Objectives

- In this session, you will learn to:
  - Use the PathMatcher class
  - Use NIO.2 classes
  - Work with threads
  - Work with non shared data
  - Synchronize threads accessing shared data
  - Identify potential threading problems

Why Threading Matters

- Application code can execute in a single threaded or multithreaded environment.
- Single threaded applications are slower due to performance bottlenecks.
- Performance bottlenecks should be avoided for programs to execute fast.
- Some of the bottlenecks include:
  - Resource contention
  - Blocking I/O operations
  - Underutilization of CPUs
- Multithreading enables to avoid the potential bottlenecks.

Why Threading Matters (Contd.)

- Types of tasks scheduled for execution:
  - Process:
    - Area of memory that contains both code and data
    - Has a thread of execution:
      - Scheduled to receive CPU time slices
  - Thread:
    - Enables scheduled execution of a process
    - For a process:
      - Shares the same data memory
      - May follow different paths

The Thread Class

- `Thread` class:
  - Used to create and start threads
- Class can create a new thread, if it:
  - Extends the `Thread` class

    Or
  - Implements the `Runnable` interface

Extending Thread

- Perform the following steps to implement threads:
  1. Extend the class from the `java.lang.Thread` class.
  2. Override the `run()` method.
     - Example:

```
public class ExampleThread extends Thread {
@Override
public void run()
{
for(int i = 0; i < 100; i++) {
System.out.println("i:" +i);
}}
}
```

- `run()` method:
  - Should not be called directly
  - Does not start a new thread

Starting a Thread

- A thread can be started only once.
- Thread's `start()` method:
  - Used to begin executing a thread
  - Example:

```
public static void main(String[] args) {
ExampleThread t1 = new ExampleThread();
t1.start();
    }
```

Schedules the `run()` method to be called.

Implementing Runnable

- The following code snippet shows how to implement the `Runnable` interface:

```
public class ExampleRunnable implements Runnable {
@Override
public void run() {
for(int i = 0; i < 100; i++) {
System.out.println("i:" + i);
}
}
}
```

Executing Runnable Instances

- The following code snippet shows how to start a thread using a runnable instance:

```
public static void main(String[] args) {
ExampleRunnable r1 = new ExampleRunnable();
Thread t1 = new Thread(r1);
t1.start();
}
```

Activity: ImplementingRunnableExample

Let us see an example of using the `Runnable` interface in Java.

A Runnable with Shared Data

- Static and instance fields may be shared by threads, as shown in the following code snippet:

```
public class ExampleRunnable implements Runnable {
private int i;
 @Override
 public void run() {
 for(i = 0; i < 100; i++) {
 System.out.println("i:" + i);
 }
 }
 }
```

Potentially shared variable

One Runnable: Multiple Threads

- `Runnable`:
  - Instance can be passed to multiple `Thread` instances
  - Fields are shared by multiple `Thread` instances
- The following embedded Word document shows how the static fields are concurrently accessed by the multiple threads.

Accessing static fields

Problems with Shared Data

- Shared data:
  - Instance and static fields
  - Must be accessed by threads with caution because it may be changed concurrently by multiple threads

Nonshared Data

- Some variable types are never shared.
- Any shared data that is immutable is thread-safe.
- Thread-safe types:
  - Local variables
  - Method parameters
  - Exception handler parameters

Out-of-Order Execution

- Thread operations:
  - Performed on one thread may not appear to be executed in the same order by another
  - May result in out-of-order execution due to:
    - Code optimization
    - Threads may be operating on cached copies of shared variables
  - Must be synchronized in their actions to ensure consistent behavior
  - Have working memory, where it keeps its own copy of variables

Out-of-Order Execution (Contd.)

- Synchronization of a thread's working memory with main memory involves:
  - Volatile read or write of a variable
  - Locking or unlocking a monitor
  - First and last action of a thread
  - Actions that start a thread or detect that a thread has terminated

The volatile Keyword

- `volatile`:
  - Modifier can be applied to a field
    - Example:
      ```
      public volatile int i;
      ```
  - Field causes a thread to synchronize its working memory with main memory
  - Does not mean atomic

Stopping a Thread

- The following embedded Word document shows how a thread stops by completing its `run()` method.

Stopping thread

- `Main` thread:
  - Executed in a separate thread
  - Automatically created by the JVM

The synchronized Keyword

- `synchronized` keyword:
  - Used to create thread-safe code blocks
- `synchronized` code block:
  - Causes a thread to write all of its changes to main memory
  - Used to group blocks of code for exclusive execution

synchronized Methods

- The following embedded Word document shows how to work with synchronized methods.

Synchronized methods

synchronized Blocks

- The following embedded Word document shows how to use the synchronized blocks.

Synchronized blocks

- Synchronization in multithreaded applications ensures reliable behavior.
- `Synchronized`:
  - Blocks and methods are used to restrict a section of code to a single thread
  - Blocks can be used instead of `synchronized` methods

Object Monitor Locking

- Each object in Java is associated with a monitor.
- `Synchronized` methods:
  - Use monitor for the `this` object
  - That are static use the classes' monitor
- `Synchronized` blocks:
  - Specify which object's monitor to lock or unlock
  - Example:

```
synchronized ( this ) { }
```

  - Nested to lock multiple monitors simultaneously

Activity: SynchronizedExample

Let us see an example of using the `synchronized` keyword in Java.

Detecting Interruption

- The following embedded Word document shows how to interrupt a thread.

Interrupt a thread

Interrupting a Thread

- `interrupt()` method:
  - Convenient way to stop a thread
  - Can Interrupt a thread that is blocked
- Every thread has the `interrupt()` and `isInterrupted()` methods, as shown in the following code snippet:

```
public static void main(String[] args) {
ExampleRunnable r1 = new ExampleRunnable();
Thread t1 = new Thread(r1);
t1.start();
// ...
t1.interrupt();
}
```

Interrupts a thread

Thread.sleep()

- The following embedded Word document shows how a `Thread` can pause execution, using the `sleep()` method.

The sleep() method

Activity: ThreadSleepExample

Let us see an example of how to use the `sleep()` method with threads.

Additional Thread Methods

- `Thread` and threading-related methods:
  - `setName(String)`
  - `getName()`
  - `getId()`
  - `isAlive()`
  - `isDaemon()`
  - `setDaemon(boolean)`
  - `join()`
  - `Thread.currentThread()`
- Threading methods of the `Object` class:
  - `wait()`
  - `notify()`
  - `notifyAll()`

Additional Thread Methods (Contd.)

- Daemon thread:
  - Background thread
  - Less important
- Non daemon thread:
  - Main thread
  - Keeps the JVM running, even after the main method has returned
  - Set for threads that prevent the JVM from quitting

Methods to Avoid

- `Thread` methods that should be avoided:
  - `setPriority(int)`
  - `getPriority()`
- Deprecated methods:
  - `destroy()`
  - `resume()`
  - `suspend()`
  - `stop()`

Deadlock

- Potential threading problems:
  - Deadlock:
    - May occur when two or more threads are blocked forever as they are waiting for each other
    - Examples:

```
synchronized (obj1){
synchronized (obj2){
}}
synchronized (obj2){
synchronized (obj1){
}}
```

Thread 1 pauses after locking `obj1`'s monitor.

Thread 2 pauses after locking `obj2`'s monitor.

  - Starvation:
    - Thread is unable to gain regular access to shared resources
  - Livelock:
    - Occurs when threads are busy responding to each other

Quiz

Get Ready for the Challenge

Quiz (Contd.)

- Fill in the blank:
  - The _____ method is a convenient way to stop a thread.

- Solution:
  - `interrupt()`

Quiz (Contd.)

- Fill in the blank:
  - _____ occurs when threads are busy responding to each other.

- Solution:
  - Livelock

Summary

- In this session, you learned that:
  - The glob syntax is similar to regular expressions.
  - The `FileStore` class provides information about a file system, such as the total, usable, and allocated disk space.
  - The `toPath()` method is added to the `java.io.File` class.
  - The `Runnable` instance can be passed to multiple `Thread` instances.
  - Any shared data that is immutable is thread-safe.
  - The `synchronized` keyword is used to create thread-safe code blocks.
  - The `interrupt()` method is a convenient way to stop a thread.
  - Daemon threads are background threads.
  - Potential threading problem includes:
    - Deadlock
    - Starvation
    - Livelock