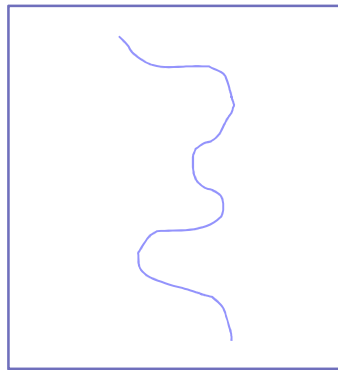# Basics of programming 3

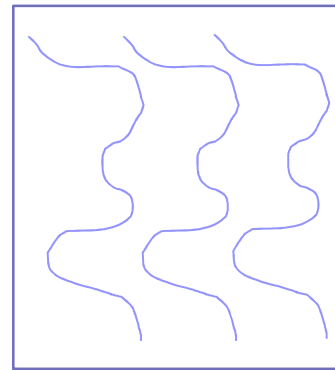## Multithreading in Java

Thread basics

- **Motivation**
  - ☐ in most cases sequential (single threaded) applications are not adequate
  - ☐ it's easier to decompose tasks into separate instruction sequences
  - ☐ e.g.: keyboard handling and graphical update
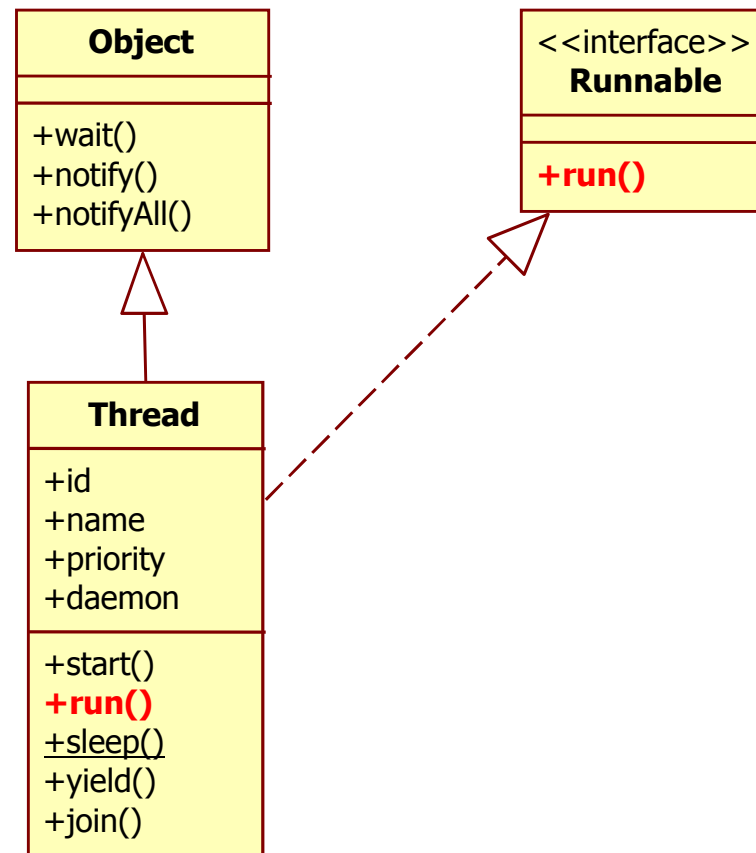
single threaded   multithreaded

Java thread basics

- **Implicit thread handling**
  - class *Thread*, interface *Runnable*
    - representing a JVM thread
  - threads share all memory
    - can have static thread-specific data
  - threads execute methods
  - each thread executing (mostly) independently
  - synchronization by object access (monitors)
  - non-strict priority scheduling

`java.lang.Thread`

```
          Object                          <<interface>>
    ──────────────────                       Runnable
                                         ──────────────────
    +wait()
    +notify()                                +run()
    +notifyAll()
```

```
          Thread
    ──────────────────
    +id
    +name
    +priority
    +daemon
    ──────────────────
    +start()
    +run()
    +sleep()
    +yield()
    +join()
```

Java thread basics

- **Entry point:** *run*
  - □ every thread must have a *void run()* method
  - □ in this method all Java features can be used

- **Starting a thread**
  - □ every thread has a *void start()* method
  - □ it has to be called to start the thread
  - □ starts a new execution thread and calls *run*

- **Implementing a thread**
  - □ inheritance (*Thread*) or delegation (*Runnable*)

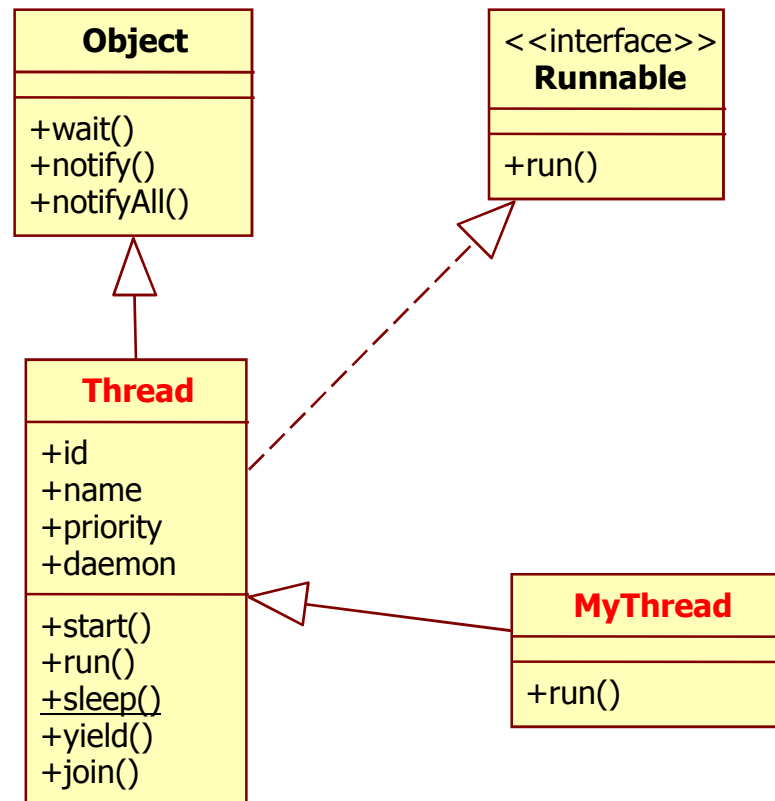# Creating and starting a thread

- Inheritance: extending class `Thread`

```java
public class MyThread extends Thread {
    int a;
    int b;
    public MyThread(int i) { b=i; }
    public void run() {
        for (a = 0; a < b; a++) { System.out.println(a); }
    }
}
```

```java
MyThread mt = new MyThread(1000);
mt.start();
MyThread mt2 = new MyThread(1500);
mt2.start();
...
```

Creating thread: inheritance

- Extending class *Thread*
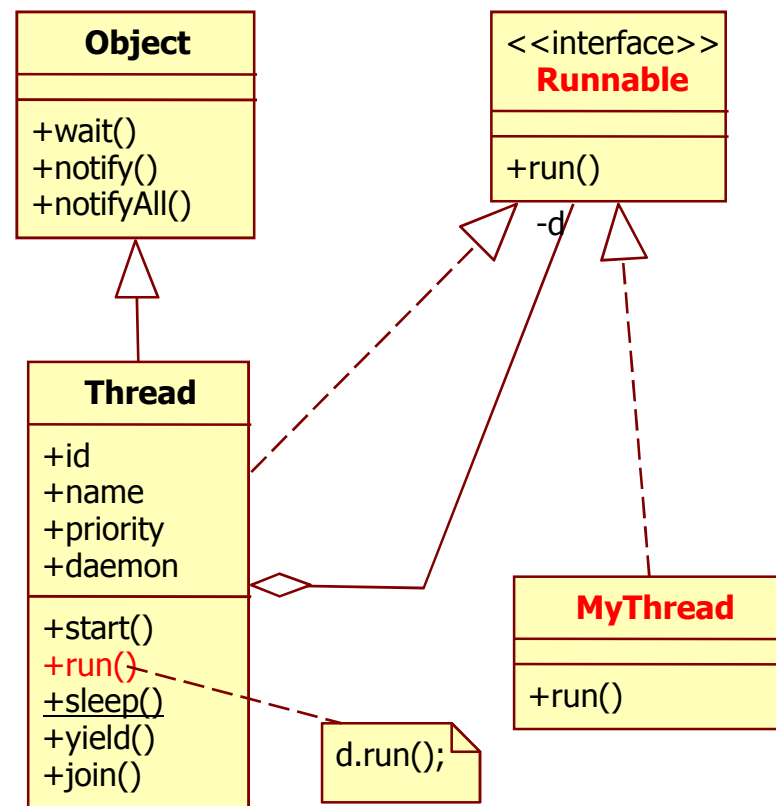
Creating and starting a thread

- Delegation: implementing if. `Runnable`

```
public class MyThread implements Runnable {
  int a;
  int b;
  public MyThread(int i) { b=i; }
  public void run() {
    for (a = 0; a < b; a++) { System.out.println(a); }
  }
}
```

```
MyThread mt = new MyThread(1000);
Thread t = new Thread(mt); // kell egy szál, ami futtat
t.start();
...
```

Creating threads: delegation

■ Implementing interface *Runnable*

java.lang.Thread

- **run()**
  - entry point of the thread (like *main* for an application)
  - does not start thread!

- **start()**
  - starts the thread, calls *run()*

- **sleep(long millis [, int nanos])**
  - thread waits for the given time

java.lang.Thread

- `interrupt()`
  - ☐ interrupts the thread when it's waiting
  - ☐ eg. for *wait*, *sleep*, etc. `InterruptedException`
- `join([long millis [,int nanos]])`
  - ☐ waits for the given thread (for a given time)

- `int getState()`

  - ☐ returns state (runnable, waiting, etc, *see later*)
- `boolean isAlive()`
  - ☐ does is still run?

java.lang.Thread

- **int getId()**
  - □ returns thread id
- **static Thread currentThread()**
  - □ reference to the running thread object
  - □ for accessing the thread executing current code

- **set/getName()**
  - □ thread's name

java.lang.Thread *(opt)*

- **`set/getPriority()`**
  - ☐ thread's priority
  - ☐ is it daemon?

- **`setDaemon(boolean on)`**
- **`boolean isDaemon()`**
  - ☐ sets daemon flag
  - ☐ when JVM stops, it stops all daemon threads
  - ☐ non-daemon threads are waited for

java.lang.Thread *(opt)*

- ## yield()
  - ☐ gives CPU usage to next thread

- ## ThreadGroup getThreadGroup()

  - ☐ returns threadgroup
- ## static int activeCount()

  - ☐ number of active threads in the threadgroup

java.lang.Thread

- Stopping a thread
  - □ *Thread.stop()* method is deprecated
  - □ *volatile*: details later…

```java
private volatile boolean stopSignal;
MyThread(){ stopSignal = false; }
public void stop() { stopSignal = true; }
public void run() {
    while (!stopSignal) {
        do a step or two...
    }
}
```

Theads vs. Objects

- **Objects have**
  - ☐ state (attributes, associations)
  - ☐ behaviour (methods)
- **Threads do**
  - ☐ execute statements described in methods
  - ☐ have *Thread* objects referring to them
    - ▪ c.f. *Thread.currentThread()*
    - ▪ OO API for thread handling

Mutual exclusion

- ## Motivation

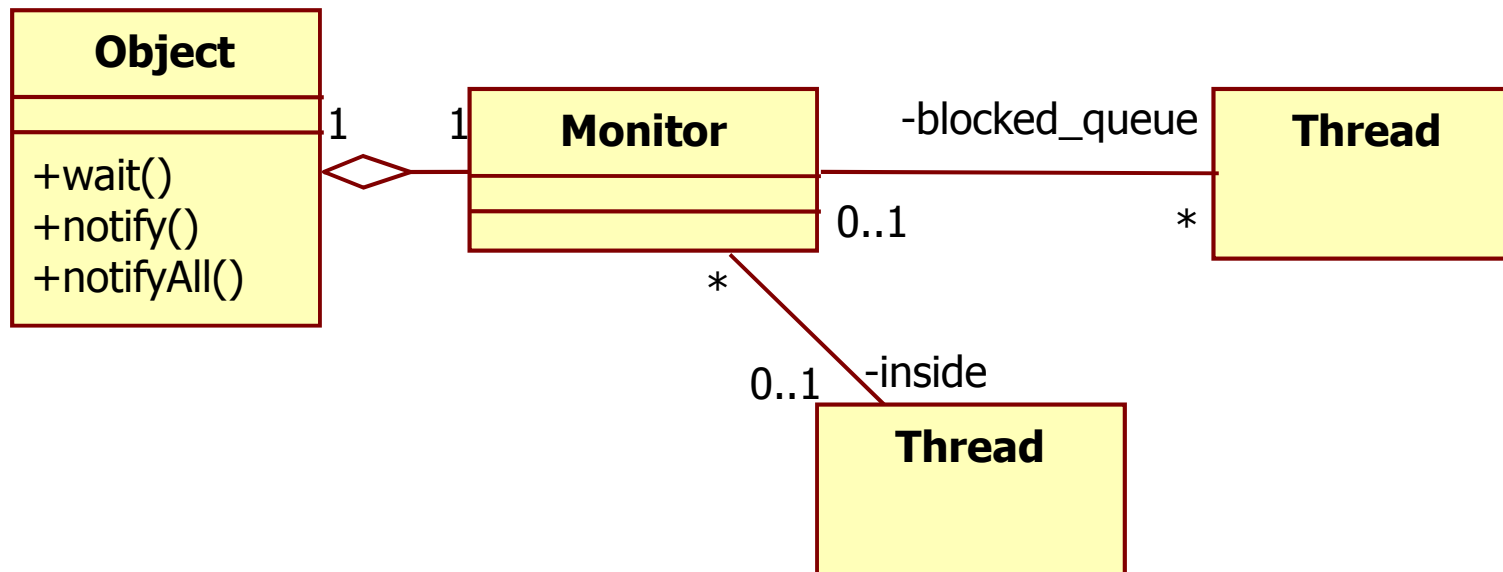 - some resources should be accessed by just a single thread at a time

- ## Every object has its own monitor

  - only one thread allowed inside the monitor

  - other threads must wait in the monitor queue

  - recursive entry is allowed
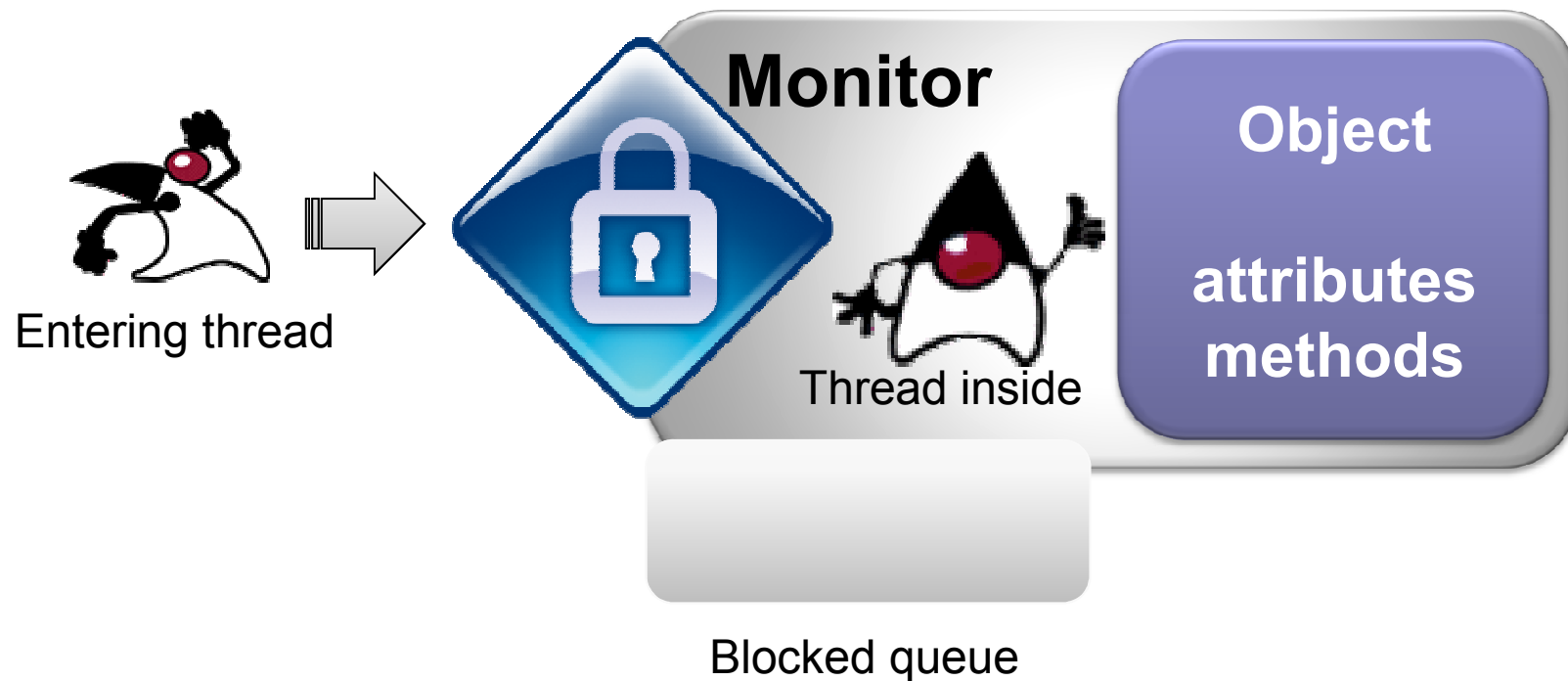
- ## `static boolean holdsLock(Object obj)`

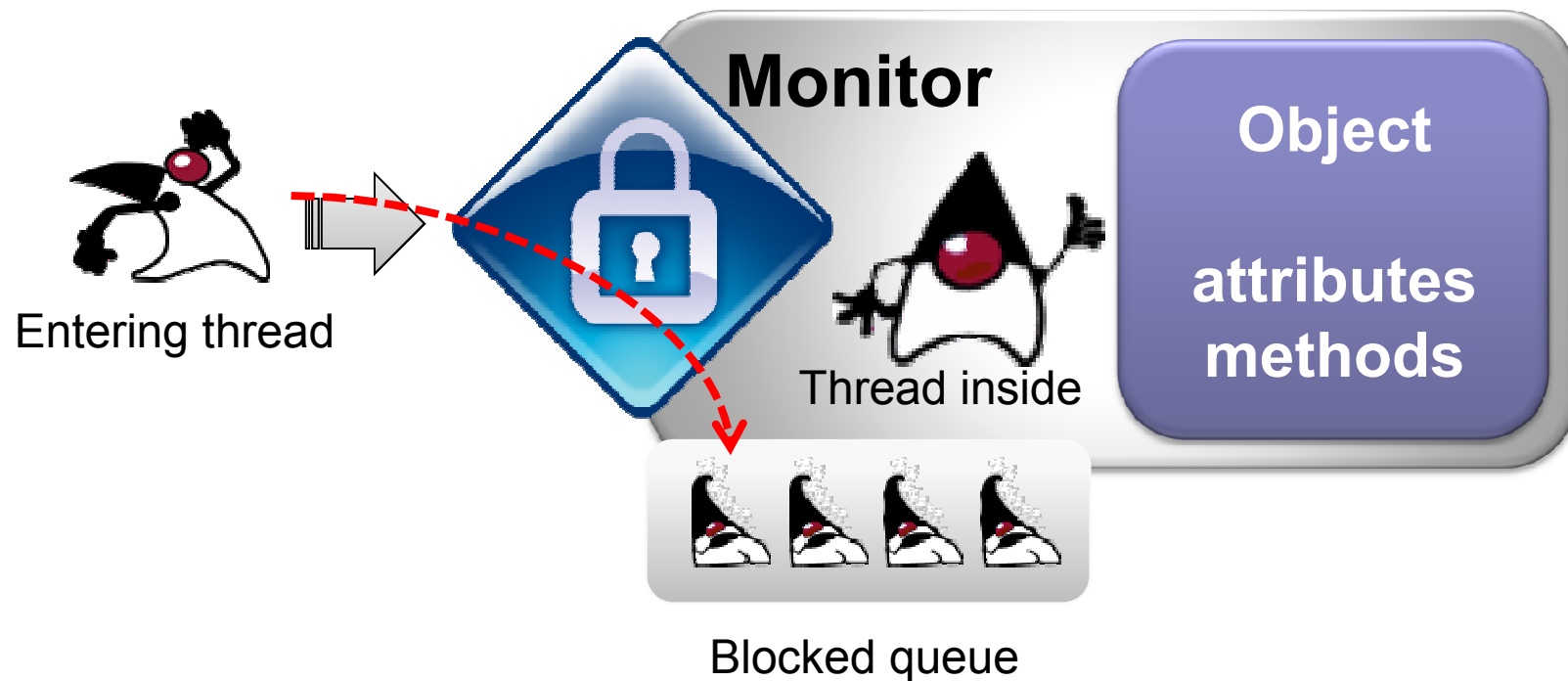 - checks if actual thread is inside monitor of *obj*
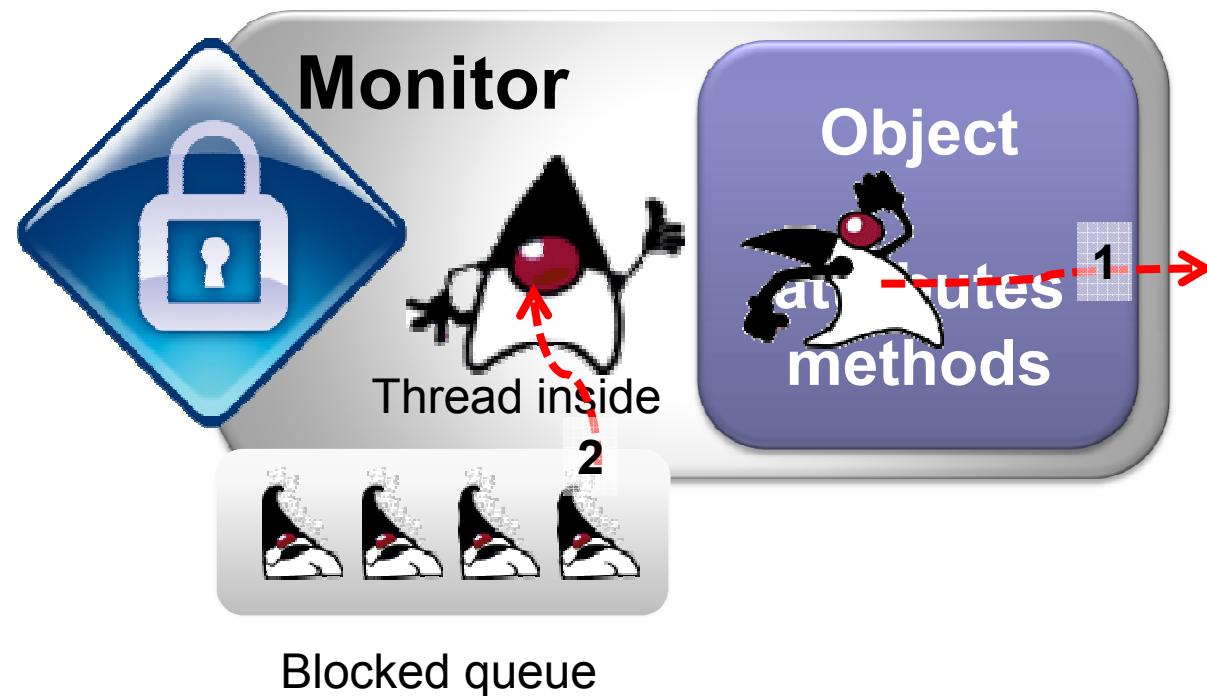
Mutual exclusion: monitor

Objects' monitors explained

Entering thread

Monitor

Object

attributes methods

Thread inside

Blocked queue

No thread in monitor

Objects' monitors explained

Entering thread

Monitor

Thread inside

Object

attributes methods

Blocked queue

Other thread in monitor

Objects' monitors explained

Mutual exclusion

- Monitor boundary: `synchronized`

```
Hashtable<String, Integer> ht = ...;
public void increment(String s) {

    ...

    synchronized (ht) {
        int i = ht.get(s);
        i++;
        ht.put(s,i);
    }

    ...
}
```

Mutual exclusion

- ## `synchronized`

  - □ with a block

    - ■ needs an object reference parameter
    - ■ the parameter specifies the monitor to use

  - □ in front of a method declaration

    - ■ monitor is that of the object whose method is called
    - ■ equivalent to a method wide synchronized block

```
void foo() {
   synchronized(this) {
      ...
   }
}
```
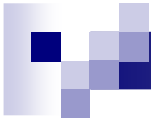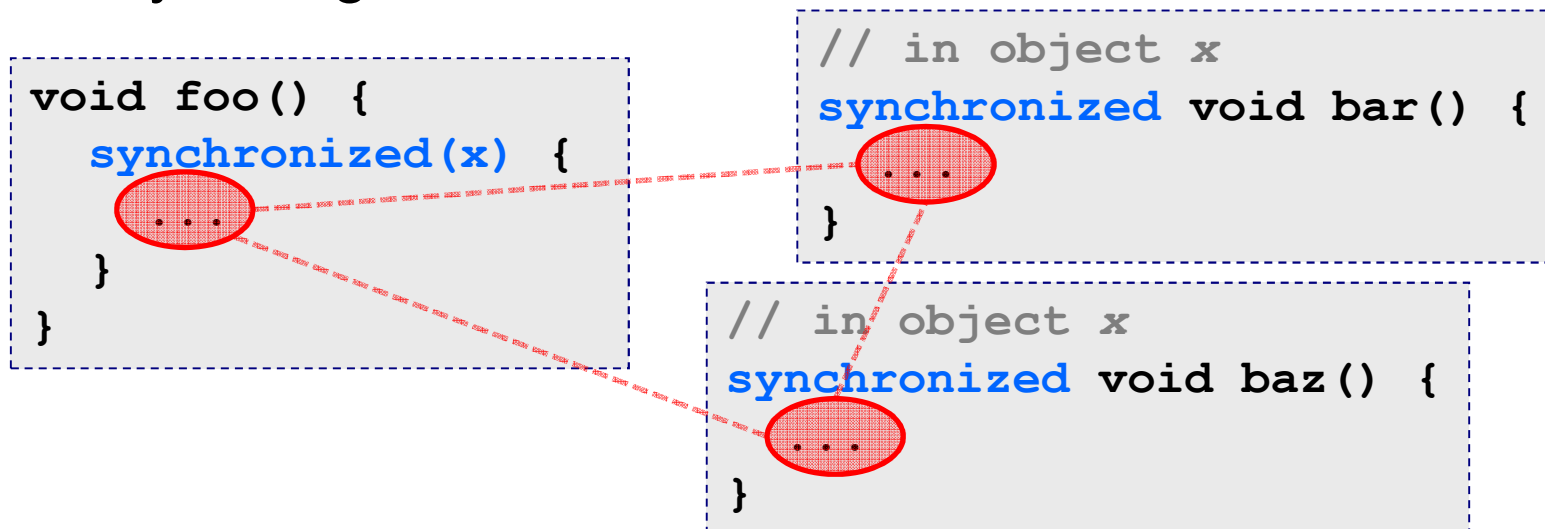
```
synchronized void foo() {

   ...

}
```

Mutual exclusion

- `synchronized` and monitors
    - □ each object has its single monitor
    - □ monitor can be entered from different synchronized blocks/methods
    - □ only a single thread allowed in the monitor!

```
void foo() {
    synchronized(x) {
        . . .
    }
}
```

```
// in object x
synchronized void bar() {
    . . .
}
```

```
// in object x
synchronized void baz() {
    . . .
}
```

Inter-thread communication

- **Monitors**
  - allow threads to share resources
    - no interference
- **Shared memory and resources**
  - allow threads to exchange data
- **Signalling**
  - allows threads to notify each other
    - e.g. about resource availability

Thread signalling

- **`Object.wait([long millis [,int nanos]])`**
  - ☐ if called, the thread will wait for signals for the specified time
  - ☐ thread must be inside the monitor of the object
  - ☐ during wait it leaves the monitor temporarily

```
synchronized (obj) {
    ...
    try {
        obj.wait(); // temporarily leaving monitor
    } catch (InterruptedException ie) {...}
    ...
}
```

Thread signalling

- **`Object.notify()`**
  - ☐ notifies a thread that waits on the object
  - ☐ thread must be inside the monitor of the object
  - ☐ notified thread enters the monitor queue of the object
   - entering monitor is handled by the monitor
- **`Object.notifyAll()`**
 - ☐ same as above, but notifies all waiting threads

```
synchronized (obj) {
    obj.notify(); // wakes a waiting thread
}
```

Threads' notification explained

1. thread calls *wait()* on object
2. other thread *enters* monitor
3. other thread calls *notify()* on object
4. other thread *leaves* monitor
5. a blocked thread *enters*

Monitor

4

1a

3a

2

5

1b

Object

attributes methods

3b

3c

Blocked queue

Waiting queue
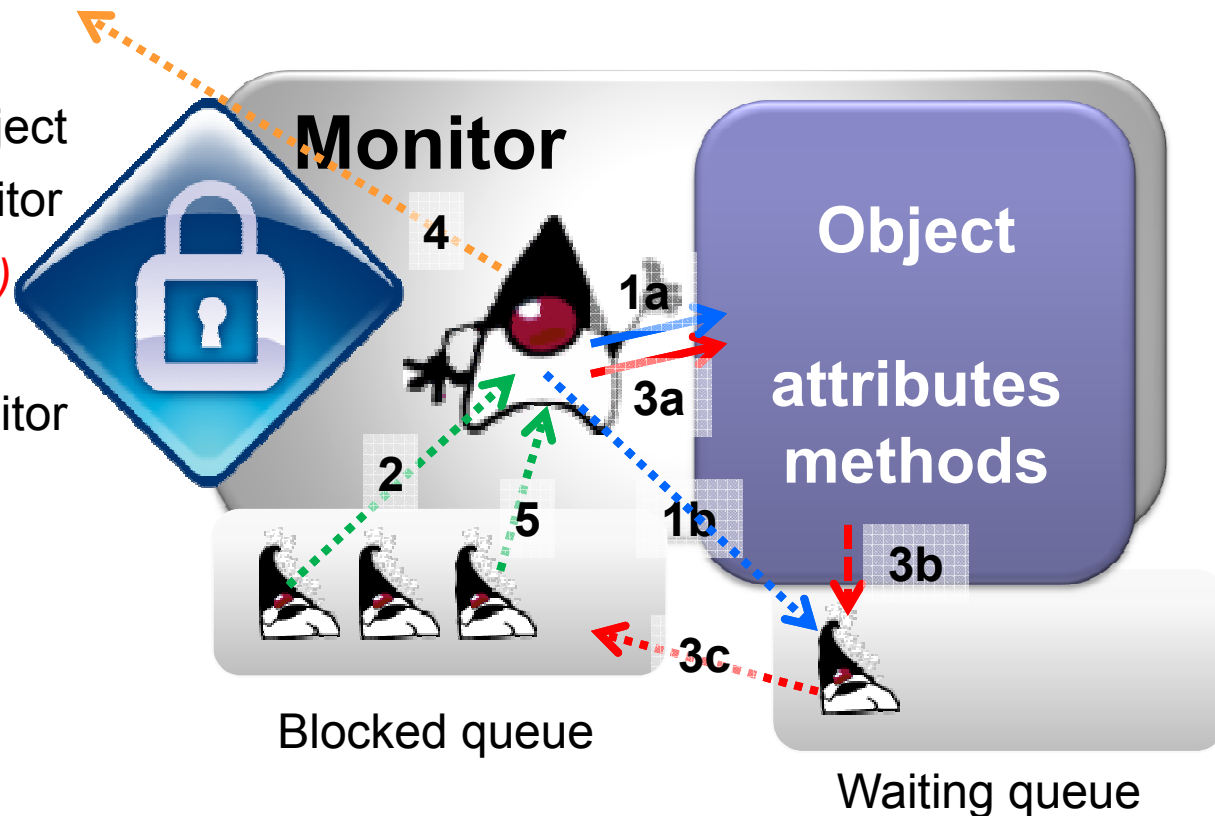
Monitors and wait-notify
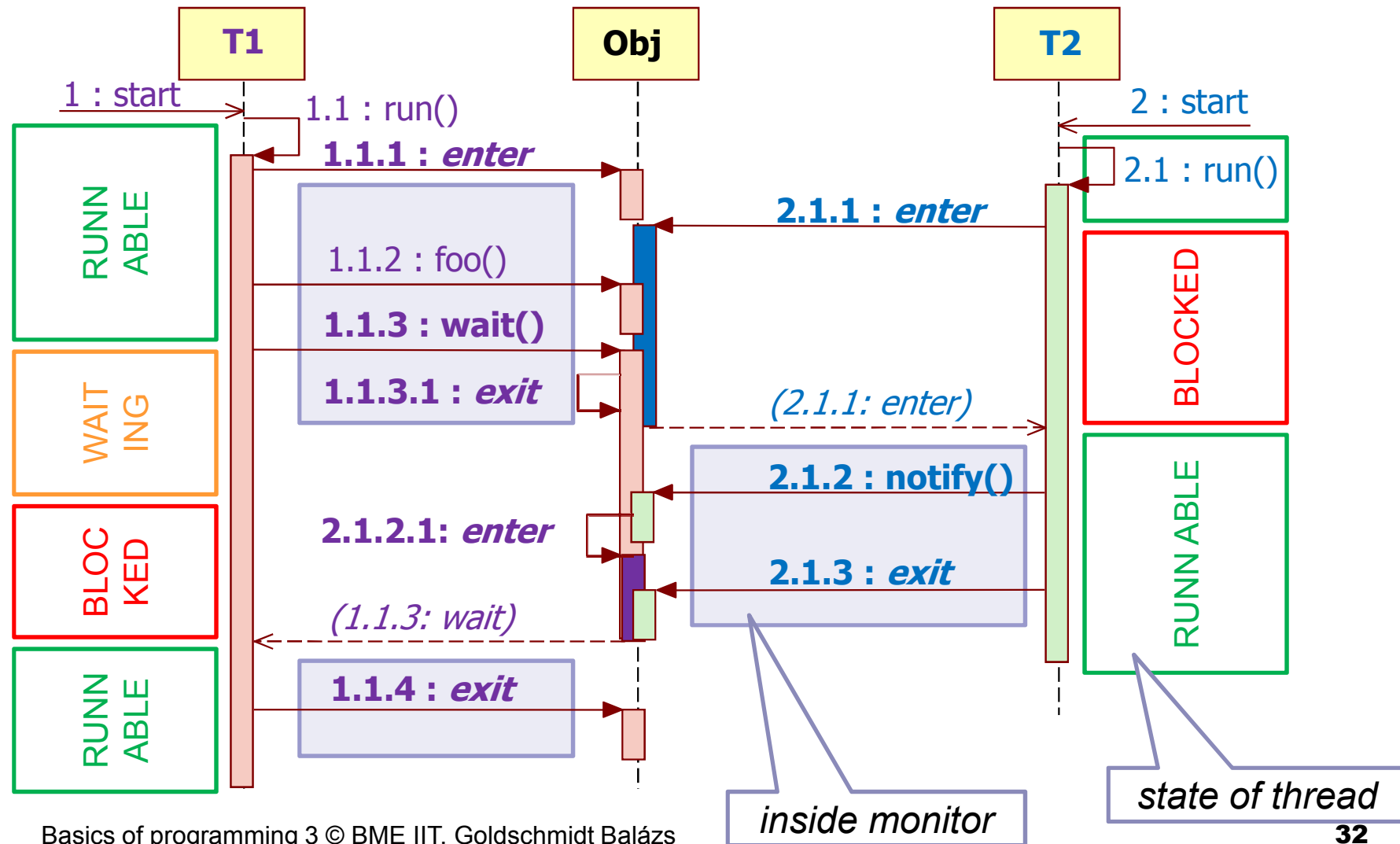
States of the threads

- **`NEW`**
  - ☐ newly created, not yet started
- **`RUNNABLE`** *`(+running)`*
  - ☐ runs or is able to run (already started)
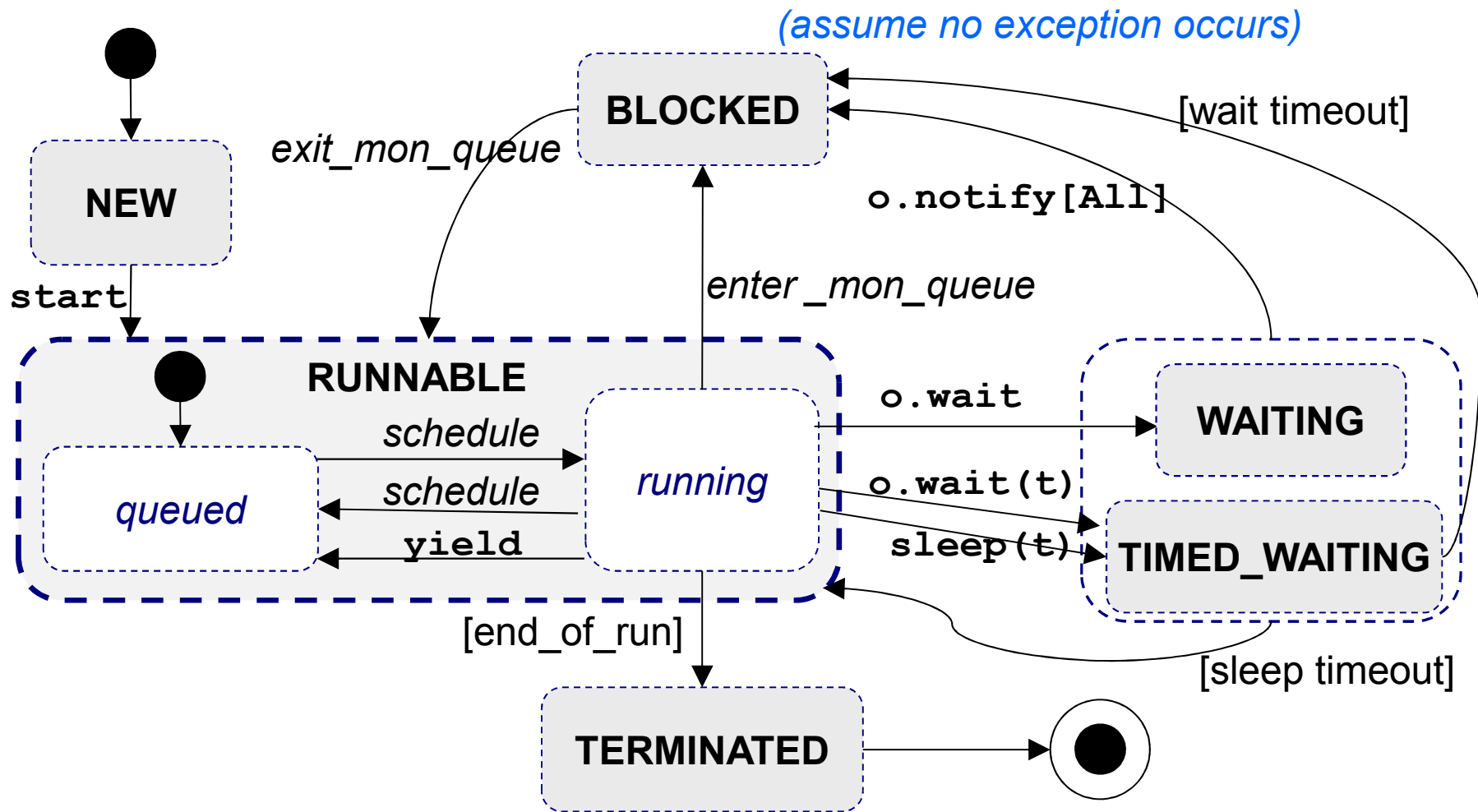- **`BLOCKED`**
  - ☐ waits for a monitor
- **`WAITING, TIMED_WAITING`**
  - ☐ waiting thread (*`Object.wait, Thread.sleep`*)
- **`TERMINATED`**
  - ☐ stopped, can not be restarted

Thread state diagram
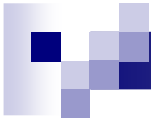
Thread-safe collections

- **Wrapper classes**
  - ☐ Fabricated by class *Collections*
    - ▪ `public static <T> Collection<T> synchronizedCollection(Collection<T> c)`
   - ▪ also for List, Set, SortedSet, Map, SortedMap
  - ☐ Backed by original collection
    - ▪ Stands between client (caller) and collection
    - ▪ Underlying collection is modified, accessed, etc
   - ▪ Makes calls synchronized

Thread-safe collections

- **Genuine thread-safe collection**
  - ☐ In package *java.util.concurrent*
  - ☐ *ConcurrentHashMap*
    - Thread-safe *Map* implementation
  - ☐ *CopyOnWriteArrayList/Set*
    - Modification creates new array
      - → modification are costly
    - Iterators are independent, but can not modify
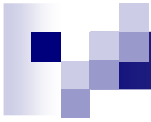
Volatile

- **Thread data is cached**
  - ☐ Cache might be out-of-date
    - ■ Update e.g. before/after synch block
    - → attributes might differ in different caches
  - ☐ Keyword *volatile*
    - ■ makes attributes's access atomic, synchronized
    - ■ Useful for 2-word types as well (eg. *double*, *long*)
      - ☐ Makes data fetch atomic
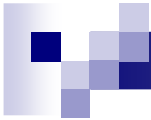
Further thread features

- Interruption
  - □ *InterruptedException*, etc
- Per-thread data
  - □ *ThreadLocal<T>*
- Threadpools
  - □ *Callable* and *ExecutorService*
- Timed starts
  - □ *TimerTask*