

Lästips
Java Network Programming
Chapter 3 "Threads"

DA343A

Objektorienterad programutveckling, trådar och datakommunikation

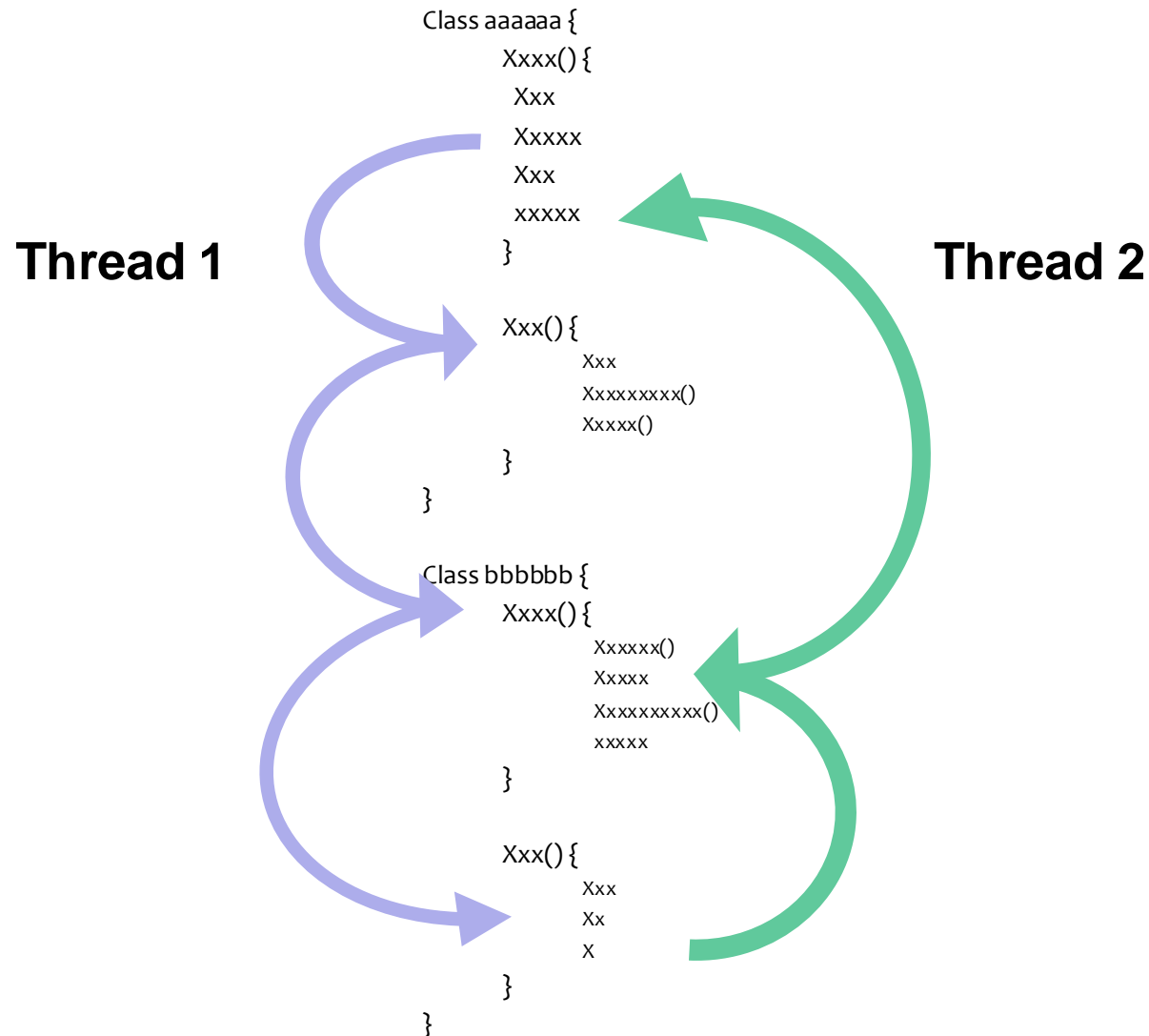
Föreläsning 8 – Callbacks
(& PropertyChangeListener)

Ben Blamey

Summary

- Threads – Revision
- Callbacks
 - Synchronous Callback
 - Synchronous Callback with Many Listeners
 - Synchronous Callback on Worker Thread
 - Asynchronous Callback, across Threads
- Java Beans
- PropertyChangeListener

Threads: Revision



1. Threads use the same code (classes)
2. Threads can start in different places.
3. Threads execute the code independently.
4. Execution is concurrent (at the same time).
5. Can even execute the same code at the same time (independently)!
6. A thread can be "blocked" (i.e. waiting) – other threads can continue.
7. We want to keep the UI Thread running, so the GUI is responsive to mouse/keyboard events.
8. Threads progress independently – nothing keeps them in "sync" – unless we programme it.
9. Have their own variables ("the stack").
10. Objects are shared between threads (on the "heap"), we can pass references to objects between threads.

Threads: Revision

```
Runnable runnable = new SampleRunnable();  
Thread t = new Thread(runnable);  
t.start();
```

- Thread Class
 - <https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html>
- Handling of parallel activities
 - Other threads can continue, if one thread blocked by IO.
 - Computation in parallel.
- Part of a process (program being executed)
 - Threads share the “heap” (objects), but have their own “stack” (variables).
 - JVM manages multi-threading, not the operating system
- Implementation in Java:
 - Inherit from "Thread" (also possible with inner class)
 - Or, Implementation of "Runnable" (overriding the run() method).



Example: Thread as a 'worker'

- A common use of a thread is that it has a task and reports as the work progresses and/or when it is completed.
- Example - ZipArchiver class, can:
 - start a thread that creates a zip file of a folder or file (inner class Zip).
 - start a thread that unzips a zip file (inner class Unzip).
- Summary:
 - The constructor of the class receives a file object as an argument (a file or a directory).
 - The thread is started and the compression/unpacking starts (messages in the Output window).
 - The result is placed in the directory containing the directory/file to be compressed/unzipped
 - Problem: we have no notifications of task progress, error or completion.
- To create/unzip the zip file, use a ZipOutputStream/ZipInputStream

Getting Results from a Thread

- How do we know when work is done, and the result (e.g. zip archive) is ready?
- *Run() and Start() return no values (return type void)*
- One Solution: **Polling**
 - "getter" method with "flag" value
 - Endless loop that queries the return value at intervals
- Disadvantage:
 - Wastes CPU resources.
 - Can't continue immediately – need to wait for next polling check.
 - Parent thread has to do polling.
 - Parent thread has other work to do... in this case, process UI events .

```
while (archive.getStatus() == -1) {  
    ...  
    Thread.sleep(1000);  
}
```

One Solution...

- We use a flag or status variable to indicate completion, progress.
- This is a form of **shared state**.
- To use shared state across threads, we need synchronization!
 - E.g. the notify()/wait() methods in combination with the synchronized keyword, and a condition.
- In this case, this isn't the right solution, as we don't want to block the UI thread.
- We will look at thread synchronization later in the course...
- **We'll return to the zip example later in the lecture!**

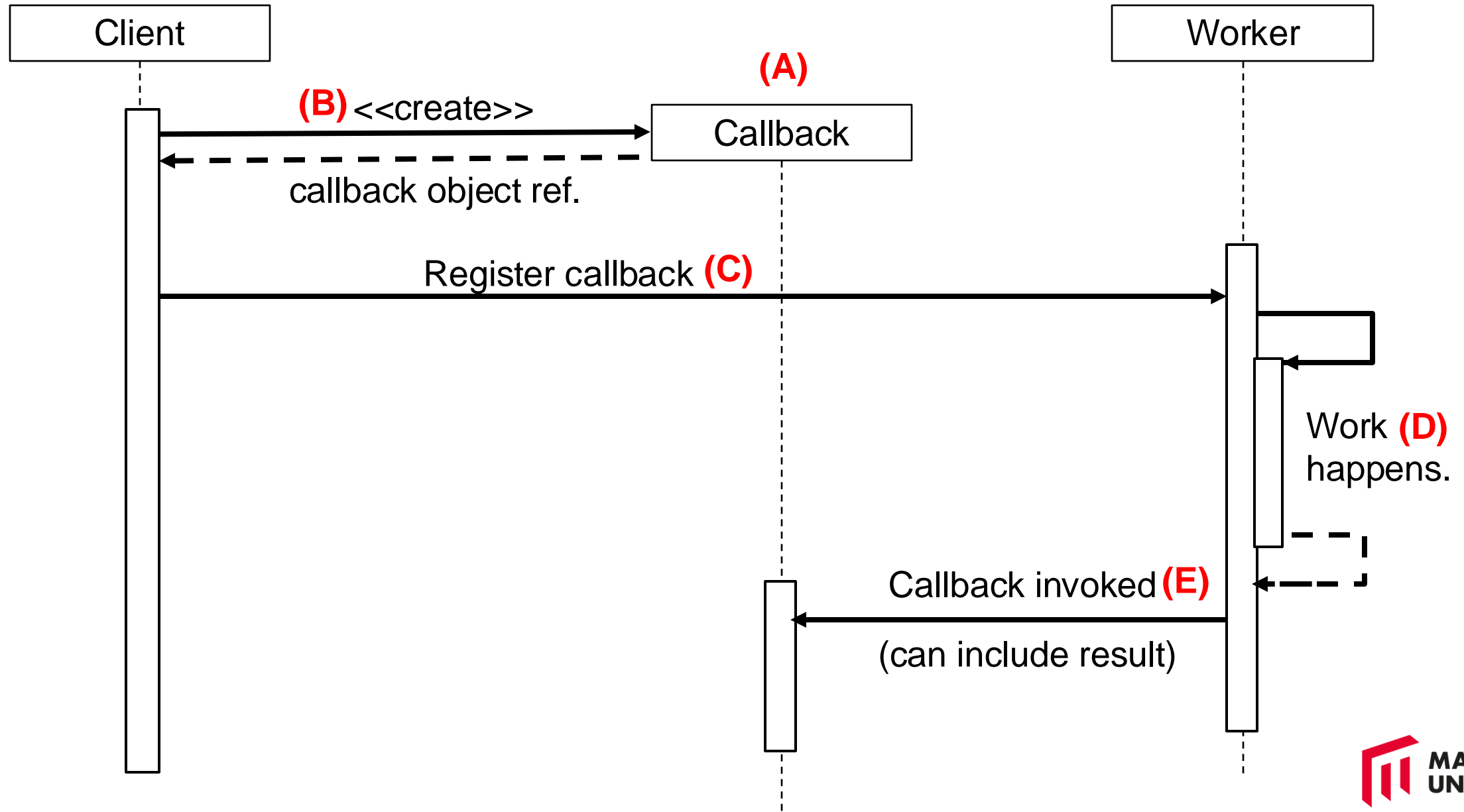
CALLBACKS



Callbacks

- An alternative to is to use **callbacks**.
- A callback a **method** executed after an event.
- Callbacks notify other objects of an event (e.g. completion).
- Callback can include event data e.g. result, error, progress etc.
- We can use callbacks **synchronously** or **asynchronously**
- We can use callbacks **with** or **without** threads.
- Sometimes we have **multiple callback** methods (e.g. to indicate success, error, progress).
- **Steps:**
 - A. Define a callback interface, and a class implementing it.
 - B. Create an instance of the callback class.
 - C. Register the callback instance in the object performing the callback (e.g. a worker)
 - D. Worker does the work!
 - E. Worker invokes callback method.

Synchronous Callback



Synchronous Callback

We start with a single, synchronous callback, with a single thread:

```
public interface CallbackInterface {  
    void myCallbackMethod();  
}  
  
public class Callback implements CallbackInterface {  
    @Override  
    public void myCallbackMethod() {  
        System.out.println(this.toString() + " – callback!");  
    }  
}
```

(A)

```
public class Worker {  
  
    private CallbackInterface callback;  
  
    public void registerCallback(CallbackInterface callback) {  
        this.callback = callback;  
    }  
  
    public void run() {  
        // Do some work... (D)  
  
        if (this.callback != null)  
            callback.myCallbackMethod(); (E)  
    }  
}
```

```
public class Client {  
    public static void main(String[] args) {  
        Worker worker = new Worker();  
        Callback callback = new Callback(); (B)  
  
        worker.registerCallback(callback); (C)  
        worker.run();  
    }  
}
```

Synchronous Callback

We start with a single, synchronous callback, with a single thread:

```
public interface CallbackInterface {  
    void myCallbackMethod();  
}  
  
public class Callback implements CallbackInterface {  
    @Override  
    public void myCallbackMethod() {  
        System.out.println(this.toString() + " – callback!");  
    }  
}
```

(A) - Define a callback interface, and a class implementing it.

```
public class Client {  
    public static void main(String[] args) {  
        Worker worker = new Worker();  
        Callback callback = new Callback(); (B)  
  
        worker.registerCallback(callback); (C)  
        worker.run();  
    }  
}
```

```
private CallbackInterface callback;  
  
public void registerCallback(Callback callback) {  
    this.callback = callback;  
}  
  
public void run() {  
    // Do some work... (D)  
  
    if (this.callback != null)  
        callback.myCallbackMethod(); (E)  
}
```


Synchronous Callback

We start with a single, synchronous callback, with a single thread:

```
public interface CallbackInterface {  
    void myCallbackMethod();  
}  
  
public class Callback implements CallbackInterface {  
    @Override  
    public void myCallbackMethod() {  
        System.out.println(this.toString() + " – callback!");  
    }  
}
```

(A)

```
public class Client {  
    public static void main(String[] args) {  
        Worker worker = new Worker();  
        Callback callback = new Callback(); (B) – Create instance  
                                         of callback class  
  
        worker.registerCallback(callback); (C)  
        worker.run();  
    }  
}
```

```
public class Worker {  
  
    private CallbackInterface callback;  
  
    public void registerCallback(Callback callback) {  
        this.callback = callback;  
    }  
  
    public void run() {  
        // Do some work... (D)  
  
        if (this.callback != null)  
            callback.myCallbackMethod(); (E)  
    }  
}
```

Synchronous Callback

We start with a single, synchronous callback, with a single thread:

```
public interface CallbackInterface {  
    void myCallbackMethod();  
}  
  
public class Callback implements CallbackInterface {  
    @Override  
    public void myCallbackMethod() {  
        System.out.println(this.toString() + " – callback!");  
    }  
}
```

(A)

```
public class Client {  
    public static void main(String[] args) {  
        Worker worker = new Worker();  
        Callback callback = new Callback(); (B)  
  
        worker.registerCallback(callback); (C) – register callback  
        worker.run();  
    }  
}
```

```
public class Worker {  
  
    private CallbackInterface callback;  
  
    public void registerCallback(Callback callback) {  
        this.callback = callback;  
    }  
  
    public void run() {  
        // Do some work... (D)  
  
        if (this.callback != null)  
            callback.myCallbackMethod(); (E)  
    }  
}
```

Synchronous Callback

We start with a single, synchronous callback, with a single thread:

```
public interface CallbackInterface {  
    void myCallbackMethod();  
}  
  
public class Callback implements CallbackInterface {  
    @Override  
    public void myCallbackMethod() {  
        System.out.println(this.toString() + " – callback!");  
    }  
}
```

(A)

```
public class Client {  
    public static void main(String[] args) {  
        Worker worker = new Worker();  
        Callback callback = new Callback(); (B)  
  
        worker.registerCallback(callback); (C)  
        worker.run();  
    }  
}
```

```
public class Worker {  
  
    private CallbackInterface callback;  
  
    public void registerCallback(Callback callback) {  
        this.callback = callback;  
    }  
  
    public void run() {  
        // Do some work... (D) – work happens!  
  
        if (this.callback != null)  
            callback.myCallbackMethod(); (E)  
    }  
}
```


Synchronous Callback

We start with a single, synchronous callback, with a single thread:

```
public interface CallbackInterface {  
    void myCallbackMethod();  
}  
  
public class Callback implements CallbackInterface {  
    @Override  
    public void myCallbackMethod() {  
        System.out.println(this.toString() + " – callback!");  
    }  
}
```

(A)

```
public class Client {  
    public static void main(String[] args) {  
        Worker worker = new Worker();  
        Callback callback = new Callback(); (B)  
  
        worker.registerCallback(callback); (C)  
        worker.run();  
    }  
}
```

```
public class Worker {  
  
    private CallbackInterface callback;  
  
    public void registerCallback(Callback callback) {  
        this.callback = callback;  
    }  
  
    public void run() {  
        // Do some work... (D)  
  
        if (this.callback != null)  
            callback.myCallbackMethod(); (E) – callback invoked  
    }  
}
```

Synchronous Callback with Many Listeners

In this example:

- List of callbacks.
- As before, everything still on the same thread.
- As before, callback is invoked synchronously (like regular method calls).
- Remember – our original main method does not continue until all callbacks have been processed.

```
public class MultipleCallbacks {  
    List<CallbackInterface> listOfCallbacks = new ArrayList<>();  
  
    public void registerCallback(CallbackInterface callback) {  
        listOfCallbacks.add(callback);  
    }  
  
    public void invokeCallbacks() {  
        for (CallbackInterface callback : listOfCallbacks) {  
            callback.myCallbackMethod();  
        }  
    }  
  
    public static void main(String[] args) {  
        MultipleCallbacks mc = new MultipleCallbacks();  
  
        Callback callback1 = new Callback();  
        mc.registerCallback(callback1);  
  
        Callback callback2 = new Callback();  
        mc.registerCallback(callback2);  
  
        // Do some work.  
  
        mc.invokeCallbacks();  
    }  
}
```

Break!

Synchronous Callback on Worker Thread

ex3_callbackexamples_workerthread

- We can use a worker thread to do work in the background (allowing the main thread to continue), and invoke the callbacks synchronously, from the worker thread.
- The worker threads can invoke blocking IO methods (networks and disks), and wait (block) for slow tasks.
- Other threads can continue their work, such as processing UI events.
- A typical application may have many worker threads.

```
public void doWork(){
    new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                System.out.println("Performing a task ...");
                Thread.sleep(1000);
            } catch (InterruptedException e) {...}

            for(CallbackInterface callback : callbacks) {
                callback.myCallbackMethod();
            }
        }
    }).start();
}
```

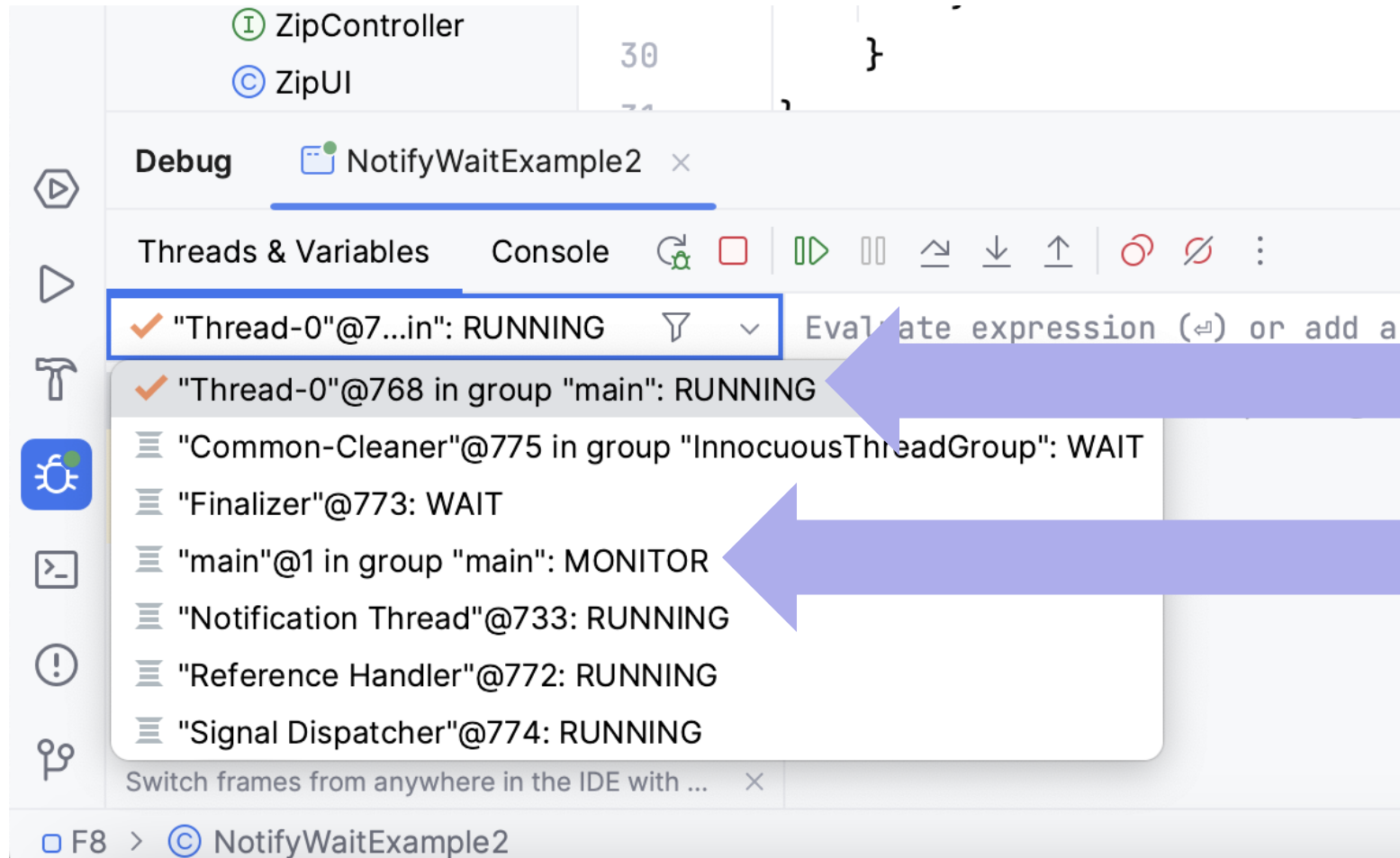
Synchronous Callback on Worker Thread

- Again, Callbacks are executed on the same thread synchronously (i.e. **immediately**), like a regular method.
- This may be the thread used to instantiate our callback object.
- We cannot interact with GUIs on worker threads.
- It is easy to forget which thread is actually running – this is a common source of bugs,
We can print it out:

```
System.out.println("We are on thread " + Thread.currentThread().getName());
```

Debugging and Multiple Threads

We can pause our application with a breakpoint, and use the IDE to see what code is being executed in each thread....



The screenshot shows the 'Debug' console of an IDE with the application 'NotifyWaitExample2' paused. The 'Threads & Variables' tab is active, displaying a list of threads. The first thread, '"Thread-0"@7...in": RUNNING', is highlighted with a blue box. A blue arrow points from the text 'Worker Thread' to this thread. The second thread, '"Thread-0"@768 in group "main": RUNNING', is also highlighted with a blue box. A blue arrow points from the text 'Main Thread' to this thread. The list of threads includes:

- ✓ "Thread-0"@7...in": RUNNING
- ✓ "Thread-0"@768 in group "main": RUNNING
- ≡ "Common-Cleaner"@775 in group "InnocuousThreadGroup": WAIT
- ≡ "Finalizer"@773: WAIT
- ≡ "main"@1 in group "main": MONITOR
- ≡ "Notification Thread"@733: RUNNING
- ≡ "Reference Handler"@772: RUNNING
- ≡ "Signal Dispatcher"@774: RUNNING

The bottom of the console shows the current frame: F8 > © NotifyWaitExample2.

Synchronous Callback on Worker Thread

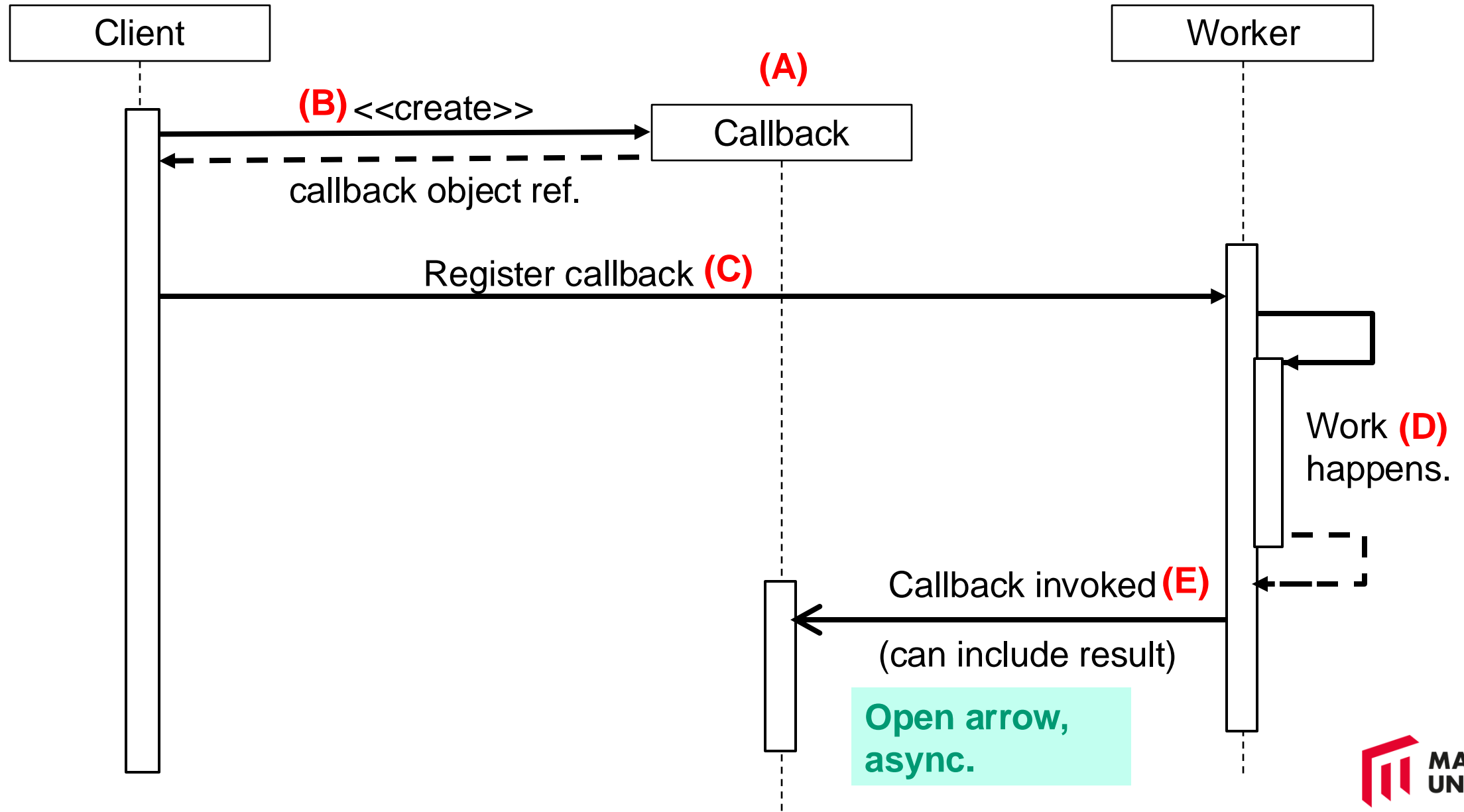
However...

- We **cannot interact with the GUI** on a worker thread (we must use the ‘UI thread’).
- Nor can we update **shared state**, or any other objects which are not **thread safe** – without some form of **synchronization!**
- Another option is to invoke the callback on the original thread (e.g. the ‘main thread’).

Can we just “invoke” a method on another thread? Sometimes...

- In general, we choose a thread’s starting point, but we **can’t arbitrarily invoke methods**.
- But some applications, like GUI applications, work in a different way:
 - Events from the GUI – like mouse clicks, and key presses, need to be processed on the GUI thread.
 - These events wait in a queue to be processed, one at a time.
 - The GUI thread is idle if there are no events.
 - If the queue gets ‘stuck’ on a slow task, the application feels like it is unresponsive (the user thinks it has crashed). So we put slow tasks on worker threads.
 - This is unlike the sequential programming model we are used to!
- If we want to invoke the callback on the GUI thread, we need to use a ‘hook’ to add our own task to the queue. It is executed **later on** – i.e. **asynchronously**. We can do this from other threads.
- In a SWING application, we can add a task (a runnable) to the queue like this:
 `SwingUtilities.invokeLater(...)`
See: <https://docs.oracle.com/javase/8/docs/api/javax/swing/SwingUtilities.html>
- In general, threads don’t have this functionality. So we need to use a buffer or shared state.
- This is just one example of asynchronous programming. It is used in other contexts, like web services.

Asynchronous Callback



Callbacks with Values

ex4_zip_with_callbacks

- Let's return to our original Zip example.
- When the callback method is invoked, we can include additional data as arguments.

```
public interface ZipArchiverCallbacks {  
    void progress(String filename);  
    void ready(String archive);  
    void exception(String message);  
}
```

The class to be observed (ZipArchiver_WithCallbacks) receives one or more instances of classes that implement ZipArchiverCallbacks. It is the observing class that implements the ZipArchiverCallbacks interface. In this case, that's the GUI (ZipUIWithCallbacks).

Now, with callbacks, we can update our GUI with progress on the task!

JAVA BEANS

Java Beans

- Java Beans are just classes which follow conventions:
 - Protect variables against unintentional changes (private)
 - Access to variables using the respective methods (getters/setters)
 - Serializable so it can be stored in a file or transferred over a network
- See also: POJO – *Plain Old Java Object*.
- We need to know about Java beans, because they are used with a callback mechanism called `PropertyChangeListener`.

Java Bean: Example

- Required GET (accessor) and SET (modifier) method to be able to change properties (variables)
- Example: Bean has property name that can be read and changed
- How do we inform other components of the software that the property was changed?

```
public class Employee implements Serializable {  
    private int id;  
    private String name;  
    public Employee(){}  
    public void setId(int id){this.id=id;}  
    public int getId(){return id;}  
    public void setName(String name){this.name=name;}  
    public String getName(){return name;}  
}
```

PropertyChangeListener

ex5_propertychangelistenerexamples

- A property is part of the objects state, but is accessed through **getter/setter methods**, instead of through an attribute.
- A bound property notifies listeners when its value changes (through callbacks).
- The callback has an argument of type **PropertyChangeEvent**
 - See <https://docs.oracle.com/javase/8/docs/api/java/beans/PropertyChangeEvent.html>
- Listeners register with a **PropertyChangeListener** callback interface to be able to receive events:

```
void propertyChange(PropertyChangeEvent evt)
```

- The GUI Widgets in Java Swing implement PropertyChangeListener – e.g. can listen to edits to a text box.
- The PropertyChangeSupport is a helper class used by the bean to manage listeners.

Questions?