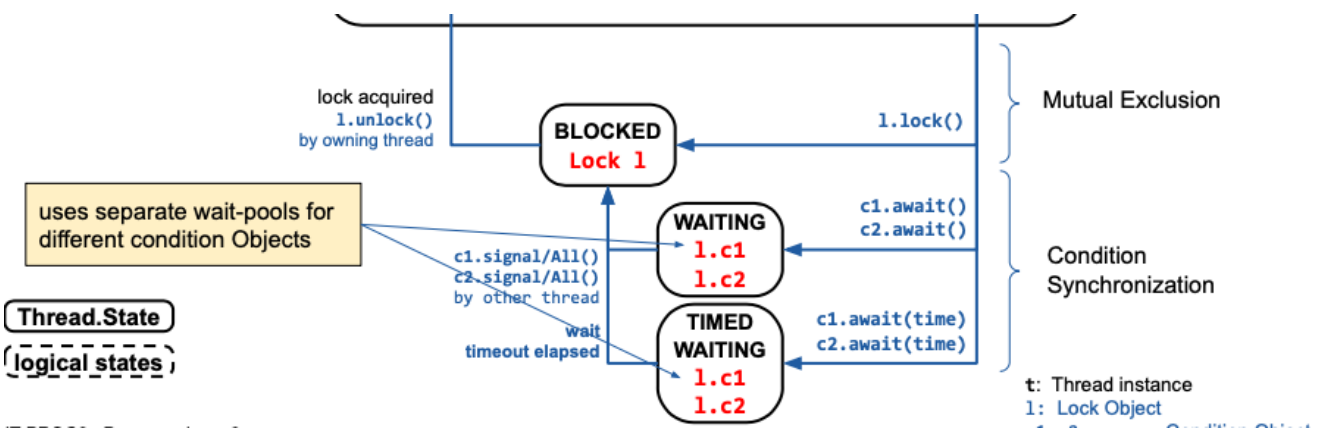
Code required for locking / mutual exclusion
Code required for waiting / condition synchronization

```java
public class ConditionalSyncQueue<E> {
  private Lock mutex = new ReentrantLock();
  private Condition notEmpty = mutex.newCondition();
  private Condition notFull  = mutex.newCondition();
  private LinkedList<E> queue = new LinkedList<E>();
  private int capacity = 5;

  public void add (E item)
  throws  InterruptedException {
    mutex.lock();         // enter critical section
    try { // condition 1: queue not full
      while(queue.size() >= capacity) {
        notFull.await();
      }
      queue.addLast(item);
      notEmpty.signal();
    } finally {
      mutex.unlock();     // exit critical section
    }
  }
```

Similar to Monitor with wait() & notify():
- **Lock** object replaces synchronized block or method
- **Condition** objects provide Wait-Sets (waiting rooms) for selective waiting (await) and wakeup (signal).

```java
  public E remove() throws InterruptedException {
    E item = null;
    mutex.lock();         // enter critical section
    try { // condition 2: queue not empty
      while (queue.empty()) {
        notEmpty.await();
      }
      item = queue.removeFirst();
      notFull.signal();
    } finally {           // why finally?
      mutex.unlock();     // exit critical section
    }
    return item;
  }
}
```

Thread lifecycle



Threads with differing access modes

→ improve performance ⇒ locking reduces parallelism (intentionally)

→ keep the amount of code in critical sections as small as possible

- One option is to differentiate between *access modes* to shared resources (i.e. shared objects):
  - **Writer** writes new content to the resource (can also read)
    → *needs exclusive access*
  - **Reader** only reads content from the resource
    → *multiple concurrent readers are ok*
- This is particularly useful when the resource is primarily read.

ReadWriteLock provides two different but associated Lock types

- write lock → exclusive access (read & write) to shared resource
- read lock → shared access (read only) to shared resource

```
public interface ReadWriteLock {
   Lock readlock();
   Lock writelock();
}
```

		Already used locks	
		r-lock	*w-lock*
Requested locks	*r-lock*	access	block
	w-lock	block	block

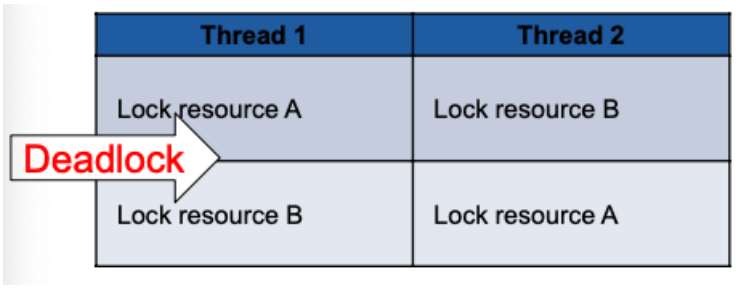
```
class Account {
  private final ReadWriteLock rwLock = new ReentrantReadWriteLock();
  private final Lock readLock  = rwLock.readLock();  // shared lock
  private final Lock writeLock = rwLock.writeLock(); // exclusive lock
  private int balance = 0;
  ...
```

```
// write & read access -> exclusive lock
public void transferAmount(int amount) {
  writeLock.lock();
  try {
    this.balance += amount;
  } finally {
    writeLock.unlock();
  }
}
```

```
// read only access -> shared lock
public int getBalance() {
  readLock.lock();
  try {
    return this.balance;
  } finally {
    readLock.unlock();
  }
}
```

Deadlocks

- can occur if two threads are waiting a resource, which is locked by the other thread



Requirements

A deadlock can only occur if all the 4 following conditions are met:

- **Mutual Exclusion**
  - Each resource is available only once
- **Hold and Wait condition**
  - Processes which are already blocking resources claim additional resources
- **No Preemption**
  - A blocked resource cannot be taken away by the OS
- **Cyclic waiting conditions**
  - A chain of processes exists which are waiting for a resource, which is blocked by a successor in the chain

Avoiding deadlocks

- **Prevent Mutual Exclusion**
  - Required to avoid shared access problems (e.g. lost updates)
- **Prevent Hold And Wait condition**
  - Atomic pre-claim all required resources
    - not supported natively in Java
    - possible with C on Windows: WaitForMultipleObjects
  - May be possible by changing abstraction of condition
- **Prevent non-preemption**
  - Not supported in Java
- **Prevent Cyclic Waiting conditions**
  - e.g. global ordering of the resources and claiming resources in same order (ascending or descending)

Thread 1	
Lock resource A	
Lock resource B	

prevent cyclic waiting conditions

```
public synchronized void transferAmount(int amount) throws AccountOverdrawnException {
  if (this.balance < -amount) { throw new AccountOverdrawnException (this.id, this.balance, amount); }
  this.balance += amount;
}

public static void transfer(Account from, Account to, int amount) throws AccountOverdrawnException {
  boolean isLower = from.getId() < to.getId();
  Account lowerAccount  =  isLower ? from : to;
  Account higherAccount = !isLower ? from : to;
  synchronized( lowerAccount ) {
    synchronized( higherAccount ) {
      from.transferAmount(-amount);
      to.transferAmount(amount);
    }
  }
```

Dining philosophers

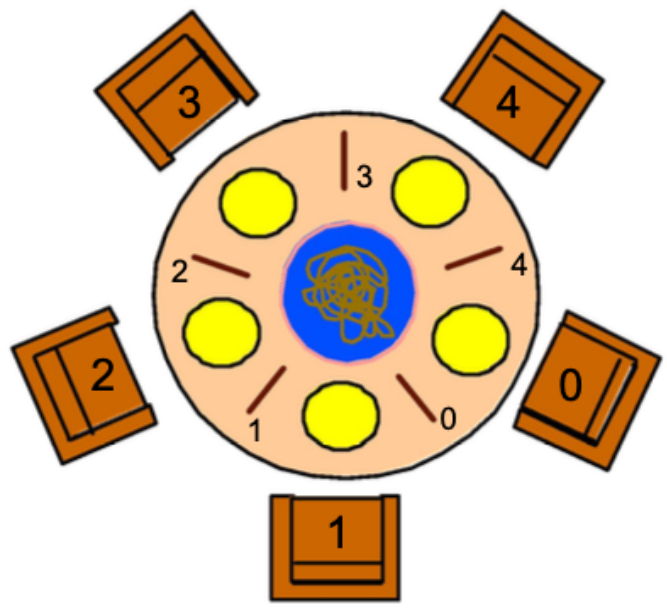
Classical synchronization example:

- 5 philosophers
- 1 bowl of spaghetti
- Philosopher:
  - either thinking
  - or eating
- Eating requires two forks
  - `acquireFork(int id)`
- Only 5 forks available

→ Starving while a neighbor is eating

Dijkstra 1965

(see also Tanenbaum, "Modern Operating Systems")




```
class ForkManager {
  public ForkManager(int numForks) {
    mutex = new ReentrantLock();
    this.numForks = numForks;
    forks = new Fork[numForks];
    for (int i=0; i < numForks; i++)
      forks[i]=new Fork(mutex);
  }

  // take fork no. i
  public void acquireFork(int i) {
    mutex.lock();
    try {
      while (forks[i].state != ForkState.FREE) {
        forks[i].cond.await();
      }
      forks[i].state = ForkState.OCCUPIED;
    } finally {
      mutex.unlock();
    }
  }
```

```
class Fork {
  public ForkState state = ForkState.FREE;
  public Condition cond;

  public Fork(Lock mutex) {
      cond = mutex.newCondition();
  }
}
```

```
// release fork no. i
public void releaseFork(int i) {
  mutex.lock();
  try {
    forks[i].state = ForkState.FREE;
    forks[i].cond.signal();
  } finally {
    mutex.unlock();
  }
}
```

Problem

- Prevent mutual exclusion
  - Tricky to share forks with your neighbor at the same time
- Prevent hold-and-wait
  - Pick up both forks simultaneously
- Prevent non-preemption
  - On the way to your mouth, the fork is rudely taken away and given to someone else
- Prevent cyclic waiting condition
  - Several feasible strategies exist

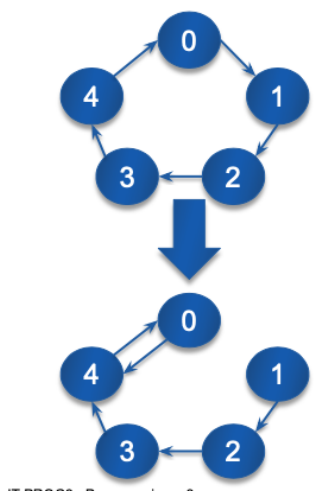
Solution 1

- Prevent Hold-and-Wait

Idea: Acquire & release forks always in pairs (change abstraction)

- Acquiring fork pair
  - **`wait()`** or **`cond.await()`** until both forks are available (`state==FREE`)
  - Use `while` (not just `if`)
  - Take both forks at once (mark them unavailable)
- Releasing fork pair:
  - Mark both forks available at own place
  - Wake up waiting threads → `cond.signalAll()` or `cond.signal()`

Solution 2




```
private static class Philosopher {
  private void takeForks() {
    state = PhiloState.hungry;
    forkMgr.acquireFork(id);
    forkMgr.acquireFork(forkMgr.right(id));
  }
```

```
private static class Philosopher {
  public void takeForks() {
    state = PhiloState.hungry;
    if (id == 0) {       // avoid cyclic wait
      forkMgr.acquireFork(id);
      forkMgr.acquireFork(forkMgr.right(id));
    } else {
      forkMgr.acquireFork(forkMgr.right(id));
      forkMgr.acquireFork(id);
    }
  }
```

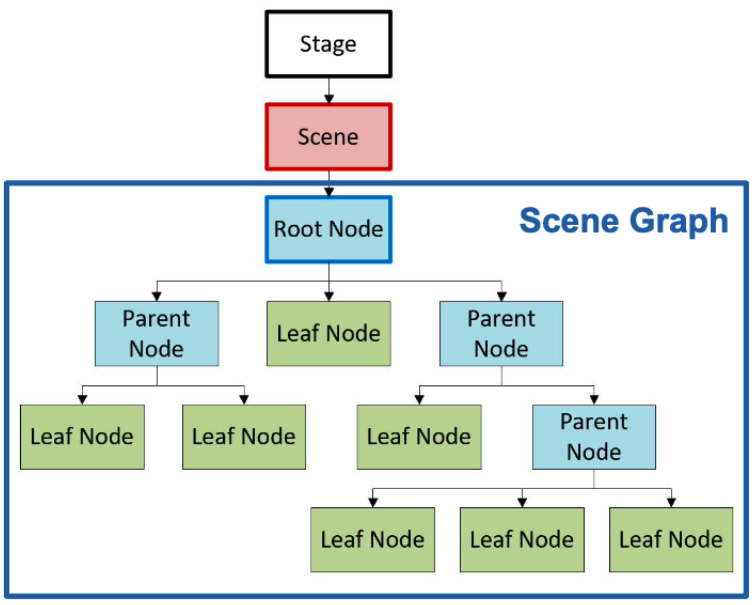
GUI

GUI → hierarchical tree of components / nodes

Scene graph

- Structure of the components in a scene
- cycle free

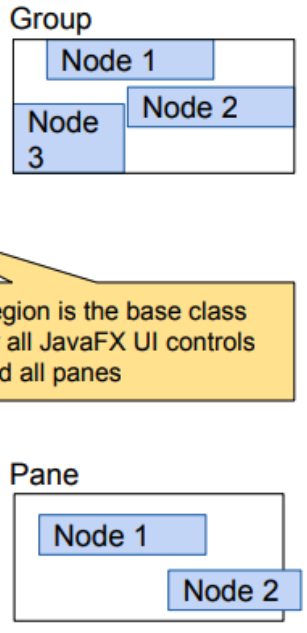
- Node: Top class for all elements in a scene graph
- Parent: Handles all hierarchical scene graph operations like adding/removing child nodes, picking, bounds calculations, etc.
- Scene based on containers and controls
- The scene graph is a tree of all used elements → all are nodes



Container nodes

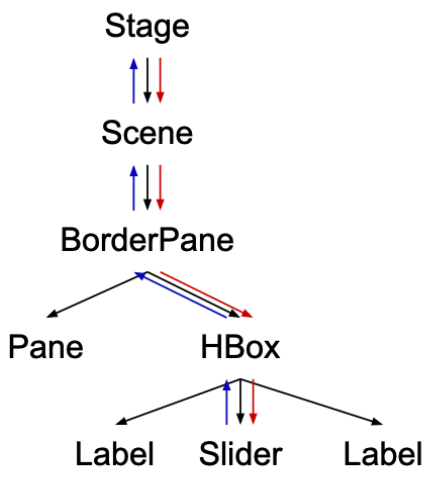
- Combine several node objects into a single node
- settings on the container apply to the children

- **Group**
  - children are keeping their size
  - just large enough to contain all the enclosing objects
- ***Region (inherited by Controls and Panes)***
  - size independent from children size
  - defines minimum, maximum and desired size
  - with regions size changes, the children are newly arranged and their size adjusted if necessary
  - different subclasses implement different layouts
- **Pane**
  - manual size and manual layout of children
  - clipping possible



Event handling

- Information about the mouse-click on the slider is encoded into a **MouseEvent** object
- The **MouseEvent** is passed through to the target object (Slider). On each step the **EventFilter** is applied. → Event capturing phase ↓
- The event object is passed back to the top. In each step the **EventHandler** is called (Example: changing the slider value). → Event bubbling phase ↑
- **EventFilter** and **EventHandler** can be arbitrarily defined by the programmer.



lifecycle

Inherits from the abstract class `javafx.application.Application`

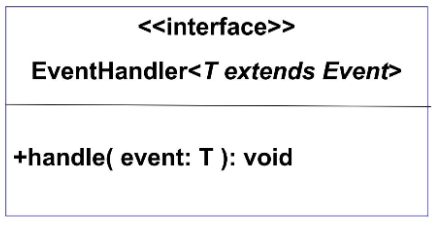
- `launch(Class<?extends Application> appClass, String… args)` or `launch(String… args)` `// uses current class`
  Static method, generally called from `main()`
  1. Constructs an instance of the given `javafx.application.Application` class
  2. Creates an initial window (stage) and starts a thread to handle events
  3. `init()` method is called
     - empty method, can be overridden to initialize environment (e.g. external connections), no GUI elements (stage) available
  4. `start(javafx.stage.Stage)` method is called
     - **Mandatory** abstract entry point method to set up JavaFX UI
  5. Waits for the application to finish, which happens when either of the following occur:
     - the application calls **`Platform.exit()`** (if `System.exit()` is called, `stop()` is not executed)
     - the last window has been closed
  6. `stop()` method is called
     - Empty method, can be overridden to cleanup environment

handler

→ don’t create duplicate handler instances

Objects implementing the `EventHandler` interface are used to process events.

- There is only one generic interface for ALL kinds of EventHandlers




```
class SayClickHandler implements EventHandler<MouseEvent> {
    @Override
    public void handle(MouseEvent event) {
        System.out.println("Click");
    }
}
```

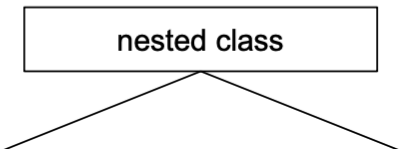
problems

Problems:

- Writing separate EventHandler classes for each event is expensive.
- External classes do not have access to UI elements, except the calling element (using `event.getSource()`)

Solution:

Use inner classes!




```
myButton.setOnMouseClicked(new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent event) {
        System.out.println("Click");
    }
});
```

Anonymous class instance

provides the same functionality as

```
myButton.setOnMouseClicked(new SayClickHandler());
```

Anonymous Object/Instance (not assigned to a variable)

other possibilities

Options to declare an event handler:

1. Define a named (inner) class *that implements EventHandler.* As we did just before.
2. Write it as an *anonymous class.*

Since Java 8 two additional ways are available

3. Write it as a *lambda expression*.
4. Implement it as a *method* and use a *method reference*.

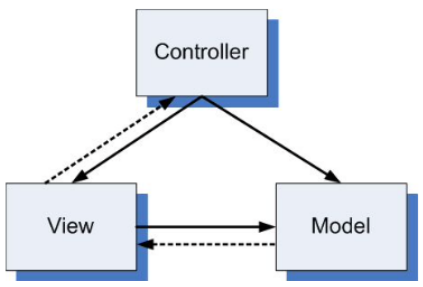
see functional programming in PROG2

MVC

💡 Every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class → single responsibility

- **Model**
  - Manages program logic
  - Offers methods to
    - Query the state of the model
    - Control the model
- **User Interface**
  - Figures out what the user wants
  - Controls the Model
  - Displays the state of the Model

- **View**
  - Displays current state (of Model & User Interface)
  - Takes user input and forwards it to controller
- **Controller**
  - Processes user input (events)
  - Controls model and view accordingly



Implications

- UI (View & Controller) has direct access to Model
- Model never calls UI directly
- View (Controller) registers as listener with the Model
  - UI gets notified on changes and can update itself

Goals & Benefits

- Design goal
  - Maximum decoupling of model and UI (View & Controlle
  - No logic in UI, no UI-aspects in model
- Benefits
  - Independent development & testing of model and UI
  - Possibility for many views/controllers for one model
  - Changes to UI or model much easier

FXML

- JavaFX enables the use of the MVC pattern through FXML

View

```
<?xml version="1.0"

<?import javafx.scene.*?>
...
<AnchorPane id="pane" prefHeight="150.0" prefWidth="500.0" xmlns="http://javafx.com/javafx/16"
   xmlns:fx="http://javafx.com/fxml/1" fx:controller="ch.zhaw.prog2.example.fxml.MainWindowController">
   <children>
      <VBox alignment="CENTER" layoutY="121.0" spacing="20.0" AnchorPane.bottomAnchor="0.0"
         AnchorPane.leftAnchor="0.0" AnchorPane.rightAnchor="0.0" AnchorPane.topAnchor="0.0">
         <children>
            <Label fx:id="viewText" layoutX="172.0" layoutY="121.0" text="Some Text" />
            <HBox alignment="CENTER" spacing="5.0">
               <children>
                  <TextField fx:id="inputText" alignment="TOP_LEFT" layoutX="60.0" layoutY="175.0" />
                  <Button onAction="#handleShow" text="Show Text" />
                  <Button onAction="#handleClear" text="Clear Text" />
               </children>
            </HBox>
         </children>
      </VBox>
   </children>
</AnchorPane>
```

FXML-file contains a scene graph of the view

reference to the controller class

fx:id of the control
(allows to access it from the controller)

event handler method to call in the controller class

→ no interaction with the Model

Controller

```
public class MainWindowController {

    // controls in views
    @FXML private Label viewText;       // reference to viewText Label
    @FXML private TextField inputText; // reference to intputText TextField

    // event handler methods
    @FXML
    private void handleShow() {        // called by showText Button
        viewText.setText(inputText.getText());
    }

    @FXML
    private void handleClear() {       // called by clearText Button
        inputText.clear();
    }
}
```

Controllers for FXML need a **DefaultConstructor** to get initialized (next slide), but we do not need the Constructor to initialize the private fields

`@FXML` annotates **fields** to be linked to controls of the view

`@FXML` annotates which action **methods** can be called from the controls event handler

action **methods** and **fields** can be made **`private`** (FXML uses reflection and can access private members)

Initialisation

```
private Pane initMainWindow() throws IOException {
    // create loader to read the FXML-description for scene graph
    FXMLLoader loader = new FXMLLoader(getClass().getResource("MainWindow.fxml"));

    // read the file, build the Scene Graph and initialize the controller
    Pane rootNode = loader.load();

    // access and further configure the controller created by the FXMLLoader (if required)
    MainWindowController controller = loader.getController();

    return rootNode;
}

// the application start method does not change
public void start(Stage primaryStage) throws Exceptio
    // Initialize the main window scene graph and con
    Pane rootNode = initMainWindow();
    // setup scene and stage
    Scene scene = new Scene(rootNode);
    primaryStage.setScene(scene);
    primaryStage.setTitle("JavaFX Example");
    primaryStage.show();
}
```

`FXMLLoader.load()`

1. Builds the scene graph from the FXML-description
2. Creates an instance of the declared controller class (e.g. **`fx:controller="ch.zhaw.prog2.example.fxml.MainWindowController"`**)
   You can get this instance using `loader.getController()`)
3. Initializes the `@FXML` annotated class properties to the controls with matching **`fx:id`** (e.g. **`Label viewText ⇒ fx:id="viewText"`**)
4. Creates EventHandlers for the actions calling the `@FXML` annotated methods (e.g. `<Button onAction="#handleShow"/> ⇒ private handleShow() { … }`
5. Returns the root node of the scene graph

Observer pattern

- one-to-many relationship between observable and observers (= Listener)
- when the observable changes, all observers are notified
- loosely coupled

`javafx.beans.Observable` and matching `ChangeListener` (observers).

– For singular values[1] this is `javafx.beans.value.ObservableValue`:

```
public interface ObservableValue<T> extends Observable {
    void addListener(ChangeListener<? super T> listener);
    void removeListener(ChangeListener<? super T> listener);
    T getValue();
}
```

A value of generic type `T` is **wrapped** by the **`ObservableValue`** class. Specific subclasses for `String`, `Object` and primitive types `int`, `boolean`,... provide an additional get method e.g. `int get(); boolean get(); ...`

Generic type that allows ChangeListener of type T and the supertypes of T

– and the observer of type `javafx.beans.value.ChangeListener`:

```
public interface ChangeListener<T> {
    void changed(ObservableValue<? extends T> observable, T oldValue, T newValue);
}
```

Registered **`ChangeListener`** objects get notified by calling the **`changed`** method when the **`ObservableValue`** has changed

Generic type that allows ObservableValue of type T and all types extending T

1) Similar Observable and Listener classes exist for collections:
- `javafx.collections.ObservableList` ⇒ `javafx.collections.ListChangeListener`
- `javafx.collections.ObservableMap` ⇒ `javafx.collections.MapChangeListener`

Bindings

When the GUI should know about changes

– from Model (independent of GUI)
⇒ use the observer pattern (see Lab, exercise 1)

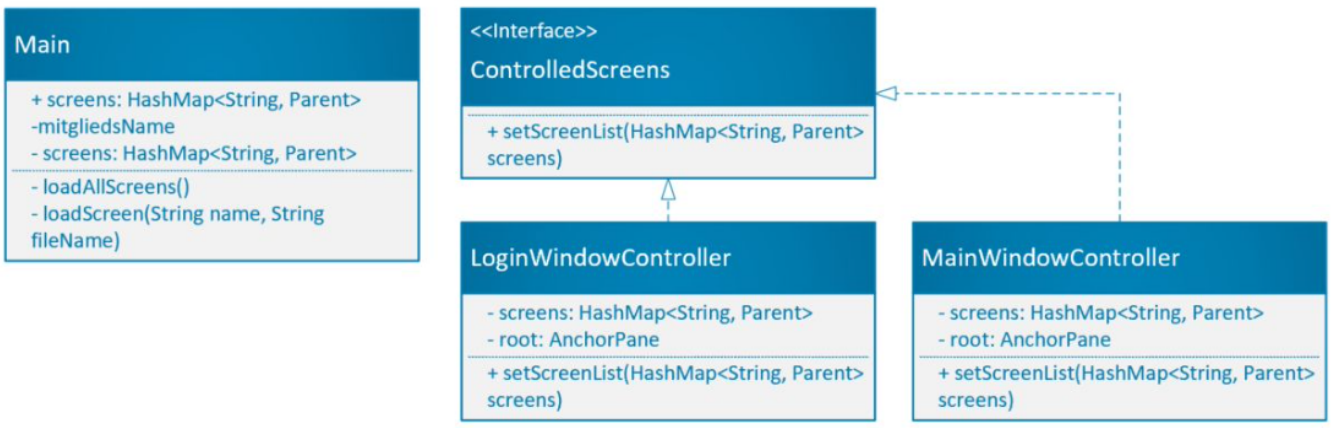
– within GUI classes
⇒ use properties (next slides)

It is possible to create direct bindings between two properties:

`targetProperty.bind(sourceProperty)`

⇒ every time **`sourceProperty`** changes, **`targetProperty`** is updated

Multiple views / scenes




```
...
private HashMap<String, Parent> screens = new HashMap<>();

private void loadAllScreens() {
      loadScreen("Main", "MainWindow.fxml");
      loadScreen("Login", "LoginWindow.fxml");
}

private void loadScreen(String name, String fileName) {
      try {
            FXMLLoader loader = new FXMLLoader(getClass().getResource(fileName));
            Parent loadScreen = (Parent) loader.load(); // load returns the root node
            ControlledScreens controlledScreen = (ControlledScreens) loader.getController();
            controlledScreen.setScreenList(screens); // write reference to controller
            screens.put(name, loadScreen);
      } catch (Exception e) { // do something     }
}
```

Open separate window

```
private void openNewWindow() {
     try {
          FXMLLoader loader = new FXMLLoader(getClass().getResource("LoginWindow.fxml"));
          Pane rootPane = loader.load();
          // connect the root-Node with the scene
          Scene scene = new Scene(rootPane);
          // create a new stage for a new window
          Stage stageOfNewWindow = new Stage();
          stageOfNewWindow.setScene(scene);
          stageOfNewWindow.show();
     } catch(Exception e) { // do something }
     ...
```

Mock-Testing

Psychology

> Testing is the process of executing a program with the intent of finding errors.
>
> The Art of Software Testing, Myers et al. 3.Edition. John Wiley & Sons.

Principles to test software economically

1. Specification of Input and Output

Test cases must consist of a description of the input data to the program and a precise description of the correct output of the program for that set of input data.

2. Separate Creation and Testing

A software developer should not test his or her own programs.

3. Completeness of tests

Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected.

4. Testing is an investment
5. Error clusters

Errors tend to come in clusters

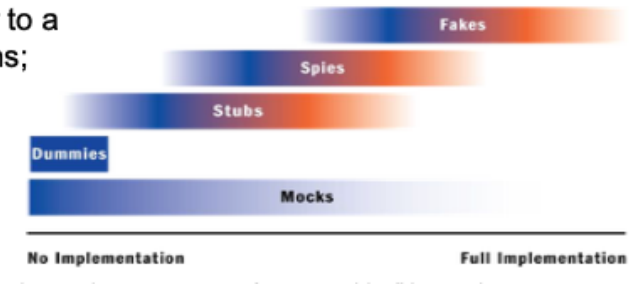
Test doubles

> 💡 A “test double” is any object or component that is installed in place of the real component for the express purpose of running a test.

→ Each class is tested independently of other classes = test isolation

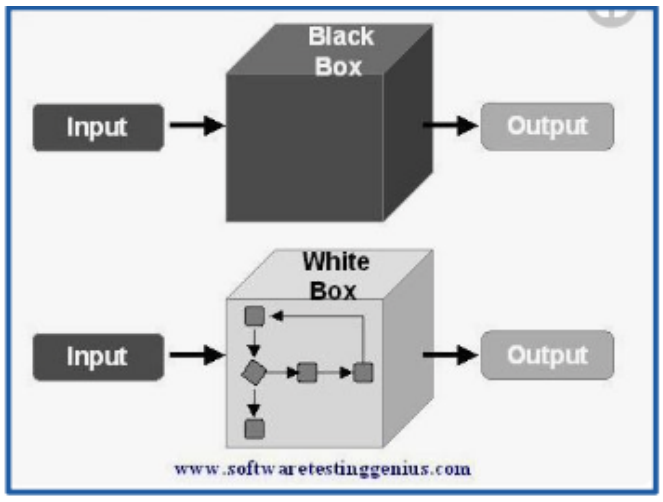
Implementations

- **Dummy** represents the **simplest** and **most primitive type of test double** objects are **passed around** but never actually used; usually they are just used to **fill parameter lists** (e.g., It is to be tested whether a constructor generates an exception if one of three parameters is null- in this case, one can use a dummy for each of the other parameters)
- **Stubs** are **minimal implementations** of interfaces or base classes; methods returning **void** will typically contain **no implementation** at all, while methods returning values will typically **return hard-coded values**; data fields and reasonable logic should only be used where necessary
- **Spies** are **similar to a stub**, but a spy will also **record** which members were **invoked** so that unit tests can verify that members were invoked as expected
- **Fakes** contain **more complex** implementations but take shortcuts; usually deals with **interactions between different methods**; can be similar to a production implementation, however with several simplifications; **not valid in production** (e.g., an in-memory database)
- **Mock** objects can be pre-programmed with **expectations** about the methods that will be invoked, and about the **interaction** with the object; Depending on the configuration, a mock can behave like a dummy, a stub, a spy or a fake



Stubbing vs Mocking

- Black-box testing
  - **No assumptions** about the inner workings
  - Only the **interface** of the class is known (public fields & methods)
  - This is **state** testing
  - Usually a case for **stubbing**
- White-box testing
  - **Some/full information** about inner workings
  - Thus, we want to test the inner workings
  - This is **behavior** testing
  - Usually a case for **mocking**



Stubs

- implementation of an interface or base class with minimum necessary functionality

```
public class OrderTest {
    @Test
    public void orderIsFilledWhenHasInventory() {
        Warehouse warehouse = new UnlimitedWarehouseStub();
        Order order = new Order("Talisker", 50);
        order.fill(warehouse);
        assertTrue(order.isFilled());
    }
}
public class UnlimitedWarehouseStub extends Warehouse {
    public boolean hasInventory(product, quantity) {
        return true;
    }
    public void remove() {  }
}
```

Stub seems to be not flexible enough for the hasInventory method
→ let's see if there is a more flexible option…

Stub works fine for the remove method

Mocks

- behavior testing → checks if certain methods were called with the correct input parameters
- whitebox testing
- verify correctness of behavior

1. **Create** a mock object for the interface or class we would like to simulate
2. **Specify** the *expected behavior* of the mock
3. **Use** the mock in standard unit testing, i.e. JUnit, as if it is a normal object
4. **Verify behavior** (and possibly state)

setup phase
exercise phase
verify phase

```
public void testFillingRemovesInventoryIfInStock() {
  // configuration
  Order order = new Order(TALISKER, 50);
  Mock warehouseMock = new Mock(Warehouse.class);
  // expectations
  warehouseMock.expects(once()).method("hasInventory").with(eq(TALISKER),eq(50)).will(returnValue(true));
  warehouseMock.expects(once()).method("remove").with(eq(TALISKER), eq(50)).after("hasInventory");
  //exercise
  order.fill((Warehouse)warehouseMock.proxy());
  //verify
  warehouseMock.verify();       //verify expected behavior
  assertTrue(order.isFilled()); //verify state
}
```

Mockito

- All methods and interactions are recorded by the mock object

```java
@Test
void test() {
    Half mockedHalf = mock(Half.class);
    Heart heart = new Heart(mockedHalf, new Half("R"));
    when(mockedHalf.isAtrioventricularValveOpen()).thenReturn(false);
    when(mockedHalf.istSemilunarValveOpen()).thenReturn(true);


    heart.executeSystole();


    verify(mockedHalf).contractVentricle();
    verify(mockedHalf, times (1)).contractAtrium();
}
```

setup phase

exercise phase

verify phase

Interaction verification

- Use `times(int wantedNumberOfInvocations)` in verification to verify how many times a method has been used/called
- Other interaction methods:
  - `never()`
  - `atLeastOnce()`
  - `atLeast(int)`
  - `atMost(int)`

Timeouts

- Interaction verification can also be combined:

```java
// Verifies that add() is called twice in 100 milliseconds
verify(mockedList, timeout(100).times(2)).add("once");
```

- Verify that no interaction happened with the mock

```java
// Verifies that no interactions happens with the mockedList
verifyZeroInteractions(mockedList);
```

Order

```java
List singleMock = mock(List.class);

// using a single mock
singleMock.add("first");  // First interaction with the mock object
singleMock.add("second"); // Second interaction with the mock object

// Wrap the mock object to an InOrder verifier to test if the mock is called in the right order
InOrder inOrder = inOrder(singleMock);

// Verify the order
inOrder.verify(singleMock).add("second"); // Verify that add was called with "second" -> Error
inOrder.verify(singleMock).add("first");  // Verify that add was called with "first"  -> Error
```

Stubbing

- `when(mock.stubbedMethod(values)).thenReturn(Object);`
- `doReturn(Object).when(mock).stubbedMethod(values);`

```java
// Create the mock for a person object which has the method getLongName()
Person mock = mock(Person.class);

// Tell the mock object to return “Hans Muster” when getLongName() is called
when(mock.getLongName()).thenReturn("Hans Muster");

// Assert
assertEquals("Hans Muster", mock.getLongName());
assertEquals("Hans", mock.getFirstName()); // will fail, getFirstName() is not stubbed
```

Callback

- sophisticated stubs with return values based on input parameters

- when(mock.*method*(*values*)).thenAnswer(*Answer<T>*);
- doAnswer(*Answer<T>*).when(mock).*method*(*values*);

```
// Create callback function using anonymous inner class
when(mock.hash(anyString())).thenAnswer(new Answer<String>() { //create anonymous inner class
   String answer(InvocationOnMock invocation) {
      return invocation.getArgument(0).toString().hashCode();
   }
});
// same callback function using a lambda expression (covered later in this course)
when(mock.hash()).thenAnswer((invocation) -> invocation.getArgument(0).toString().hashCode());
```

Argument matchers

- Any-matchers: anyInt(), anyString(), anyCollection(), any(Class xx),...
- String-matchers: startsWith(), endsWith(), contains(), matches(),...
- Object-matchers: isNull(), IsNotNull(), isA(),...
- Compare-matchers: eq(), same(),...
- Custom-matchers: argThat(), intThat(), floatThat(),...

```
List mockedList = mock(List.class);
//stubbing using anyInt() argument matcher
when(mockedList.get(anyInt())).thenReturn("defaultValue"); // return defaultValue for any argument
```

Spy

- deal with legacy code or only parts of the interface needs to be modified

```
List list = new LinkedList();
// create a spy on the real object instance
List spy = spy(list);

// stub the size() method
when(spy.size()).thenReturn(100);

// add() is not stubbed. So it will use the real method
spy.add("one"); spy.add("two");

assertEquals("one", spy.get(0));
assertEquals(100, spy.size());
```

IO

Filesystem

- organised hierarchically → root node

File

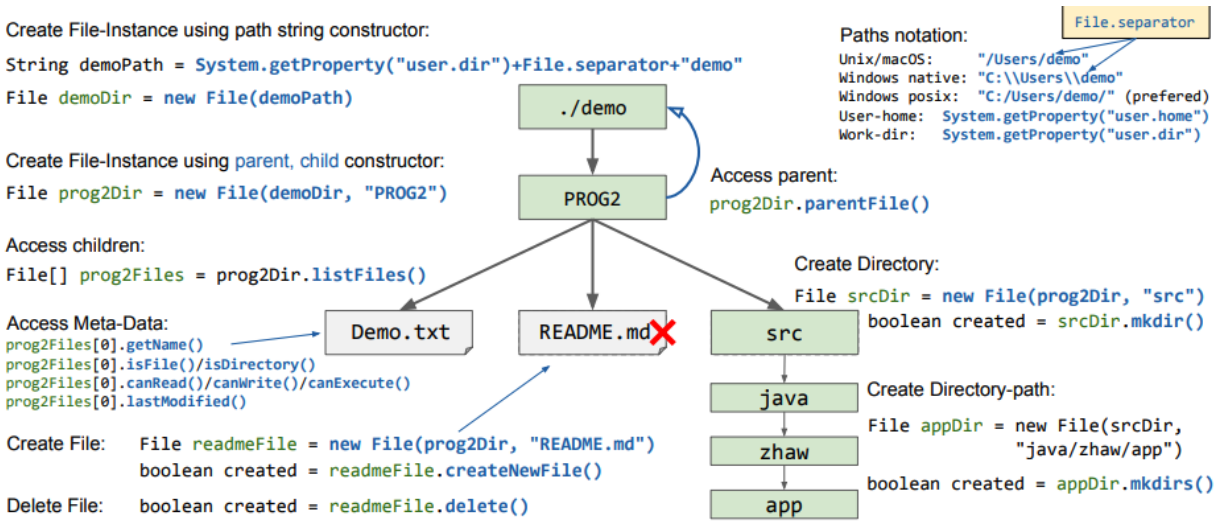
- represents a set of related data

consists of:

1. Content (collection of bytes)
2. Metadata (management information)

`java.io.File`

Everything is a `File`



`java.nio.Files`

- The **`java.nio.file`** package supports a different approach which is providing these features and fits better for many use cases.
- It uses a different approach using two main classes*):
  - **`java.nio.file.Path`**: represents the path to a location in the FS-tree (file-path) (may be absolute, or relative)
  - **`java.nio.file.Files`**: providing static methods that operate on files and directories

```
class FilesDemo {

  public static void main(String[] args) throws IOException {
    String pathName = (args.length == 1)? args[0] : "./demo";

    Path path = Path.of(pathName);
    System.out.println("File Name: " + path);
    if (Files.isDirectory(path)) {
      // print directory content
      for (Path filePath : Files.newDirectoryStream(path)) {
        printAttributes(filePath);
      }
    } else {
      printAttributes(path);
    }
  }
```

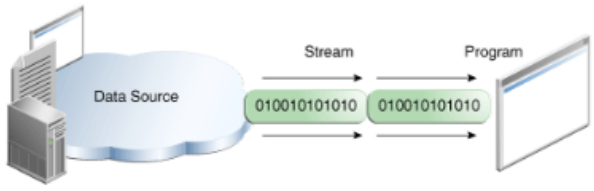
```
static void printAttributes(Path path) throws IOException {
    System.out.printf(
      " - %-10s %c%c%c%c%c modified %tF %<tT size %d bytes%n",
      path.getFileName(),
      Files.isDirectory(path)  ? 'd':'-',
      Files.isReadable(path)   ? 'r':'-',
      Files.isWritable(path)   ? 'w':'-',
      Files.isExecutable(path) ? 'x':'-',
      Files.isHidden(path)     ? 'h':'-',
      Files.getLastModifiedTime(path).toInstant()
           .atZone(ZoneId.systemDefault())
           .toLocalDateTime(),
      Files.size(path)
    );
}
```

I/O-Streams

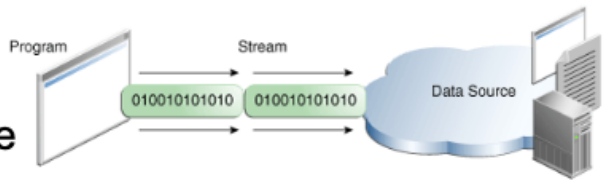
- Java defines two basic types of streams
  - Byte Streams
    - Byte oriented (8-Bit)
    - Generic, i.e. binary data
  - Character Streams
    - Character oriented
    - Unicode based
- The lowest level is always byte-oriented
- Character stream is just an abstraction (for convenience)

Byte streams

- Byte Streams are defined by two abstract classes
  - `InputStream`
    - Read data from a source “into” a program
  - `OutputStream`
    - Write data from a program “out to” a destination



- InputStream and OutputStream define stream handling methods, i.e.
  - `InputStream.read()`
  - `OutputStream.write()`
- Most of these stream handling methods are defined as abstract
  - Enforces redefinition by derived classes



try-with-resource

- statements inside () are automatically closed at end of the block → `AutoClosable`

```java
try {
    fin = new FileInputStream(path);
    return fin.read();
} catch (IOException e) { System.out.println("Error: " + e.getMessage()); }
finally {
    try { if (fin != null) fin.close(); } catch (IOException e) {;}
}
```

Try-with-resources - Same semantic as `finally` statement

```java
try (FileInputStream fin = new FileInputStream(path)) {
    return fin.read();
} catch (IOException e) { System.out.println("Error: " + e.getMessage()); }
```

Reading from / Writing to Files (Byte-oriented)

```java
public class FileStreamReadWrite {
    public static void main(String[] args) {
        String sourcePathName = (args.length >= 1)? args[0] : "./Demo.txt";
        String targetPathName = (args.length >= 2)? args[1] : "./DemoCopy.txt";
        try (FileInputStream fin = new FileInputStream(sourcePathName);
             FileOutputStream fout = new FileOutputStream(targetPathName)) {

            // Copy (READ, WRITE) File *byte by byte*
            int byteValue; // variable to hold a value
            do {
                byteValue = fin.read();     // read next byte from fin
                if(byteValue != -1) {       // if not reached end of file
                    fout.write(byteValue);  // write byte value to fout
                }
            } while(byteValue != -1);       // reached end of file ?

        } // fin & fout will be automatically closed here ( -> try-with-resource)
        catch(FileNotFoundException e) { System.out.println("File not found: " + e.getMessage()); }
        catch(IOException e) { System.out.println("IO Error: " + e.getMessage()); }
    }
}
```

why not byte?

The value returned is an unsigned byte → 0..255
Java **byte** has a range from -128..+127
and **-1** is used to signal **end of file (EOF)**

Reading/Writing

- Reading and writing byte-by-byte is not optimal for most IO devices → `BufferedInputStream`

```java
try (BufferedInputStream in = new BufferedInputStream(new FileInputStream(sourcePathName));
     BufferedOutputStream out = new BufferedOutputStream(new FileOutputStream(targetPathName))) {
    int totalBytesCopied = 0;
    int bytesRead;
    byte[] byteBuffer = new byte[32];
    while ((bytesRead = in.read(byteBuffer)) != -1) { // read available bytes, until end of file
      out.write(byteBuffer, 0, bytesRead);
      totalBytesCopied += bytesRead;
    }
    System.out.println("Copied bytes: " + totalBytesCopied);
} // in & out will be automatically closed here ( -> try-with-resource)
```

Reads max. `byteBuffer` size of bytes and returns number of bytes read.
-1 if end of file.

Writes `bytesRead` bytes from `byteBuffer` starting on index `0`.

Character streams

- Character Streams are defined by two abstract classes
  - `Reader`
    - Read characters from a source “into” a program
  - `Writer`
    - Write characters from a program “out to” a destination
- Reader and Writer define a basic stream handling methods, i.e.
  - `Reader.read()` a single / array of character
  - `Writer.write()` a single / array of character
- Most of these stream handling methods are defined as abstract
  - Enforces redefinition by derived classes

Character set

→ for text based reading and writing it is essential to specify the used character set

> 💡 A character set is used to convert a char to bytes and vice versa.

```java
try (FileReader reader = new FileReader(sourcePathName, Charset.forName("UTF-8"));
     FileWriter writer = new FileWriter(targetPathName, StandardCharsets.ISO_8859_1)) {
    // read character-by-charachter
    int charValue;   // variable to hold a value
    do {
        charValue = reader.read();    // read next char from file
        if(charValue !=-1)            // if not end of file
            writer.write(charValue);  // write char value to console
    } while(charValue != -1);  // reached end of file ?
```

Use specific Charsets for reading and writing

Convert Byte-Stream to Char-Stream

- For some sources / destinations no Reader/Writer classes are available (e.g. SocketI/OStreams, PipedI/OStream)
- Use Decorator Classes to convert between Byte- and Char Stream
  - **`InputStreamReader`** for reading chars from byte streams
  - **`OutputStreamWriter`** for writing chars to byte streams
- Both classes take `Charset` arguments to convert to / from the right character set.

```java
public class InputStreamReaderWriter {
  public static void main(String args[]) throws IOException {
    String sourcePathName = (args.length >= 1)? args[0] : "./Demo.txt";
    String targetPathName = (args.length >= 1)? args[0] : "./Demo-iso-8859-1.txt";
    try (Reader reader = new InputStreamReader(new FileInputStream(sourcePathName), StandardCharsets.UTF_8);
         Writer writer = new OutputStreamWriter(new FileOutputStream(targetPathName), StandardCharsets.ISO_8859_1)){
         // read character-by-charachter
         int charValue;   // variable to hold a value
         do {
             charValue = reader.read();    // read next char from file
             if(charValue !=-1)            // if not end of file
                 writer.write(charValue);  // write char value to console
         } while(charValue != -1);  // reached end of file ?
```

Converting bytes to/from char using the specified Charset
If none is specified the defaultCharset will be used

BufferedReader

```java
public static void main(String[] args) throws IOException {
    try (BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
         BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(System.out))) {
        System.out.println("Enter lines of text.");
        System.out.println("Enter 'stop' to quit.");
        String line;
        do {
            line = br.readLine();
            bw.write(line);
            bw.newLine();
        } while(!line.equals("stop"));
    } // br & bw will be automatically closed here ( -> try-with-resource)
```

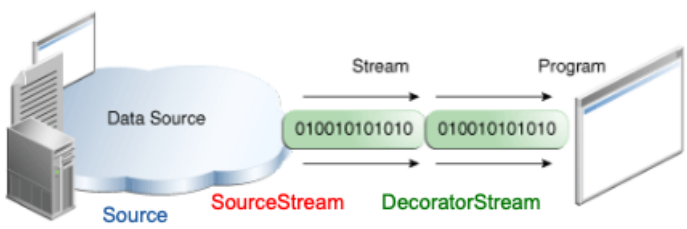
Decorator pattern

- add additional features or behaviour to an existing stream/class

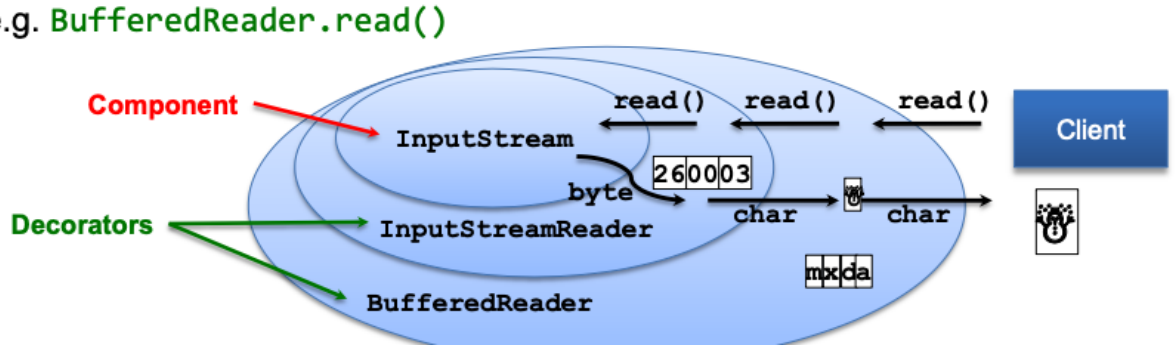
```
DecoratorStream stream = new DecoratorStream(new Source/TargetStream(src/tgt));
```

add methods / behavior and use wrapped stream for reading/writing

read/write data from source/target



– e.g. `BufferedReader.read()`



Positioning

Reading

For reading, `InputStream` and `Reader` support limited navigation within the stream:

```
skip(long n) // skip a number of bytes/chars
available() // returns the estimate number of bytes/chars until EOF
```

```
public class FileInputStreamPositioning {
    public static void main(String args[]) throws IOException {
        try (InputStream file = new FileInputStream("demo/Demo.txt");
             PrintWriter console = new PrintWriter(System.out, true, StandardCharsets.ISO_8859_1))
        {
            int size= file.available();
            int n = size/40;
            console.println("* First " + n +" bytes of the file one read() at a time");
            for (int i=0; i < n; i++) {
                console.print((char) file.read());
            }
            console.println("\n* Still Available: " + (size = file.available()));
            file.skip(size/2);   // forward file pointer half of the remaining bytes
            console.println("* Skipped "+(size/2)+" Bytes still available: " + file.available());
            n = size/5;
            for (int i=0; i < n; i++) {
                console.print((char) file.read());
            }
            console.println("\n* Still Available: " + file.available());
        }
    }
```

Mark

Some InputStreams support: (BufferedInputStream/Reader, ByteArrayInputStream/Reader)
- `mark(int readLimit)` marks the current position and keeps the mark for at least `readLimit` bytes/chars.
- `reset()` resets the current position in the stream to the set mark

Writing to files

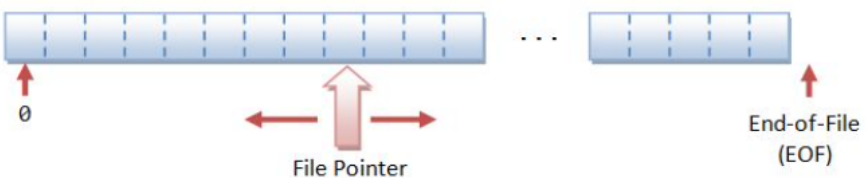
→ removing or inserting content into a file requires the use of temp files (read original file, write to temp file skipping or inserting content, move temp file to original file)

File on the filesystem is stored as blocks of bytes with size `2^n` . That's the reason why it's not possible to insert something at the middle of a file e.g.

Random access files

→ "wahlfreier Zugriff"

The File Pointer position can be read by the `getFilePointer()` method, set absolute by the `seek()` method and relativ by the `skipBytes()` method



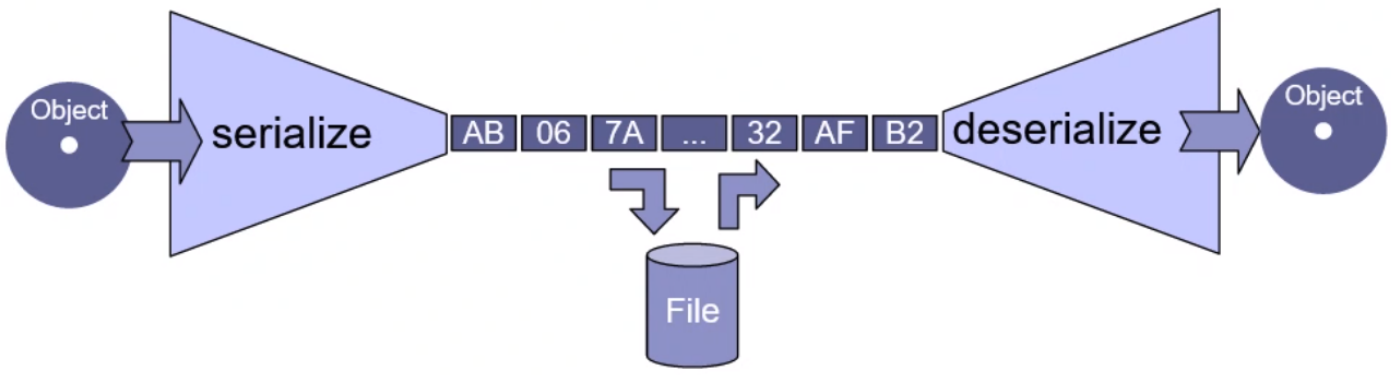
- Reading and writing moves the File Pointer to the next position after the last accessed byte.
- Reading over the end of file (EOF) mark throws an `EOFException`
- Writing always overwrites the following bytes (no insert), like in an array
- Writing over the end of the file (EOF) automatically extends the file
- Not set bytes have a default value of 0

Serialization / Deserialization

Sending objects over the wire

Serialization: Converting/saving objects into a Byte-Stream

Deserialization: Recovering/reading objects from a Byte-Stream



Object streams

→ Objects to be serialized must implement the `Serializable` marker interface (no methods)

```
Employee harry = new Employee("Dirty Harry", 50000, LocalDate.of(1967, 3, 11));
Manager boss = new Manager("Walter Smith", 80000, LocalDate.of(1950, 12, 4)); // Manager is extending Employee
boss.setAssistant(harry);

// Save (serialize) two objects to the file empolyee.dat
try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("employee.dat"))) {
    // objects are written in the given order to the file
    out.writeObject(harry);  // write object Dirty Harry
    out.writeObject(boss);   // write object Walter Smith
    out.writeInt(12);        // write value of primitive type int
}
```

```
// Load (deserialize) two objects from the file employee.dat
try (ObjectInputStream in = new ObjectInputStream(new FileInputStream("employee.dat"))) {
   // number and order of reading the objects must match, also the type must be compatible (match or super class)
   Employee e1 = (Employee) in.readObject();   // object for Dirty Harry
   Employee e2 = (Employee) in.readObject();   // object for Walter Smith
   // e2.getClass().getName() would return type Manager
   int count = in.readInt();                   // read primitive type int
}
```

Stream content

1. Magic number (2 byte): AC ED
2. Version number (currently): 00 05
3. Class description of all serialized objects (incl. superclasses)
   - full name of the class (incl. package path)
   - unique version id of class (`serialVersionUID`)
     - SHA hash over class description
       (class, superclasses, interfaces, fields, method signatures,...)
     - final constant `serialVersionUID` defined within the class (avoid that small changes change the hash)
   - description of the data fields
     - type code (1 byte), length of the field name (2 byte), field name, class name (if Object type)
       - type codes: B=byte, C=char, D=double, F=float, I=int, J=long, L=Object, S=short, Z=boolean, [=Array
       - example: `I 00 06 salary` → `int salary`
       - example: `L 00 07 hireDay 74 00 10 Ljava/util/Date` → `java.util.Date hireDay`
4. Values of the serialized objects

Transient fields

→ used for fields that cannot be serialized / deserialized

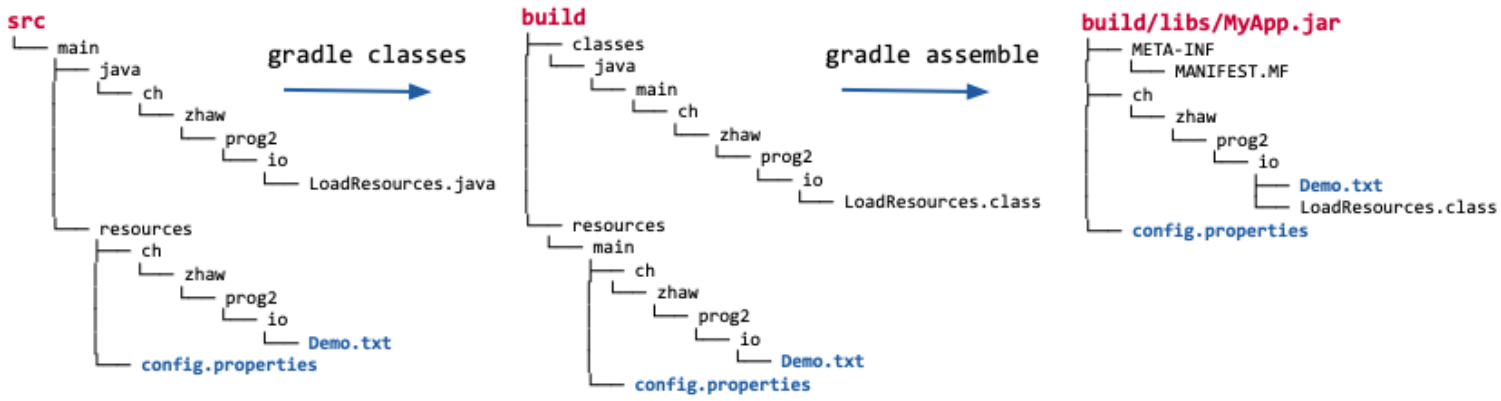
Example: window handles, file descriptors, network socket, db connections

```
private transient Date createDate = new Date(); // default value necessary
```

Resource files

Resource files get copied to the `build` directory during the build process (`gradle build`)

→ usually only files that don't need to be changed



Access

- Using the Classloader*)

```
// get the classLoader
ClassLoader contextClassLoader = Thread.currentThread().getContextClassLoader();
ClassLoader classLoader        = this.getClass().getClassLoader();

// get resource path string using classloader
String resourceRoot = classLoader.getResource("").getPath();                        // "../build/classes/java/main/"
String configPath   = classLoader.getResource("config.properties").getPath();      // "../build/resources/main/config.properties"
String demoTextPath = classLoader.getResource("ch/zhaw/prog2/io/Demo.txt").getPath(); // "../b../r../main/ch/zhaw/prog2/io/Demo.txt"
// get input stream for resource file
InputStream stream  = classLoader.getResourceAsStream("config.properties");
```

`getResource()` is looking up in the classpath, which in IDEs includes `build/classes/java/main/` as well as `build/resources/main/`. If the file/dir is available in both paths, `build/classes/` is prefered.

`getResource()` is returning an object of Type `java.net.URL` in the form "**`file:`**`/path/to/Demo.txt`". `getPath()` returns the path component of the URL.

`getResourceAsStream()` directly returns the content in a `java.io.InputStream`, which can be used with the Java IO-Streams API. This works also for resources within JAR archives.

- Using a Class object

```
// getting the class instance
Class staticClass = MyClass.class;
Class clazz       = this.getClass();

// getting resource path string using class instance
String classPath    = clazz.getResource("").getPath();                  // "../build/classes/java/main/ch/zhaw/prog2/io"
String configPath   = clazz.getResource("/config.properties").getPath(); // "../build/resources/main/config.properties
String demoTextPath = clazz.getResource("Demo.txt").getPath();          // "../build/resources/main/ch/zhaw/prog2/io/Demo.txt"
// get input stream for resource file
InputStream stream  = classLoader.getResourceAsStream("Demo.txt");
```

`Class.getResource()` applies the given resource path relative to the class-file. A path starting with **`/`** is using an absolute path from resource root.

*) Classloaders are looking up and loading the class files from the directories or jar-archives referenced in the classpath into memory and create Class-Instances.

Properties

```
# Property file rules (see also java.util.Properties or https://en.wikipedia.org/wiki/.properties)
# Format is "key = value" or "key : value" (each pair on a new line)
website = https://en.wikipedia.org/
language : English

# Keys are often using dotted names to represent hierarchical values
dir.log = ./logs

# A backslash means to continue reading the value onto the next line.
message.welcome = Welcome to \
                  Wikipedia!
# If keys or values contain a backslash, it should be escaped by another backslash
path=c:\\wiki\\templates
```

Usage

```
// create Properties object (similar to a Map)
Properties config = new Properties();

// load entries from a properties file
config.load(Files.newBufferedReader("./demo/config.properties"));                // from user file
config.load(getClass.getClassloader().getResourceAsStream("config.properties")); // from resource file

// accessing properties
String appName = config.getProperty("name");  // returns the value matching the key "name" (null if not available)
String lastUsed = config.getProperty("date.used", LocalDate.now().toString()); // provide default value if not available
```

```
// getting all available properties
for (String propertyName : config.stringPropertyNames()) {
    System.out.printf("- Name: %s -> Value: %s%n", propertyName, config.getProperty(propertyName));
}

// setting properties
config.setProperty("date.used", LocalDate.now().toString());

// store entries to a properties file
config.store(Files.newBufferedWriter("./demo/config.properties"));
```

Logging

`java.util.logger`

- Each logger instance is identified by a globally unique name → Singleton (= instance with specific name exists only once)

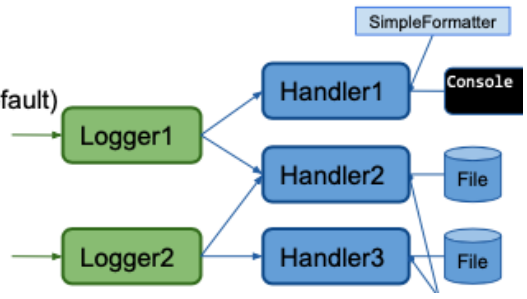
Levels

- Log Levels are ordered and internally represented by an integer value
  - Higher value → higher significance / severity → used for alerting (e.g. WARNING, SEVERE)
  - Lower value → more fine-grained / more details → used for debugging (e.g. FINE, FINEST)
- Seven pre-defined System-levels exist as static variables of class `Level` *)
  - SEVERE (1000), WARNING (900), INFO (800), CONFIG (700), FINE (500), FINER (400), FINEST (300)
  - user-defined levels are possible
- Log Levels can be used to "filter" log messages; i.e. record and show only relevant log messages (see Controlling Log Output)

Handlers

→ write log-records to specific channels (console, file, ...)

- Loggers send messages to 1..n Log Handlers
  - Handlers are responsible to write the log messages to a specific channel
    - `ConsoleHandler` (PRESET), is connected to `System.err`
    - `FileHandler` – writes log messages to a file. It allows rollover to a set of files, e.g. `new FileHandler("myapp.%g.log", 1024*1024, 10)` will create up to 10 files of max. 1MB (`myapp.0.log,myapp.1.log,..., myapp.9.log`) whereas lower generation (%g) numbers contain newer entries
    - `StreamHandler` – writes to a byte stream
    - `SocketHandler` – writes to a remote socket (raw TCP channel, no HTTP or similar)
    - `MemoryHandler` – writes to a memory buffer
  - The same Handler can receive messages from multiple Loggers
    - add Handler to Logger(s) using `logger.addHandler()` method
  - Handlers use Formatters to format the log messages
    - `SimpleFormatter` – Line based summary based on format pattern (Default)
    - `XMLFormatter` – Formatting records as XML-Elements (default for FileHandler)



- Better option is to defer building the log message using a Supplier function, called only if the message will actually be logged:
  - All log functions also accept a Supplier function as an argument. e.g.:

```
logger.log(Level level, java.util.function.Supplier<String>);
// Usage:
logger.log(Level.FINE, () -> String.format("Current Status: status = %s", detectStatus())
logger.log(Level.FINE, this::detectStatus);                    // static function referen
logger.log(Level.FINE, new Supplier<String>() {                // with inner class
    String get() { return String.format("Current Status: status = %s", detectStatus()); }
});
```

Config

```
public class LogConfiguration {
    // class logger: "ch.zhaw.prog2.io.LogConfiguration"
    private static final Logger logger = Logger.getLogger(LogConfiguration.class.getCanonicalName());

    public LogConfiguration() throws IOException {
        // load base configuration from config file in root resources directory (class path root)
        InputStream logConfig = this.getClass().getClassLoader().getResourceAsStream("log.properties");
        LogManager.getLogManager().readConfiguration(logConfig);

        ...
    }
}
```

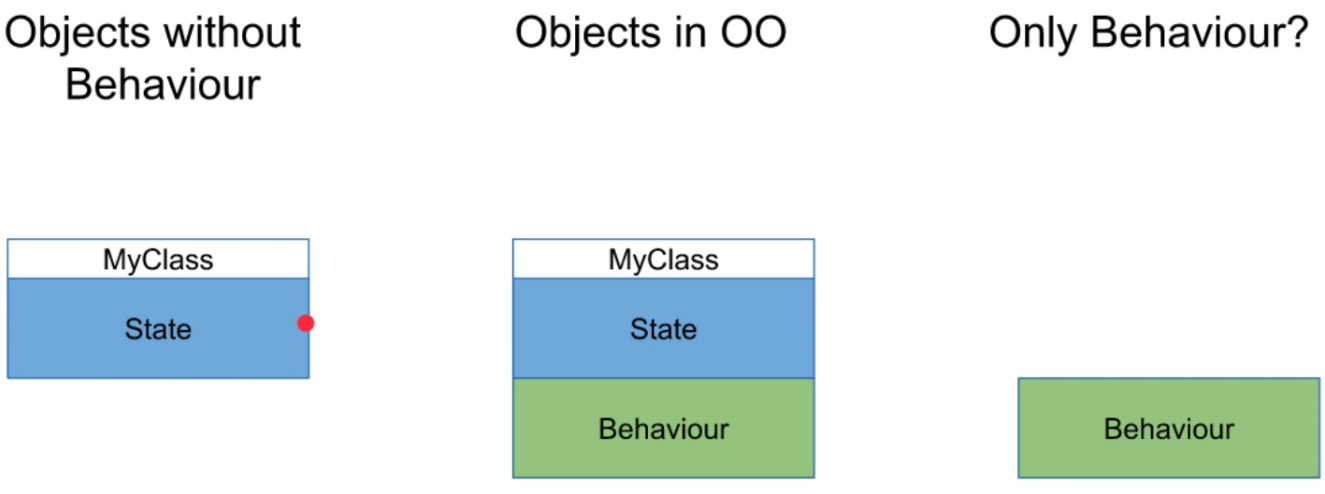
- Using java properties format

Only one handler of each type can be configured (others need to be added and configured programmatically, see Annex)

```
## configure handlers
java.util.logging.ConsoleHandler.level = ALL

## File handler configuration
## see https://docs.oracle.com/en/java/javase/11/docs/api/java.logging/java/util/logging/FileHandler.html
java.util.logging.FileHandler.level = ALL
# %g = generation number, %u = unique number to resolve conflicts
java.util.logging.FileHandler.pattern = log-%g-%u.log
# use SimpleFormatter instead of default XMLFormatter
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
java.util.logging.FileHandler.encoding = UTF-8
# max log file size in byte before switching to next generation (=1kB); 0 = unlimited
java.util.logging.FileHandler.limit = 1024
# max number of generations (%g) before overwriting (5 -> 0..4)
java.util.logging.FileHandler.count = 5
java.util.logging.FileHandler.append = true

## configure Formatter (see SimpleFormatter documentation)
java.util.logging.SimpleFormatter.format = [%1$tc] %4$s: %5$s {%2$s}%6$s%n
...
```

```
## configure default log level (for all loggers, if not overwritten below)
.level = INFO

## configure root logger ""
handlers = java.util.logging.ConsoleHandler
level = INFO

## Application specific logger configuration
# loggers starting with "ch.zhaw.prog2.io" -> write to console and file and do not forward to parent handlers
ch.zhaw.prog2.io.level = FINE
ch.zhaw.prog2.io.handlers = java.util.logging.FileHandler, java.util.logging.ConsoleHandler
ch.zhaw.prog2.io.useParentHandlers = false

# logger for class ch.zhaw.prog2.io.LogConfiguration
ch.zhaw.prog2.io.LogConfiguration.level = FINEST
```

Functional



Lambdas

A lambda expression is like an anonymous method (in reality the method is not anonymous, its name is **derived by the compiler from the method in the functional interface**)

1. `->` operator
2. type inference

```
(observable, oldValue, newValue) -> canvas.setWidth(newValue.doubleValue());

// instead of
vbox.widthProperty().addListener(new ChangeListener<Number>() {
  public void changed(ObservableValue<? extends Number> o, Number oldVal, Number newVal) {
       canvas.setWidth(newVal.doubleValue());
   }
});
```

```java
@FunctionalInterface
public interface ChangeListener<T> {
    void changed(ObservableValue<? extends T> observable,
            T oldValue, T newValue)
}
```

Operator

- creates an instance of a functional interface → this instance is a class

→ does not execute anything yet!

Functional interface

Docs: https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/function/package-summary.html

- interface with only one abstract method (aside from the methods of `Object`)

`@FunctionalInterface` annotation

- default and static methods are allowed

```java
interface Runnable {
  void run();
}

interface Func extends NonFunc {
  int compare(String o1, String o2);
  default void whatever() {};
}

interface Comparator<T> {
  boolean equals(Object obj);
  int compare(T o1, T o2);
}
```

```java
interface NonFunc {
  boolean equals(Object obj);
}

interface Foo {
  int m();
  Object cloneMe();
}
```

Examples:

```java
for (int i = 1; i < 4; i++) {
    IntToLongFunction intSquare = x -> x * x;
    System.out.println(i + ":" + intSquare.applyAsLong(i));
}
```

Demo

```
x -> x * x
uses the interface
```

```java
@FunctionalInterface
public interface IntToLongFunction {
    long applyAsLong(int var1);
}
```

Method references

A method reference expression is used to refer to the invocation of a method without actually performing the invocation.

```java
maxDemo.showByFunction((a, b) -> Math.max(a, b));
maxDemo.showByFunction(Math::max);
```

Is handled as
(a, b) -> Math.max(a, b)

Functional composition

- States are passed from function to function → function composition

```
DoubleUnaryOperator addOne = x -> x + 1;
DoubleUnaryOperator divFour = x -> x / 4;   double points = 13;
DoubleUnaryOperator
pointToGrade = divFour.andThen(addOne);  // divide and then add
```

New Function

```
pointToGrade = addOne.compose(divFour);  // add, but divide before

pointToGrade.applyAsDouble(points);      // apply function
```

Predicate

```
IntPredicate isEven = x -> x % 2 == 0;
IntPredicate isDividableBy3 = x -> x % 3 == 0;
IntPredicate isDividableBy6 = isEven.and(isDividableBy3);
if (isDividableBy6.test(24)) { ... }
```

`Optional<T>`

- final & immutable

```
public Optional<User> getLoggedInUser() {
   if (hasLoggedInUser) {
       User user = new User("Muster Andrea", "muster.andrea");
       return Optional.of(user);
   } else {
       return Optional.empty();
   }
}
```

First class citizens

Functions can be assigned to variables, passed as arguments and returned as result

– Like objects functions can be

- assigned to variables
  `IntToLongFunction intSquare = x -> x * x;`
- passed as arguments
  ```
  public long doSomeThing(IntToLongFunction function) { ... }
  long result = doSomeThing(x -> x * x);
  ```
- returned as results
  ```
  public static ToLongBiFunction<Long, Integer> functionFactory(...) {
     return (longValue, intValue) -> longValue + intValue;
  }
  ```

Pure functions

- Return value depends only on the input elements
- have no side effect → no modification of (external) state

```
// pure function
public class Calculator {
    public int sum(int a, int b) {
       return a+b;
    }
}
```

```
// function with side effect
public class Calculator {
    private int amount = 0;
    public int sum(int a) {
        amount += a
        return amount;
    }
```

Avoiding Side effects

→ use immutable objects

→ do not modify global variables or shared mutable objects

Streams

Docs: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html

```
Stream<Integer> listStream = list.stream();

IntStream intArrayStream = Arrays.stream(intArray);
intArrayStream.forEach(System.out::println);
```

What do we do here? *(imperative!)*

```
List<Integer> list = List.of(3, 2, 12, 5, 6, 11, 13);
int evenSum = 0;
for (Integer i: list) {
    if (i % 2 == 0) {
       evenSum += i;
    }
}
```

Same, but using Java functional Streams: *(declarative…)*

```
List<Integer> list = List.of(3, 2, 12, 5, 6, 11, 13);
int evenSum = list.stream()
                .filter(i -> i % 2 == 0)
                .mapToInt(Integer::intValue)
                .sum();
```

Imperative

- give instructions line by line → how to achieve a result

Declarative

- focus on what we want to achieve → delegating the how to the library / framework

Stream API achieves this by using internal iteration along with lambda expressions

Functional Streams

- stream represents an infinite sequence of data elements

Lifecycle

Divided into three types of operations:

- A source operation to obtain a stream from a data source
- Zero or more intermediate operations which are transforming the stream into another stream
- A terminal operation which produces the final result.


```
List<Integer> list = List.of(3, 2, 12, 5, 6, 11, 13);
int evenSum = list.stream()                // (1) Stream<Integer> [3,2,12,5,6,11,13]
                .filter(i -> i % 2 == 0) // (2) Stream<Integer> [2,12,6]
                .map(Integer::intValue)  // (3) IntStream [2,12,6]
                .sum();                  // (4) int 20
```

Intermediate operation

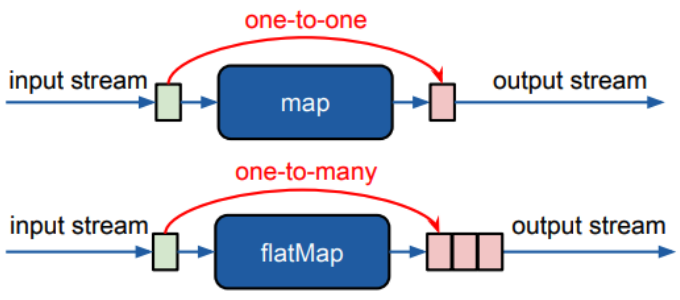
- transform a stream into another steam

Terminal operation

- triggers the processing of the pipeline → lazy evaluation
- produces explicit result

`flatMap`

- flatten nested collections




```
printStreamContent("FlatMap result:",
List.of(List.of(2,3,6,5), List.of(12,13,11
    .stream()  // Stream<List<Integer>>
    .flatMap(List::stream)
);
```

```
SourceList: [[2,3,6,5],[12,13,11]]
FlatMap result: [2,3,6,5,12,13,11]
```

`reduce`

- aggregates results (terminal operation)

Identity: element that is the initial value of the reduction operation and the default result if the stream is empty (e.g. 1 for multiplication)

Interface	Description

`reduce(BinaryOperator<T> accumulator)`	Aggregates all elements of the stream of type T into a single result using an accumulator function. returns an Optional
	`integerStream.reduce((sum, nextValue) -> sum + nextValue)`
	`integerStream.reduce(Integer::sum)`

`reduce(T identity,` `BinaryOperator<T> accumulator)`	Aggregates all elements of the stream of type T into a single result using an accumulator function, using `identity` as startValue `integerStream.reduce(0, Integer::sum)`

`reduce(U identity,`
`BiFunction<U,T,U> accumulator,`
`BinaryOperator<U,U> combiner)`
	Aggregates all elements of the stream of Type T into a single result of type U, using an accumulator function, using `identity` as startValue. Combiner function is only used when merging parallel streams
	`integerStream.reduce(0,`
	`(sum, nextValue) -> sum + nextValue,`
	`(sum1, sum2) -> sum1 + sum2)`

`collect`

Docs: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Collectors.html

`collect(Collector<T,A,R>)`	Collects the stream values of type T into a container of type R. (using an accumulator function of type A)

`java.util.Collectors` provides a large set of predefined collectors, e.g.

- `toList()` accumulates the elements into a new **`List<T>`**
- `toSet()` accumulates the elements into a new **`Set<T>`**
- `toMap(Function<T,K> keyMapper, Function<T,U> valueMapper)`
  - creates a map using `keyMapper`/`valueMapper` functions to create the key/value
- `groupingBy(Function<T,K> classifier)`
  - creates a map of value lists **`Map<K,List<T>>`**, where the key <K> for each value <T> is generated by the classifier function
- `joining(CharSequence delimiter)`
  - creates a String by joining String elements using the delimiter

```
printObject("Group even/odd numbers:",
    integerList.stream()
       .distinct()
       .collect(Collectors.groupingBy(i -> i % 2 == 0 ? "even":"odd")) // -> Map<String,List<Integer>>
);
```

```
Group even/odd numbers: {even=[6, 2, 12], odd=[3, 5, 11, 13]}
```

```
printObject("Create square Map:",
    integerList.stream()
       .distinct()
       .collect(Collectors.toMap(i -> i, i -> i * i))                // -> Map<Integer,Integer>
);
```

```
Create squareValue Map: {2=4, 3=9, 5=25, 6=36, 11=121, 12=144, 13=169}
```

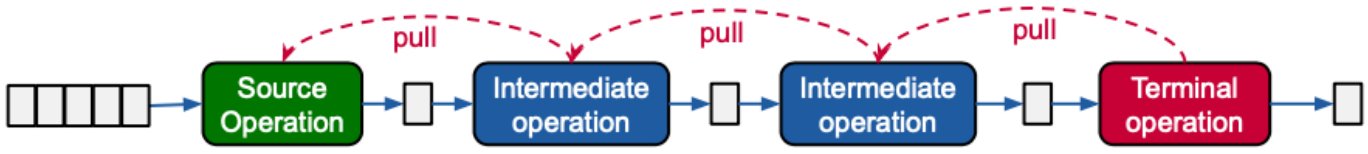
```
printObject("Joining values to CSV String:",
    integerList.stream()
        .map(String::valueOf)
        .collect(Collectors.joining(";", "[", "]"))
 );
```

```
Joining values to CSV String: [3;6;2;5;6;12;11;13]
```

Execution of the pipeline

- processing of the pipeline is optimized → lazy-evaluation, short-circuiting and merging of operations

- The pipeline is only evaluated when the terminal operation is called
- The terminal operations pulls the data, the source does not push it



- Stream characteristics are used to identify optimization
- This allows intermediate operations to be merged
  - avoid multiple redundant passes on data
    (no recurring iterations, avoid intermediate state)
  - short-circuit operations (terminate processing early)
  - evaluate as late as possible → lazy evaluation

Parallel streams

```
.stream().parallel()
```

Calculating 5 million primes on an 8 core processor

Sequential run:

```
Needed 42.23 seconds for 5000000 candidates, 253482 primes found. 0.00 seconds for build stream and 42.23 seconds to put into a collection.
```

Parallel run:

```
Needed 4.80 seconds for 5000000 candidates, 253482 primes found. 0.00 seconds for build stream and 4.80 seconds to put into a collection.
```