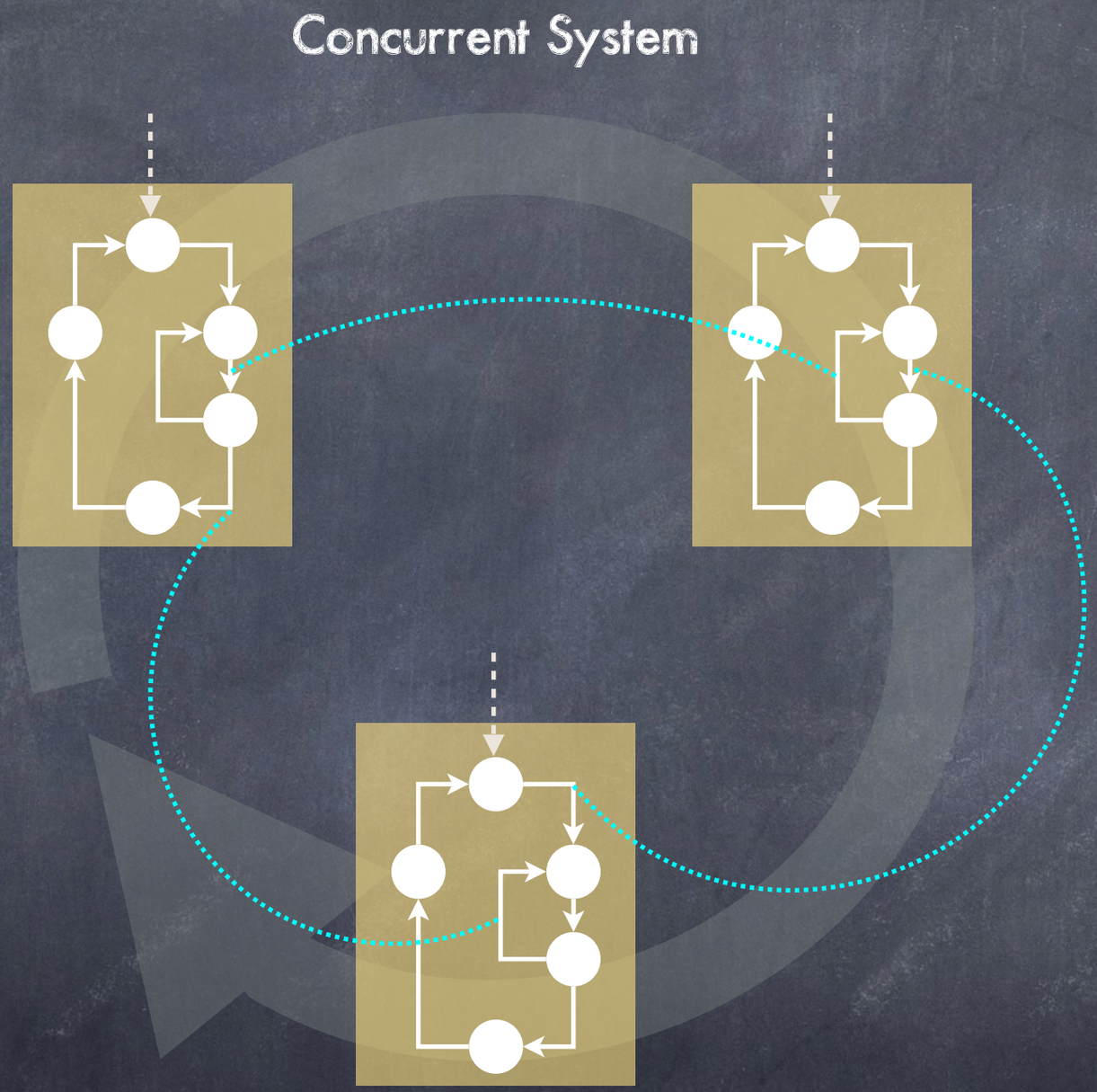
Introduction to Java Threads

SFWRENG 3BB4:

Software Design III — Concurrent System Design

Concurrent Systems in Java



Introduction to Java Threads

What are Threads?

- In Java, threading is a facility to allow multiple activities to coexist within a single program.
- Threads are independent, concurrent paths of execution through a program.
- They have their own stack, program counter, and local variables.

  However, multiple threads within a program share the same memory address space.

  This means that threads interact via variables and objects. While this makes it easy for threads to share information with each other, care must be taken to ensure that they do not interfere with other threads in the same program.
- The Java thread facility and API is relatively simple. Writing programs that use threading effectively is not.

Introduction to Java Threads

What are Threads?

- Every Java program has at least one thread – the main thread.

  When a Java program starts, the JVM creates the main thread and calls the program's `main()` method within that thread.

- The JVM also creates other threads that run concurrently with the `main()` method.

  For example, threads associated with garbage collection, object finalization, and other JVM housekeeping tasks.

Introduction to Java Threads

Example

```
import ...

public class Main
{
    /* CLASS DECLARATIONS */
    ...

    public static void main(String args[])
    {
        /* METHOD DECLARATIONS */
        ...
    }
}
```

The Lifecycle of a Thread

Creating a Thread

Through the **`Thread`** constructor or by instantiating classes that extend the **`Thread`** class.

Example

```
import java.lang.Thread;


public class MyThread extends Thread
{
    /* CLASS DECLARATIONS */
    ...
}


MyThread myThread = new MyThread();
```

The Lifecycle of a Thread

Starting a Thread

- Creating threads and starting threads are not the same
- A thread doesn't actually begin to execute until another thread calls the `start()` method on the Thread object for the new thread.

Example

```
import java.lang.Thread;


public class Main
{
   /* CLASS DECLARATIONS */
   Thread thread = new Thread();


   public static void main(String args[])
   {
      /* METHOD DECLARATIONS */
      thread.start();
   }
}
```

The Lifecycle of a Thread

Ending a Thread

- A thread will end in one of three ways:
  1. The thread comes to the end of its `run()` method.
  2. The thread throws an `Exception` or `Error` that is not caught.
  3. Another thread calls one of the deprecated `stop()` methods.

     Deprecated means they still exist, but you shouldn't use them in new code and should strive to eliminate them in existing code.
- When all the threads within a Java program complete, the program exits.

The Lifecycle of a Thread

Joining with Threads

- The Thread API contains a method for waiting for another thread to complete: the `join()` method.
- When `join()` is called, the calling thread will block until the target thread completes.

Example

```
import java.lang.Thread;


public class Main
{
    /* CLASS DECLARATIONS */
    Thread thread = new Thread();

    public static void main(String args[])
    {
        /* METHOD DECLARATIONS */
        thread.start();
        thread.join();
    }
}
```

The Lifecycle of a Thread

Summary

- Like programs, threads have a life cycle:
    1. They start.
    2. They execute.
    3. They complete.
- A program may contain multiple threads.
- Threads execute independently of each other.
- A thread is created by instantiating a Thread object, or an object that extends Thread.
- A thread doesn't start to execute until the `start()` method is called on the new Thread object.
- Threads end when they come to the end of their `run()` method or throw an `unhandled Exception`.
- The `join()` method can be used to wait until another thread completes.

Questions?