

# Java Concurrency

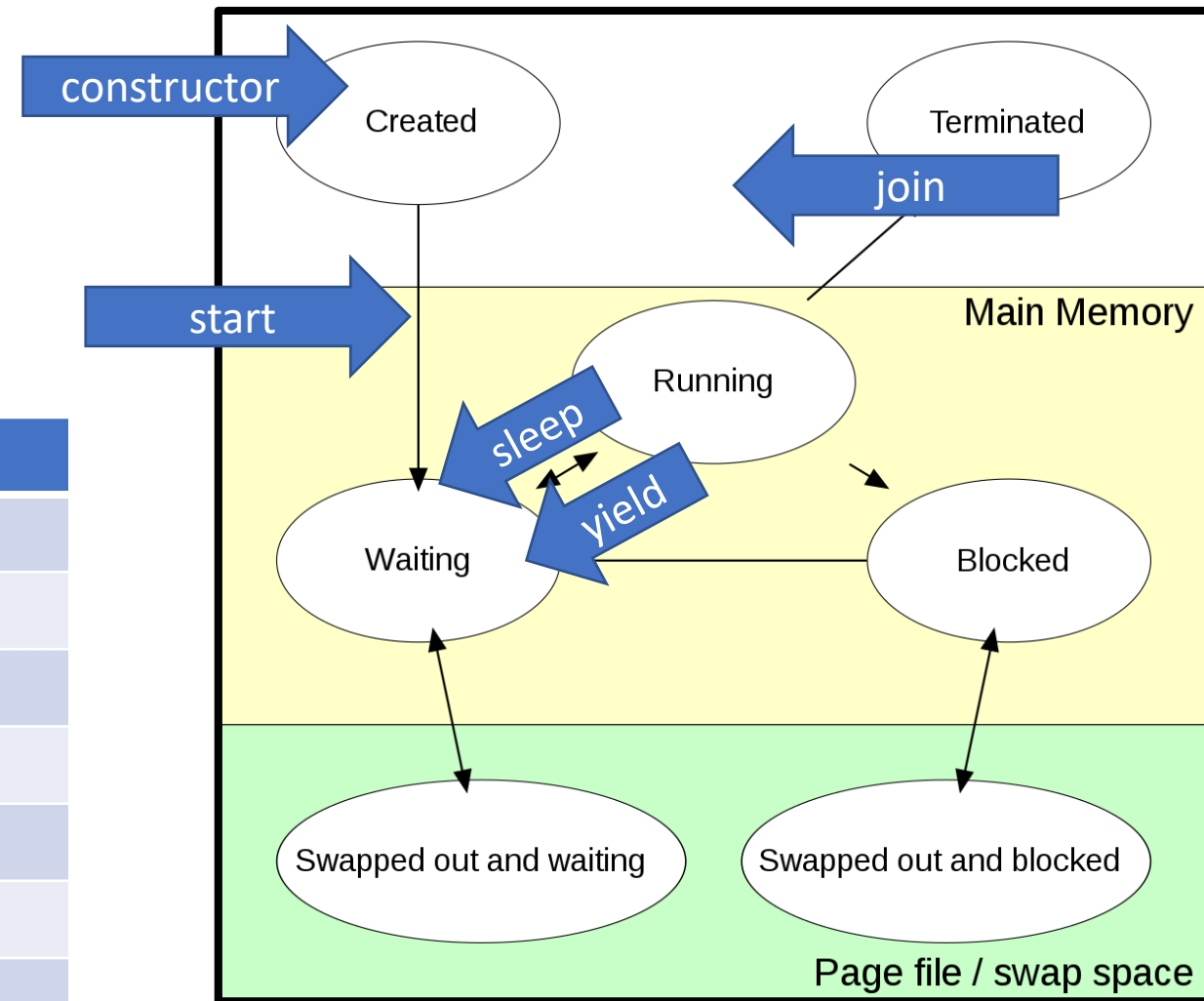
Nir Piterman

TDA 384 / DIT 391

# Creating Threads

- What does a thread need to do?

Method	
start()	Start a thread by calling run() method
run()	Entry point for a thread
join()	Wait for a thread to end
isAlive()	Checks if thread is still running or not
setName()	
getName()	
getPriority()	



[https://en.wikipedia.org/wiki/Process\\_state](https://en.wikipedia.org/wiki/Process_state)

# Extend Thread

```
class MyThread extends Thread
{
    public void run()
    {
        System.out.println("concurrent thread started running..");
    }
}

class MyThreadDemo
{
    public static void main(String args[])
    {
        MyThread mt = new MyThread();
        mt.start();
    }
}
```

# Extend?

## Hierarchy: Animals

- Animal
  - Mammal
    - Canine
      - Dog
      - Wolf
    - Feline
      - Cat
  - Fish
    - Tuna
    - Shark
  - Reptile
    - Crocodile
    - Iguana

## Object - Bank Account

- Accounts have certain data and operations
  - Regardless of whether checking, savings, etc.
- Data
  - account number
  - balance
  - owner
- Operations
  - open
  - close
  - get balance
  - deposit
  - withdraw

## Kinds of Bank Accounts

- Account
  - **Checking**
    - Monthly fees
    - Minimum balance.
  - **Savings**
    - Interest rate
- Each type shares some data and operations of "account", and has some data and operations of its own.

# Implement Runnable

- Java does not support multiple inheritance.
- If you need your class to inherit.

```
class MyThread implements Runnable
{
    public void run()
    {
        System.out.println("concurrent thread started running..");
    }
}

class MyThreadDemo
{
    public static void main(String args[])
    {
        MyThread mt = new MyThread();
        Thread t = new Thread(mt);
        t.start();
    }
}
```

# Data Races

## Data races

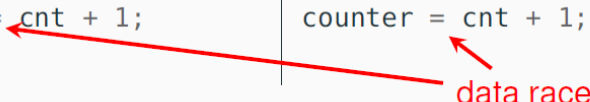
Race conditions are typically caused by a **lack of synchronization** between threads that **access shared** memory.

A **data race** occurs when two concurrent threads

- access a shared memory location,
- at least one access is a **write**,
- the threads use no explicit **synchronization mechanism** to protect the shared data.

```
int counter = 0;
```

	thread t		thread u	
	<pre>int cnt;</pre>		<pre>int cnt;</pre>	
1	<pre>cnt = counter;</pre>		<pre>cnt = counter;</pre>	1
2	<pre>counter = cnt + 1;</pre>		<pre>counter = cnt + 1;</pre>	2

 data race

6/46

## Concurrency humor

*Knock knock.*

– “Race condition.”

– “Who’s there?”

5/46

# Locks

## Lock implementations in Java

The most common implementation of the `Lock` interface in Java is `class ReentrantLock`.

### Mutual exclusion:

- `ReentrantLock` guarantees **mutual exclusion**

### Starvation:

- `ReentrantLock` does **not** guarantee freedom from starvation by default
- however, calling the constructor with `new ReentrantLock(true)` “favors granting access to the **longest-waiting** thread”
- this still does not guarantee that thread **scheduling** is fair

### Deadlocks:

- one thread will succeed in acquiring the lock
- however, deadlocks may occur in systems that use multiple locks (remember the dining philosophers)

```
interface Lock {  
    void lock();    // acquire lock  
    void unlock(); // release lock  
}
```

# Implicit Locking

## Built-in locks in Java

Every object in Java has an **implicit** lock, which can be accessed using the keyword **synchronized**.

Whole method locking  
(**synchronized methods**):

```
synchronized T m() {  
    // the critical section  
    // is the whole method  
    // body  
}
```

Every call to `m` **implicitly**:

1. acquires the lock
2. executes `m`
3. releases the lock

Block locking  
(**synchronized block**):

```
synchronized(this) {  
    // the critical section  
    // is the block's content  
}
```

Every execution of the block **implicitly**:

1. acquires the lock
2. executes the block
3. releases the lock



# Combinations?

**Can you mix? Which of these work?**

- Mix 1
- Mix 2
- Both
- None



<http://etc.ch/Wmtx>

# Semaphores

```
interface Semaphore {  
    int count();    // current value of counter  
    void up();      // increment counter  
    void down();    // decrement counter  
}
```

## Mutual exclusion for **two** processes with semaphores

With semaphores the entry/exit protocols are trivial:

- initialize semaphore to 1
- **entry protocol**: call `sem.down()`
- **exit protocol**: call `sem.up()`

```
Semaphore sem = new Semaphore(1);
```

	thread t	thread u	
	<code>int cnt;</code>	<code>int cnt;</code>	
1	<code>sem.down();</code>	<code>sem.down();</code>	5
2	<code>cnt = counter;</code>	<code>cnt = counter;</code>	6
3	<code>counter = cnt + 1;</code>	<code>counter = cnt + 1;</code>	7
4	<code>sem.up();</code>	<code>sem.up();</code>	8

The implementation of the Semaphore interface guarantees mutual exclusion, deadlock freedom, and starvation freedom.

# Speed?

## Which one is fastest?

- No synchronization
- Locks
- Synchronization
- Semaphores



<http://etc.ch/Wmtx>

# Polling vs waiting

## Common reasons for rejection

Using *polling/busy waiting* for synchronization is a common mistake that leads to submissions being *rejected*.

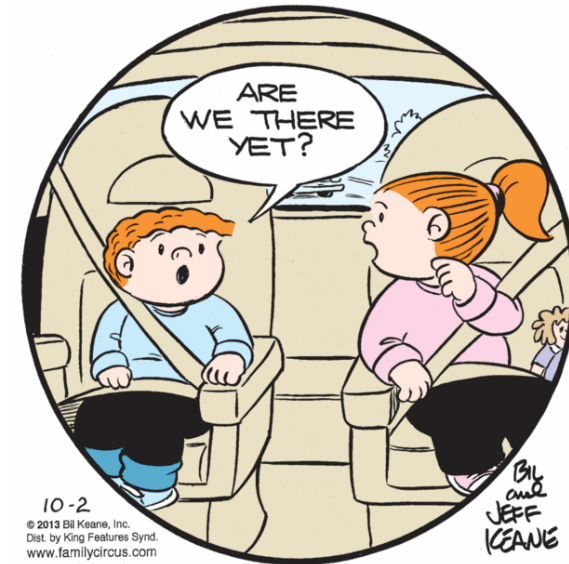
Here are some examples of polling/busy waiting in pseudo code. Loops that behave similarly to the situations below (where the dots do not include any *blocking wait*) are considered as polling.

```
while (e) { // POLLING!  
  ...  
  sleep(t);  
  ...  
}
```

Using a blocking operation within a loop is not considered as polling

```
while (e) { // NO POLLING!  
  ...  
  wait(o);  
  ...  
}
```

**provided** that it is **not** the case that that the waiting process is woken up from its wait at regular intervals. Thus, the following example is also an instance of polling:



“It depends on where you think we’re going.”