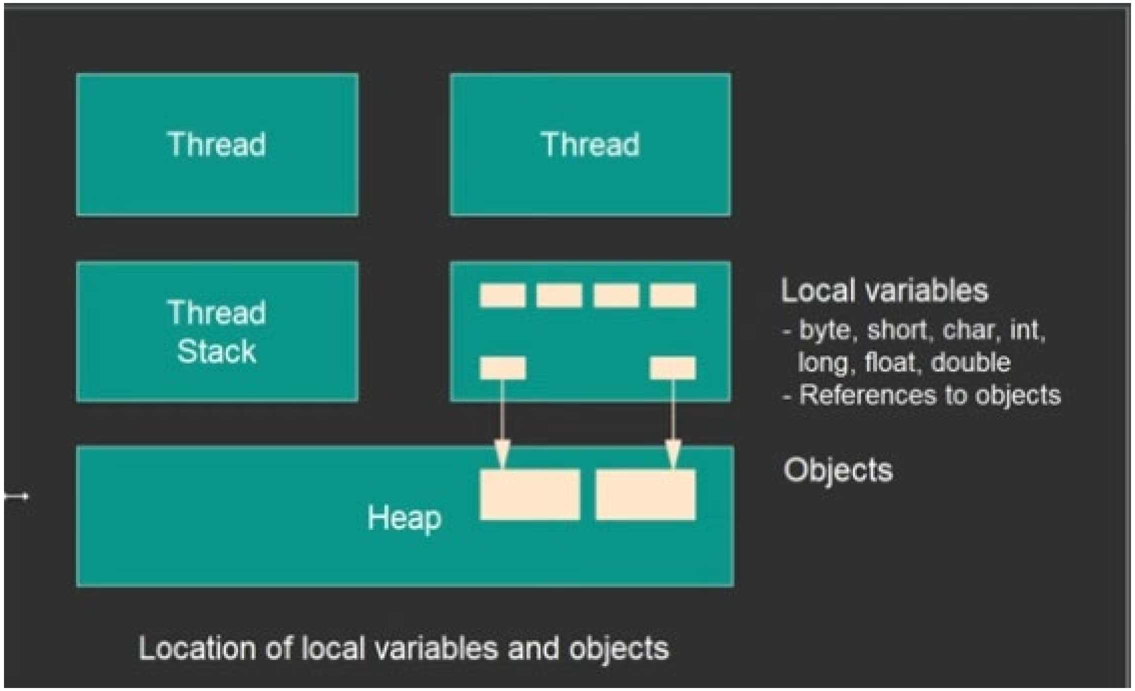
LearnQuest

Presents

Concurrency II

Java Thread Memory

- Java threads have their own stacks but share heap memory
  - ✓ For objects on the heap, each thread has its own reference to that object on its stack



Thread Objects

- Objects created by a runnable object are only accessible to that object
  - ✓ The object is created on the shared heap
  - ✓ But only the thread's stack has a reference to it
  - ✓ No other threads can access the object because they don't have a reference to it.
- Consider the following test object

```java
public class MyObject {
    String name = null;

    public MyObject(String s) {
        this.name = s;
    }

}
```

Thread Objects

- Each Runnable object creates its own copy

```
public class Task implements Runnable {

    String name = null;
    MyObject obj = new MyObject(this.name);

    public Task(String n) {
        this.name = n;
    }

    @Override
    public void run() {
        System.out.println("Thread "+ this.name + " using object " + obj);
    }

}
```

Thread Objects

- Running two threads shows the addresses of the MyObject objects are different

```
public class Runner {

    public static void main(String[] args) {
    Thread t1 = new Thread(new Task("Alpha"));
    Thread t2 = new Thread(new Task("Beta"));
    t1.start();
    t2.start();
    }
```

```
nsole
nated> Runner (8) [Java Application] C:\tools\java\Java18\bin\javaw.exe (Jul. 9, 2022, 11:17:24 a.m. – 11:17:24
Thread Alpha using MyObject Unshared.MyObject@b8bf527
Thread Beta using MyObject Unshared.MyObject@3d00d211
```

Shared Object

- If the object exists prior to the creation of the runnable object
  - ✓ Then the runnable object can be passed a reference to the object
  - ✓ Then all references in the threads resolve to the same object
  - ✓ Even though each thread has its own copy of the reference, they still share the object their references point to.

Shared Object

```java
public class SharedTask  implements Runnable{

        String name = null;
        MyObject myObj = null;

        public SharedTask(String n, MyObject obj) {
            this.name = n;
            this.myObj = obj;
        }

        @Override
        public void run() {
            System.out.println("Thread "+ this.name + " using object " + this.myObj);
        }
```

Shared Objects

- Each runnable object gets a reference to an existing object

```java
public class SharedRunner {

    public static void main(String[] args) {
        MyObject singleton = new MyObject("shared object");

        Thread t1 = new Thread(new SharedTask("one",singleton));
        Thread t2 = new Thread(new SharedTask("two",singleton));
        t1.start();
        t2.start();
```

Problems | Javadoc | Declaration | Console

<terminated> SharedRunner [Java Application] C:\tools\java\jdk-17.0.2\bin\javaw.exe (Feb. 27, 2022, 2:01:16 p.m. – 2:01:16 p.m.)

```
Thread two using object threads.MyObject@e0c18a6 shared object
Thread one using object threads.MyObject@e0c18a6 shared object
```

Race Conditions

- A race condition occurs when more than one thread tries to access a shared resource
  - ✓ Not a problem if all the threads are just reading the same resource
  - ✓ Modifying or writing data in a shared resource may result in corruption
- Result of how the threads are interleaved
  - ✓ Some instructions from one thread may overlap with instructions from and another thread
- The problem is that because race conditions are environment dependent
  - ✓ They are hard to test for
  - ✓ They may be transient (not occur every time)

Race Example

- Suppose we have a shared counter and two threads are trying to update it.

```
int counter = 0;
int newVal = 0

newVa1l = counter + 1;      // Thread one reads the counter and increments
newVal2 = counter + 1;      // Thread two reads and increments
counter =  newVal1          // Thread 1 updates counter
counter = newVal2           // Thread 2 overwrites the value just saved
```

Race Types

- Check-Then-Act race condition
  - ✓ Logic executed by a thread depends on the value of a shred variable
  - ✓ A thread may read a stale value and perform the wrong action
- Read-Modify-Write
  - ✓ This is the example we just saw
  - ✓ Multiple threads modifying the same resource
  - ✓ Because of the interleaving of reads and writes, the value becomes incorrect

Avoiding Race Conditions

- Restrict access to critical sections of code where race conditions could occur
- Java uses the “synchronized” keyword
  - ✓ Ensures that the resource identified uses atomic operations
  - ✓ This means essentially only one thread at a time can access the resource
- When a thread enters a synchronized block all other threads trying to execute that block must wait
  - ✓ Once the thread executes the block, the next thread can have access to the block
- This is illustrated in the lab

Semaphores

- The problem with using the synchronized keyword is that theads block waiting for their turn
  - ✓ This can create slowdowns in processing
  - ✓ The threads are idle while waiting to use the synchronized resource
- A semaphore is an integer variable shared by the threads
  - ✓ Semaphores regulate access to a shared resource
  - ✓ Eliminates busy waiting
  - ✓ Can be binary taking the values 0 and 1
  - ✓ Or can be a count of available resources

Semaphores

- Two operations
- Wait or wait for:
  - ✓ Take a resource and decrements the semaphore
  - ✓ If the semaphore is 0, then the thread sleeps until one is available
- Signal:
  - ✓ Releases a resource and increments the semaphore
  - ✓ If there are sleeping threads, a waiting thread is woken up

Java Semaphore

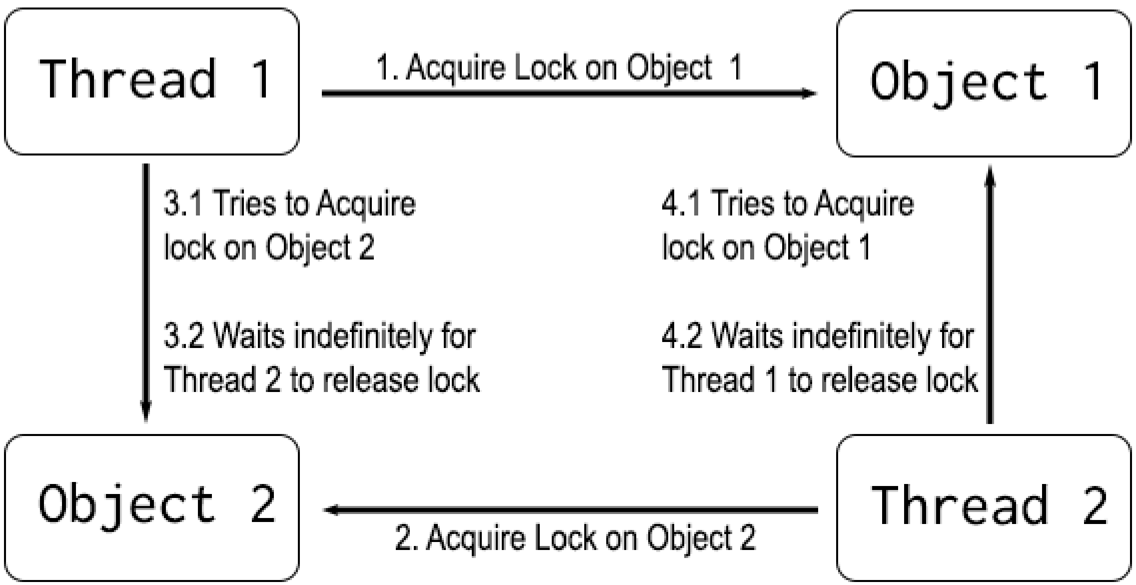
- A counting semaphore.
  - ✓ Maintains a set of permits
  - ✓ Each acquire() blocks if necessary, until a permit is available, and then takes it
  - ✓ Each release() adds a permit, potentially releasing a blocking acquirer.
  - ✓ However, no actual permit objects are used; the Semaphore just keeps a count of the number available and acts accordingly.

Mutex

- Similar to a binary semaphore
  - ✓ However, it is more like a lock in that a thread can take ownership of a mutex
  - ✓ It then has exclusive use of the resource until it releases the mutex
- The synchronized keyword in Java essentially is a mutex
- We can emulate a mutex in Java using a semaphore

LearnQuest

Mutex Deadlock



Atomic Access

- An atomic action happens all at once.
  - ✓ It cannot stop in the middle
  - ✓ it either happens completely, or it doesn't happen at all
  - ✓ No side effects of an atomic action are visible until the action is complete
- In Java
  - ✓ All primitive types use atomic reads and writes except for longs and doubles
  - ✓ All variables marked volatile also use atomic reads and writes

Atomic Variables

- Java provides atomic classes that support atomic operations on single variables
  - ✓ They have get() and set() methods that operate like atomic reads and writes

Atomic Variables Example

- Consider the following Counter class

```
class Counter {
    private int value = 0;

    public void increment() {this.value++;}

    public void decrement() {this.value--;}

    public int value() { return this.value;}

}
```

Atomic Variables Example

- We can use a mutex to control access

```
class SynchCounter {
    private int value = 0;

    public synchronized void increment() {this.value++;}

    public synchronized void decrement() {this.value--;}

    public synchronized int value() { return this.value;}

}
```

Atomic Variables Example

- We can get the same effect using an Atomic variable

```
class AtomicCounter {
    private AtomicInteger value = new AtomicInteger(0);

    public void increment() {this.value.incrementAndGet();}

    public void decrement() {this.value.decrementAndGet();}

    public int value() {return this.value.get();}

}
```

Concurrent Collections

- Memory Consistency Errors
  - ✓ Occur when different threads have inconsistent views of what should be the same data
  - ✓ Managed by a "happens-before" relationship
  - ✓ Guarantee that memory writes by one specific statement are visible to another specific statement
- If multiple threads are accessing a collection
  - ✓ We may have two threads trying to add a "last" object into the same position while another thread is reading the last object
  - ✓ It's not clear what the "last object" will actually to each thread

Concurrent Collections

- BlockingQueue
  - ✓ Defines a first-in-first-out data structure that blocks or times out on attempts to add to a full queue, or retrieve from an empty queue.
- ConcurrentMap
  - ✓ Defines atomic operations for maps.
- There are other more specialized collections
- Concurrency issues are managed by the collection
  - ✓ Recall that the Executor Service uses a BlockingQueue

The Thread Problem

- Threads run asynchronously
  - ✓ The spawning thread does not wait for the child thread to finish
  - ✓ The Runnable run() method returns a void
  - ✓ Where would a return value be returned to?
- Callbacks
  - ✓ One approach is to include a function that is called when the thread is done
  - ✓ Named a “call back” (“Call me when you finish”)
  - ✓ Standard pattern in asynchronous programming

Callback Issues

- Error handling
  - No clean mechanism for handling errors in a callback
  - Also, how does a callback handle an error in the thread?
- Nested callbacks
  - If we have a chain of threads, each of which has a callback
  - Then we have a nested chain of callbacks
  - Can be very difficult to manage at runtime
  - Very had to debug
  - What thread manages the return values from the callback,
- Blackboard pattern
  - Callbacks all update a common data structure

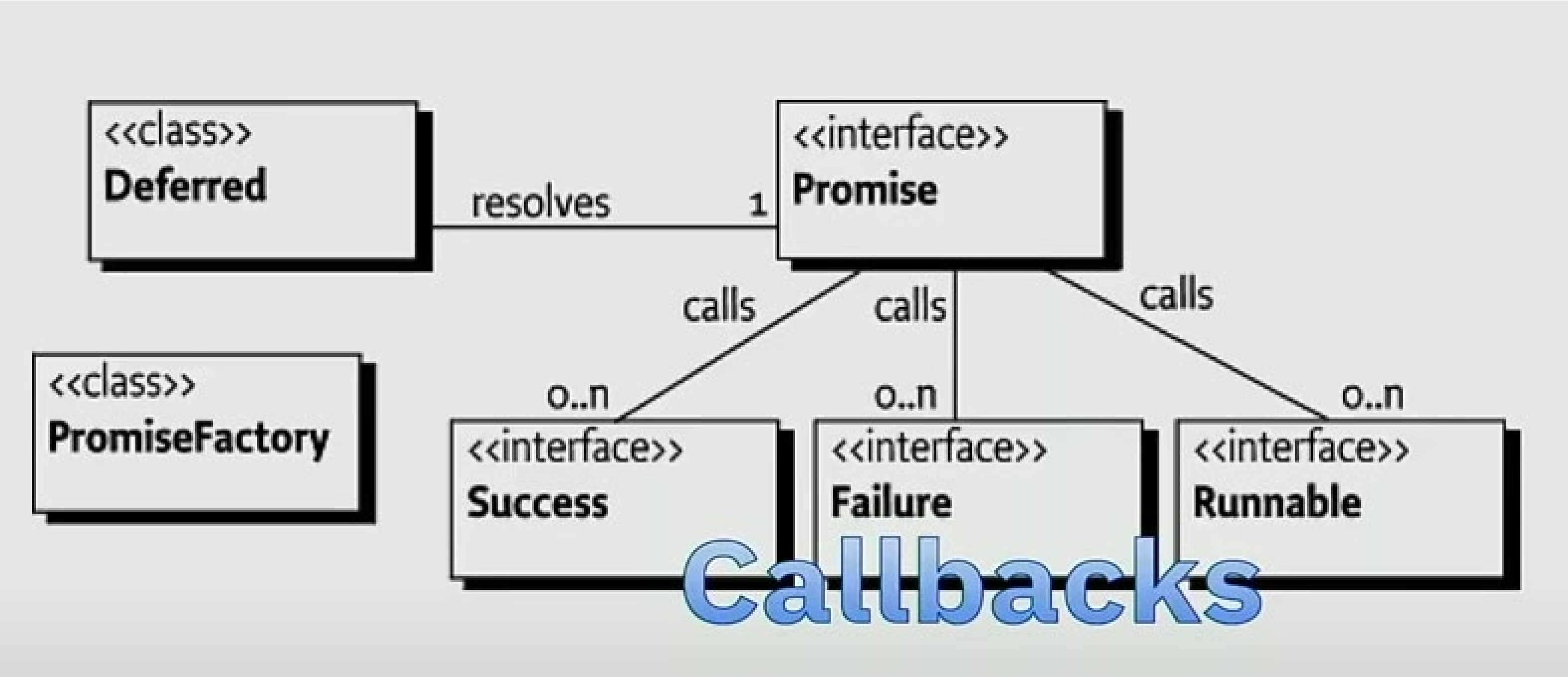
Callbacks in Java

- Callbacks are handled in various languages by passing a function pointer to the callback function
  - ✓ Java doesn’t have function pointers
- Java emulates this by using a callback interface
  - ✓ The callback object implements the callback interface
  - ✓ When the thread starts, it is passed a callback object
  - ✓ When an event that requires a callback happens, the callback method is invoked on the callback object

Promises

- A Promise represents a proxy
- Stands in for for a value not known when the promise is created
- Promises are associated to an asynchronous action's eventual success value or failure
- A promise is a placeholder for the result of an ongoing asynchronous operation
- A promise can be resolved or unresolved

Promises



<<class>>
Deferred
resolves
1
<<interface>>
Promise
calls
calls
calls
<<class>>
PromiseFactory
0..n
0..n
0..n
<<interface>>
Success
<<interface>>
Failure
<<interface>>
Runnable
Callbacks

Promises

- The task creates a deferred object that will be resolved to a promise
  - ✓ A promise to be filled in after the task is completed
- After the task completes the promise is resolved to one of the outcomes in the interface
  - ✓ The outcomes are callbacks, just like we saw in the previous section

Questions