Threads

- **Threads** (an abbreviation of threads of control) are how we can get more than one thing to happen at once in a program. A thread has a minimum of internal state and a minimum of allocated resources.

Threads

- **Threads** (an abbreviation of threads of control) are how we can get more than one thing to happen at once in a program. A thread has a minimum of internal state and a minimum of allocated resources.
- A thread (a.k.a. lightweight process) is associated with a particular process (a.k.a. heavyweight process, a retronym). A heavyweight process may have several threads and a thread scheduler. The thread scheduler may be in the user or in the system domain.

Threads

- **Threads** (an abbreviation of threads of control) are how we can get more than one thing to happen at once in a program. A thread has a minimum of internal state and a minimum of allocated resources.
- A thread (a.k.a. lightweight process) is associated with a particular process (a.k.a. heavyweight process, a retronym). A heavyweight process may have several threads and a thread scheduler. The thread scheduler may be in the user or in the system domain.
- Threads share the text, data and heap segments. Each thread has its own stack and status.

Threads

- **Threads** (an abbreviation of threads of control) are how we can get more than one thing to happen at once in a program. A thread has a minimum of internal state and a minimum of allocated resources.
- A thread (a.k.a. lightweight process) is associated with a particular process (a.k.a. heavyweight process, a retronym). A heavyweight process may have several threads and a thread scheduler. The thread scheduler may be in the user or in the system domain.
- Threads share the text, data and heap segments. Each thread has its own stack and status.

Examples where threads are useful: Windowing systems, Web browsers, Servers and Clients

---

How can you be in two place at once when you're not anywhere at all?
–Firesign Theater.

Threads in Java

Threads are part of the core Java language. There are two ways to create a new thread of execution.

Threads in Java

Threads are part of the core Java language. There are two ways to create a new thread of execution.

- One is to the extend the class `java.lang.Thread`. This subclass should override the `run` method of class Thread. An instance of the subclass can then be allocated and started.

Threads in Java

Threads are part of the core Java language. There are two ways to create a new thread of execution.

- One is to the extend the class `java.lang.Thread`. This subclass should override the `run` method of class Thread. An instance of the subclass can then be allocated and started.
- The other way to create a thread is to declare a class that implements the `Runnable` interface. That class then implements the `run` method.

Threads in Java

Threads are part of the core Java language. There are two ways to create a new thread of execution.

- One is to the extend the class `java.lang.Thread`. This subclass should override the `run` method of class Thread. An instance of the subclass can then be allocated and started.
- The other way to create a thread is to declare a class that implements the `Runnable` interface. That class then implements the `run` method. An instance of the class can then be allocated, passed as an argument when creating Thread, and started.

Basic Thread Examples in Java

- Example 1: Shows how to create a thread by subclassing `Thread` class.
  threads/ThreadExample.java

Basic Thread Examples in Java

- Example 1: Shows how to create a thread by subclassing `Thread` class.
  threads/ThreadExample.java
- Example 2: Shows how to create a thread by implementing the `Runnable interface`.
  threads/RunnableExample.java

Basic Thread Examples in Java

- Example 1: Shows how to create a thread by subclassing `Thread` class.
  threads/ThreadExample.java
- Example 2: Shows how to create a thread by implementing the `Runnable interface`.
  threads/RunnableExample.java
- Example 3: Create a thread quagmire!
  threads/MaxThreads.java

In Java, each thread is an object!

Relevant Java Classes/Interfaces

- See documentation for basic classes: `java.lang.Thread`, `java.lang.ThreadGroup` and `java.lang.Runnable` interface.
- See the `java.lang.Object` class for synchronization methods.
- For automatic management of threads, see: `Executor` interface from `java.util.concurrent` package.

Controlling Threads

- `start()` and `stop()`

Controlling Threads

- `start()` and `stop()`
- `suspend()` and `resume()`
- `sleep()`

Controlling Threads

- `start()` and `stop()`
- `suspend()` and `resume()`
- `sleep()`
- `join()`: causes the caller to block until the thread dies or with an argument

Controlling Threads

- `start()` and `stop()`
- `suspend()` and `resume()`
- `sleep()`
- `join()`: causes the caller to block until the thread dies or with an argument
- `interrupt()`: wake up a thread that is sleeping or blocked on a long I/O operation (in millisecs) causes a caller to wait to see if a thread has died

Controlling Threads

- `start()` and `stop()`
- `suspend()` and `resume()`
- `sleep()`
- `join()`: causes the caller to block until the thread dies or with an argument
- `interrupt()`: wake up a thread that is sleeping or blocked on a long I/O operation (in millisecs) causes a caller to wait to see if a thread has died
- Example: threads/InterruptTest.java

A Thread's Life

A thread continues to execute until one of the following thing happens.

- it returns from its target `run()` method.
- it's interrupted by an uncaught exception.
- it's `stop()` method is called.

A Thread's Life

A thread continues to execute until one of the following thing happens.

- it returns from its target `run()` method.
- it's interrupted by an uncaught exception.
- it's `stop()` method is called.

What happens if the `run()` method never terminates, and the application that started the thread never calls the `stop()` method?

A Thread's Life

A thread continues to execute until one of the following thing happens.

- it returns from its target `run()` method.
- it's interrupted by an uncaught exception.
- it's `stop()` method is called.

What happens if the `run()` method never terminates, and the application that started the thread never calls the `stop()` method?

The thread remains alive even after the application has finished!
(so the Java interpreter has to keep on running...)

Daemon Threads

- Useful for simple, periodic tasks in an application.

Daemon Threads

- Useful for simple, periodic tasks in an application.
- The `setDaemon()` method marks a thread as a daemon thread that should be killed and discarded when no other application threads remain.

Daemon Threads

- Useful for simple, periodic tasks in an application.
- The `setDaemon()` method marks a thread as a daemon thread that should be killed and discarded when no other application threads remain.
- Code snippet:

```
class Devil extends Thread {
    Devil() {
        setDaemon( true);
        start();
    }
    public void run() {
        //perform evil tasks
        ...
    }
}
```

Thread Synchronization (1)

- Java threads are preemptible. Java threads may or may not be time-sliced. We should not make any timing assumptions.

Thread Synchronization (1)

- Java threads are preemptible. Java threads may or may not be time-sliced. We should not make any timing assumptions.
- Threads have priorities that can be changed (increased or decreased).

Thread Synchronization (1)

- Java threads are preemptible. Java threads may or may not be time-sliced. We should not make any timing assumptions.
- Threads have priorities that can be changed (increased or decreased).
- This implies that multiple threads will have race conditions (read/write conflicts based on time of access) when they run.

Thread Synchronization (1)

- Java threads are preemptible. Java threads may or may not be time-sliced. We should not make any timing assumptions.
- Threads have priorities that can be changed (increased or decreased).
- This implies that multiple threads will have race conditions (read/write conflicts based on time of access) when they run.
- We have to resolve these conflicts with proper design and implementation.

Thread Synchronization (1)

- Java threads are preemptible. Java threads may or may not be time-sliced. We should not make any timing assumptions.
- Threads have priorities that can be changed (increased or decreased).
- This implies that multiple threads will have race conditions (read/write conflicts based on time of access) when they run.
- We have to resolve these conflicts with proper design and implementation.
- Example of a race condition: Account.java, TestAccount.java

Thread Synchronization (2)

- Java has `synchronized` keyword for guaranteeing mutually exclusive access to a method or a block of code. Only one thread can be active among all synchronized methods and synchronized blocks of code in a class.

Thread Synchronization (2)

- Java has `synchronized` keyword for guaranteeing mutually exclusive access to a method or a block of code. Only one thread can be active among all synchronized methods and synchronized blocks of code in a class.

```
// Only one thread can execute the update method at a time in
    the class.
synchronized void update() { //... }

// Access to individual datum can also be synchronized.
// The object buffer can be used in several classes,
// enabling synchronization among methods from multiple classes.

synchronized(buffer) {
    this.value = buffer.getValue();
    this.count = buffer.length();
}
```

Thread Synchronization (2)

- Java has synchronized keyword for guaranteeing mutually exclusive access to a method or a block of code. Only one thread can be active among all synchronized methods and synchronized blocks of code in a class.

```
// Only one thread can execute the update method at a time in
    the class.
synchronized void update() { //... }

// Access to individual datum can also be synchronized.
// The object buffer can be used in several classes,
// enabling synchronization among methods from multiple classes.

synchronized(buffer) {
     this.value = buffer.getValue();
     this.count = buffer.length();
}
```

- Every Java object has an implicit monitor associated with it to implement the synchronized keyword. Inner class has a separate monitor than the containing outer class.

Thread Synchronization (2)

- Java has `synchronized` keyword for guaranteeing mutually exclusive access to a method or a block of code. Only one thread can be active among all synchronized methods and synchronized blocks of code in a class.

```java
// Only one thread can execute the update method at a time in
    the class.
synchronized void update() { //... }

// Access to individual datum can also be synchronized.
// The object buffer can be used in several classes,
// enabling synchronization among methods from multiple classes.

synchronized(buffer) {
     this.value = buffer.getValue();
     this.count = buffer.length();
}
```

- Every Java object has an implicit monitor associated with it to implement the synchronized keyword. Inner class has a separate monitor than the containing outer class.
- Java allows Rentrant Synchronization, that is, a thread can reacquire a lock it already owns. For example, a synchronized method can call another synchronized method.

Synchronization Example 1

- Example of a race condition: Account.java, TestAccount.java
- Thread safe version using `synchronized` keyword: SynchronizedAccount.java

Thread Synchronization (3)

- The `wait()` and `notify()` methods (of the `Object` class) allow a thread to give up its hold on a lock at an arbitrary point, and then wait for another thread to give it back before continuing.

Thread Synchronization (3)

- The `wait()` and `notify()` methods (of the `Object` class) allow a thread to give up its hold on a lock at an arbitrary point, and then wait for another thread to give it back before continuing.
- Another thread must call `notify()` for the waiting thread to wakeup. If there are other threads around, then there is no guarantee that the waiting thread gets the lock next. Starvation is a possibility. We can use an overloaded version of `wait()` that has a timeout.

Thread Synchronization (3)

- The `wait()` and `notify()` methods (of the `Object` class) allow a thread to give up its hold on a lock at an arbitrary point, and then wait for another thread to give it back before continuing.
- Another thread must call `notify()` for the waiting thread to wakeup. If there are other threads around, then there is no guarantee that the waiting thread gets the lock next. Starvation is a possibility. We can use an overloaded version of `wait()` that has a timeout.
- The method `notifyAll()` wakes up all waiting threads instead of just one waiting thread.

Example with wait()/notify()

```
class MyThing {
  synchronized void waiterMethod() {
    // do something
    // we need to wait for the notifier to do something
    // give up the lock, put calling thread to sleep
    wait();
    // continue where we left off
  }

  synchronized void notifierMethod() {
    // do something
    // notifier the waiter that we've done it
    notify();
    //do more things
  }

  synchronized void relatedMethod() {
    // do some related stuff
  }
}
```

Synchronization Example 2: Producer/Consumer Problem

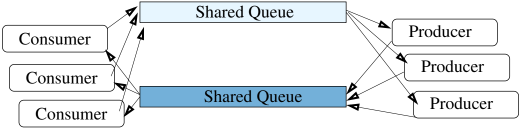
- A *producer* thread produces objects and places them into a queue, while a *consumer* thread removes objects and consumes them.

Synchronization Example 2: Producer/Consumer Problem

- A *producer* thread produces objects and places them into a queue, while a *consumer* thread removes objects and consumes them.
- Often, the queue has a maximum depth.

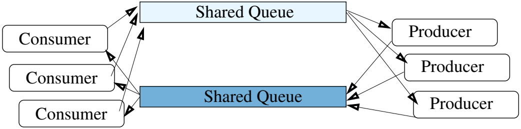
Synchronization Example 2: Producer/Consumer Problem

- A *producer* thread produces objects and places them into a queue, while a *consumer* thread removes objects and consumes them.
- Often, the queue has a maximum depth.
- The producer and consumer don't operate at the same speed. We can also have multiple producers/consumers as well as multiple shared queues!



Synchronization Example 2: Producer/Consumer Problem

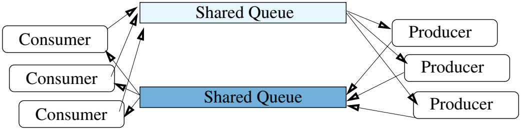
- A *producer* thread produces objects and places them into a queue, while a *consumer* thread removes objects and consumes them.
- Often, the queue has a maximum depth.
- The producer and consumer don't operate at the same speed. We can also have multiple producers/consumers as well as multiple shared queues!



- Example: Suppose the producer creates messages every second but the consumer reads and displays only every two seconds.

Synchronization Example 2: Producer/Consumer Problem

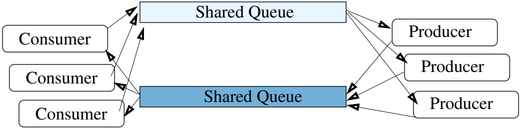
- A *producer* thread produces objects and places them into a queue, while a *consumer* thread removes objects and consumes them.
- Often, the queue has a maximum depth.
- The producer and consumer don't operate at the same speed. We can also have multiple producers/consumers as well as multiple shared queues!



- Example: Suppose the producer creates messages every second but the consumer reads and displays only every two seconds. *How long will it take for the queue to fill up? What will happen when it does?*

Synchronization Example 2: Producer/Consumer Problem

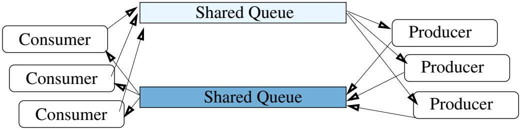
- A *producer* thread produces objects and places them into a queue, while a *consumer* thread removes objects and consumes them.
- Often, the queue has a maximum depth.
- The producer and consumer don't operate at the same speed. We can also have multiple producers/consumers as well as multiple shared queues!



- Example: Suppose the producer creates messages every second but the consumer reads and displays only every two seconds. *How long will it take for the queue to fill up? What will happen when it does?*
- Example: SharedQueue.java, Producer.java, Consumer.java, PC.java

Synchronization Example 2: Producer/Consumer Problem

- A *producer* thread produces objects and places them into a queue, while a *consumer* thread removes objects and consumes them.
- Often, the queue has a maximum depth.
- The producer and consumer don't operate at the same speed. We can also have multiple producers/consumers as well as multiple shared queues!



- Example: Suppose the producer creates messages every second but the consumer reads and displays only every two seconds. *How long will it take for the queue to fill up? What will happen when it does?*
- Example: SharedQueue.java, Producer.java, Consumer.java, PC.java
- The Producer/Consumer or a Thread Pool pattern is a widely used one for multi-threaded applications as well as in servers and clients.

Synchronization Example 3: Ping Pong

- Example: PingPong.java. Will this work correctly? Why or why not?

Synchronization Example 3: Ping Pong

- Example: PingPong.java. Will this work correctly? Why or why not?
- Example: SynchronizedPingPong.java. This solves the problem using `wait()` and `motify()` methods.
- Are the threads really simulating ping pong?

Synchronization Example 3: Ping Pong

- Example: PingPong.java. Will this work correctly? Why or why not?
- Example: SynchronizedPingPong.java. This solves the problem using `wait()` and `motify()` methods.
- Are the threads really simulating ping pong? We need them to exchange an object over the network!

Thread Groups

- Thread groups are hierarchical collection of threads.

```
ThreadGroup myTaskGroup = new ThreadGroup("My Task Group");
Thread myTask = new Thread(myTaskGroup, taskPerformer);
```

Thread Groups

- Thread groups are hierarchical collection of threads.

```
ThreadGroup myTaskGroup = new ThreadGroup("My Task Group");
Thread myTask = new Thread(myTaskGroup, taskPerformer);
```

- The `ThreadGroup` methods `stop()`, `suspend()`, `resume()` operate on all the threads in the group. A thread group can also be marked as a daemon. Otherwise we have to use `destroy()` to remove the thread group.

Thread Groups

- Thread groups are hierarchical collection of threads.

```
ThreadGroup myTaskGroup = new ThreadGroup("My Task Group");
Thread myTask = new Thread(myTaskGroup, taskPerformer);
```

- The `ThreadGroup` methods `stop()`, `suspend()`, `resume()` operate on all the threads in the group. A thread group can also be marked as a daemon. Otherwise we have to use `destroy()` to remove the thread group.
- The method `activeCount()` tells us how many threads are in the thread group; the method `enumerate()` gives us a list of the threads. The argument to enumerate is an array of threads.

Thread Groups

- Thread groups are hierarchical collection of threads.

```
ThreadGroup myTaskGroup = new ThreadGroup("My Task Group");
Thread myTask = new Thread(myTaskGroup, taskPerformer);
```

- The `ThreadGroup` methods `stop()`, `suspend()`, `resume()` operate on all the threads in the group. A thread group can also be marked as a daemon. Otherwise we have to use `destroy()` to remove the thread group.
- The method `activeCount()` tells us how many threads are in the thread group; the method `enumerate()` gives us a list of the threads. The argument to enumerate is an array of threads.
- Thread groups can contain other thread groups recursively. In that case the methods apply recursively.

Thread Groups

- Thread groups are hierarchical collection of threads.

```
ThreadGroup myTaskGroup = new ThreadGroup("My Task Group");
Thread myTask = new Thread(myTaskGroup, taskPerformer);
```

- The `ThreadGroup` methods `stop()`, `suspend()`, `resume()` operate on all the threads in the group. A thread group can also be marked as a daemon. Otherwise we have to use `destroy()` to remove the thread group.
- The method `activeCount()` tells us how many threads are in the thread group; the method `enumerate()` gives us a list of the threads. The argument to enumerate is an array of threads.
- Thread groups can contain other thread groups recursively. In that case the methods apply recursively.
- Example: ThreadGroupExample.java

Thread Pool

- Thread Pool: A number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads.
- Java provides a thead pool via the `Executor` interface from the `java.utl.concurrent` package.

Synchronization Wrappers (1)

- Many of the data structures in `java.util` package aren't synchronized. We can convert them into synchronized versions using wrappers.

Synchronization Wrappers (1)

- Many of the data structures in `java.util` package aren't synchronized. We can convert them into synchronized versions using wrappers.
- The synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary collection. Each of the six core collection interfaces – Collection, Set, List, Map, SortedSet, and SortedMap – has one static factory method.

Synchronization Wrappers (1)

- Many of the data structures in `java.util` package aren't synchronized. We can convert them into synchronized versions using wrappers.
- The synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary collection. Each of the six core collection interfaces – Collection, Set, List, Map, SortedSet, and SortedMap – has one static factory method.

```java
public static <T> Collection<T> synchronizedCollection(Collection<T> c);
public static <T> Set<T> synchronizedSet(Set<T> s);
public static <T> List<T> synchronizedList(List<T> list);
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);
public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V>
    m);
```

Synchronization Wrappers (1)

- Many of the data structures in `java.util` package aren't synchronized. We can convert them into synchronized versions using wrappers.
- The synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary collection. Each of the six core collection interfaces – Collection, Set, List, Map, SortedSet, and SortedMap – has one static factory method.

```java
public static <T> Collection<T> synchronizedCollection(Collection<T> c);
public static <T> Set<T> synchronizedSet(Set<T> s);
public static <T> List<T> synchronizedList(List<T> list);
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);
public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V>
    m);
```

- Create the synchronized collection with the following trick.

```java
List<Type> list = Collections.synchronizedList(new ArrayList<Type>());
```

Synchronization Wrappers (1)

- Many of the data structures in `java.util` package aren't synchronized. We can convert them into synchronized versions using wrappers.
- The synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary collection. Each of the six core collection interfaces – Collection, Set, List, Map, SortedSet, and SortedMap – has one static factory method.

```
public static <T> Collection<T> synchronizedCollection(Collection<T> c);
public static <T> Set<T> synchronizedSet(Set<T> s);
public static <T> List<T> synchronizedList(List<T> list);
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);
public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V>
    m);
```

- Create the synchronized collection with the following trick.

```
List<Type> list = Collections.synchronizedList(new ArrayList<Type>());
```

A collection created in this fashion is every bit as thread-safe as a normally synchronized collection, such as a `Vector`.

Synchronization Wrappers (2)

- In the face of concurrent access, it is imperative that the user manually synchronize on the returned collection when iterating over it. The reason is that iteration is accomplished via multiple calls into the collection, which must be composed into a single atomic operation.

Synchronization Wrappers (2)

- In the face of concurrent access, it is imperative that the user manually synchronize on the returned collection when iterating over it. The reason is that iteration is accomplished via multiple calls into the collection, which must be composed into a single atomic operation.

```java
Collection<Type> c = Collections.synchronizedCollection(myCollection);
synchronized(c) {
    for (Type e: c)
        process(e);
}
```

Synchronization Wrappers (2)

- In the face of concurrent access, it is imperative that the user manually synchronize on the returned collection when iterating over it. The reason is that iteration is accomplished via multiple calls into the collection, which must be composed into a single atomic operation.

```
Collection<Type> c = Collections.synchronizedCollection(myCollection);
synchronized(c) {
    for (Type e: c)
        process(e);
}
```

- If an explicit iterator is used, the iterator method must be called from within the synchronized block. Failure to follow this advice may result in nondeterministic behavior.

Synchronization Wrappers (2)

- In the face of concurrent access, it is imperative that the user manually synchronize on the returned collection when iterating over it. The reason is that iteration is accomplished via multiple calls into the collection, which must be composed into a single atomic operation.

```
Collection<Type> c = Collections.synchronizedCollection(myCollection);
synchronized(c) {
    for (Type e: c)
        process(e);
}
```

- If an explicit iterator is used, the iterator method must be called from within the synchronized block. Failure to follow this advice may result in nondeterministic behavior.
- One minor downside of using wrapper implementations is that you do not have the ability to execute any noninterface operations of a wrapped implementation.

Synchronization Wrappers (2)

- In the face of concurrent access, it is imperative that the user manually synchronize on the returned collection when iterating over it. The reason is that iteration is accomplished via multiple calls into the collection, which must be composed into a single atomic operation.

```
Collection<Type> c = Collections.synchronizedCollection(myCollection);
synchronized(c) {
    for (Type e: c)
        process(e);
}
```

- If an explicit iterator is used, the iterator method must be called from within the synchronized block. Failure to follow this advice may result in nondeterministic behavior.
- One minor downside of using wrapper implementations is that you do not have the ability to execute any noninterface operations of a wrapped implementation.

For more details, see:
http://docs.oracle.com/javase/tutorial/collections/implementations/wrapper.html

References

- Brian Goetz, Tim Peierls, Joshua Bloch and Joseph Bowbeer: *Java Concurrency in Practice*
- Lewis and Berg: *Multithreaded Programming with Java Technology*