Thread example

```java
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        Thread tt1 = new HelloRunnable();
        tt1.start();

        Thread tt2 = new HelloRunnable();
        tt2.start();
    }
}
```

Thread Interference

```java
public class counter {
    private int count = 0;
    public void increment() {
        count++;
    }
    public void decrement() {
        count--;
    }
    public int value() {
        return count;
    }
}
```

Suppose Thread A invokes `increment()` at about the same time Thread B invokes `decrement()`.

- If the initial value of `count` is 0, their interleaved actions might follow this sequence:
  - Thread A: Retrieve count.
  - Thread B: Retrieve count.
  - Thread A: Increment retrieved value; result is 1.
  - Thread B: Decrement retrieved value; result is -1.
  - Thread A: Store result in `count`; `count` is now 1.
  - Thread B: Store result in `count`; `count` is now -1.

Thread Interference

- Interference happens when two operations, running in different threads, interleave.
- Scheduling of multiple threads is not guaranteed to be fair.
- If a producer thread and a consumer thread are sharing the same kind of data in a program then either producer may produce the data faster or consumer may retrieve an order of data and process it without its existing.

Thread states

![Thread states diagram]
Synchronize threads

- When two or more threads share the same resource (variable or method), only one of them can access the resource at one time.
- Two approaches to synchronize threads

```java
synchronized void myObjectMethod() {
    ...
}
synchronized(Object) {
    ...
}
```

Synchronized methods

// A synchronized method -> mutual exclusion (lock) per object
// A synchronized static method -> mutual exclusion (lock) per class

```java
synchronized void myObjectMethod() {
    
}synchronized static void myClassMethod() {
    
```

Account Example

// This is an example of synchronized methods.
public class Account {
    private double balance;
    public Account(double initialDeposit) {
        balance = initialDeposit;
    }
    public synchronized double getBalance() {
        return balance;
    }
    public synchronized void deposit(double a) {
        balance += a;
    }
}

Synchronized statements

- The synchronized statement:
  - Intrinsic lock: an internal entity associated with an object
  - mutual exclusion (lock) of data access instead of a synchronized method
  - very lightweight

- Syntax:

```java
synchronized(Object) {
    ...
```

- Synchronized methods: Any method is specified with the keyword `synchronized` is only executed by one thread at a time.
- Synchronized statements: Any object is stated as `synchronized` is accessed by only one thread at any given time.
Intrinsic Lock and Synchronization

- Every object has an intrinsic lock associated with it.
- As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.
- When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent operation of the same lock.
- When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method’s object.

Synchronized statements

/* Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock: */
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}

Synchronized statements

// Example (mutual exclusive lock on the variable values):
public static void abs(int[] values) {
    synchronized(values) {
        for (int i=0; i < values.length; i++)
            if (values[i] < 0)
                values[i] = -values[i];
    }
}

Synchronized Statements

// Here we create two objects solely to provide locks:
public class MoreComplex {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();
    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }
    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}

Deadlock

public class Deadlock {
    static class Person {
        private final String name;
        public Person(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
        public synchronized void wave(Person p) {
            System.out.format("%s: %s has waved to me!\n", this.name, p.getName());
            p.waveBack(this);
        }
        public synchronized void waveBack(Person p) {
            System.out.format("%s: %s has waved back to me!\n", this.name, p.getName());
        }
    }
    public static void main(String[] args) {
        final Person alex = new Person("Alex");
        final Person beth = new Person("Beth");
        new Thread(new Runnable() {
            public void run() { alex.wave(beth); }
        }).start();
        new Thread(new Runnable() {
            public void run() { beth.wave(alex); }
        }).start();
    }
}

- When Deadlock runs, it’s extremely likely that both threads will block when they attempt to invoke waveBack.
Starvation & Liveloop

- **Starvation:**
  - Thread unable to gain access to shared resource.
  - Hence unable to make progress.
  - Because shared resources are made unavailable for long periods by "greedy" threads

- **Liveloop:**
  - Thread 1 responds to action of thread 2
  - Thread 2 responds to thread 1
  - Repeat forever!

Thread pools

- **java.util.concurrent** package
- A thread pool has **worker threads**. Worker thread exist separately from the tasks it executes and is often used to execute multiple tasks.
- One common type of thread pool is the **fixed thread pool**. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread.

Multiple thread server

```java
int portNumber = 8250;
ServerSocket s = new ServerSocket(portNumber);
while(true) // accept multiple clients
{
    Socket clientSoc = s.accept();
    Thread t = new ThreadedEchoHandler(clientSoc);
    t.start(); // Handle the client communication
}
```

Executors

- Objects that create and manage threads.
- Executor interfaces:
  - The **Executor** interface
  - The **ExecutorService** interface
  - The **ScheduledExecutorService** interface
- If r is a **Runnable** object, and e is an **Executor** object you can replace
  ```java
  (new Thread(r)).start();
  ```
  with
  ```java
  e.execute(r);
  ```

To create a thread pool

```java
public static ExecutorService newFixedThreadPool(int nThreads) {
    Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue. At any point, at most nThreads threads will be active processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. If any thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.
```

Fork/Join

- The fork/join framework is an implementation of the **ExecutorService** interface that helps you take advantage of multiple processors.
- **Class ForkJoinPool**
- **Usage of ForkJoinPool**
  ```java
  static final ForkJoinPool mainPool = new ForkJoinPool();
  ...
  public void sort(long[] array) {
      mainPool.invoke(new SortTask(array, 0, array.length));
  }
  ```
Summary

• Thread interference
• Synchronization
  – Synchronized method
  – Synchronized statements
• Intrinsic lock
• Executor interfaces
  – Thread pools
  – Fork/Join