Threads

Java Threads
- A thread is computer code being executed.
- More than one thread can be executed virtually simultaneously (actually interleaved).
  - The code for the threads can be the same, or different.
  - Each thread has its own state, sort of.
  - Threads can share variables, and modify the variables they share.
- Programs with > 1 thread are called "concurrent programs".

Timing of Threads
- Threads don’t progress in lock-step fashion.
- One may be started and another stopped in an unpredictable fashion by the operating system.
- This behavior is called asynchronous.

Similar Idea: Processes
- A process is also code in execution.
- Typically processes don’t share variables, although limited sharing is possible.
- Multiple processes is common in, e.g. UNIX.
- Processes are "heavy weight", threads are "light weight".
- Weight refers to the cost of switching the processor from one unit’s state to another’s.

Why are Threads Useful?
- May wish to have multiple activities going on at once.
- Don’t want one activity’s waiting (e.g. for an event) to stop the other activities.
- This is only doable with threads (or processes).

Thread Example
- On thread is a computational one, that occasionally needs to wait for input from the outside, say from an input stream of characters.
- Another thread may be a graphical user interface, responding to mouse events.
- We don’t want waiting for input to hold up the graphics, or waiting for a click to hold up the computational thread.
- In fact, the click might tell the computational thread to alter its behavior.
Thread Example

- Bouncing Balls Example
- Each ball is run by a separate thread.
- Each thread can, in principal, be interrupted and re-started independently of the others.
- If a ball is "clicked" in mid-air, it will suspend, and resume if clicked a second time.

Two Ways to Have Threads in Java

- extends Thread
  - Thread is a base class with threading capability.
- implements Runnable
  - Runnable is an interface that requires method
    - void run()
  - The latter is preferred, because it does not take away your ability to inherit from another class (multiple inheritance is not allowed in Java).

Using “implements Runnable”

- The class that implements Runnable still needs to contain a Thread.
- This Thread is what controls starting and stopping.

Ball “extends Thread” Code

```java
/**
 * Ball class represents ball's state information
 */
public class Ball extends Thread // Thread implements Runnable
{
    double x, y; // this ball's coordinates
    double deltaX, deltaY; // this ball's velocities
    String myNumber; // ball's number as a string

    public Ball(...) // constructor
    {
        myThread = new Thread(this); // make thread for Ball
    }

    public void run() // run method for this Runnable
    {
        while(true)
        {
            move(); // move the ball
            sleep(app.delay); // sleep (defined in Thread)
        }
    }
}
```

Ball “implements Runnable” Code

```java
class Ball implements Runnable
{
    Thread myThread; // this ball's thread
    double x, y; // this ball's coordinates
    double deltaX, deltaY; // this ball's velocities
    String myNumber; // ball's number as a string

    Ball(x, y, number) // constructor
    {
        myThread = new Thread(this); // make thread for Ball
    }

    public void run() // run method for this Runnable
    {
        while(true)
        {
            move(); // move the ball
            myThread.sleep(app.delay); // sleep
        }
    }
}
Cautions about Threads

- Reasoning about concurrent programs is inherently more difficult than reasoning about sequential ones.
- They can exhibit non-deterministic behavior, when variables are shared among threads.

Non-Determinism

Suppose \( x \Leftarrow 1 \) initially.

\[
\begin{align*}
\text{Thread 1} & \quad \text{Thread 2} \\
\{ & \quad \{ \\
\quad x \Leftarrow x + 2; & \quad x \Leftarrow x \times 5; \\
\quad \} & \quad \} \\
\text{What is } x \text{ now?} & \\
\end{align*}
\]

Interesting Methods of Thread

**start**

```java
public void start() {
    // Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
}
```

The result is that two threads are running concurrently: the current thread (which returns from the call to the start method) and the other thread (which executes its run method).

Throws:

- `IllegalThreadStateException` - if the thread was already started.

Methods of Thread

**currentThread**

```java
public static Thread currentThread() {
    // Returns a reference to the currently executing thread object.
}
```

So "executing" is more specific than "running".
- "executing" means "has the processor"
- "running" means "able to execute"

Methods of Thread

**yield**

```java
public static void yield() {
    // Causes the currently executing thread object to pause temporarily and allow other threads to execute.
}
```

Methods of Thread

**sleep**

```java
public static void sleep(long millis) throws InterruptedException {
    // Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
    // The thread does not lose ownership of any monitors.
}
```

Parameters:

- `millis` - the length of time to sleep in milliseconds.

Throws:

- `InterruptedException` - if another thread has interrupted the current thread.
- The interrupted status of the current thread is cleared when this exception is thrown.
Methods of Thread

**interrupt**

```java
public void interrupt()
```

Interrupts this thread.

First the `checkAccess` method of this thread is invoked, which may cause a `SecurityException` to be thrown.

**setPriority**

```java
public final void setPriority(int newPriority)
```

Changes the priority of this thread.

First the `checkAccess` method of this thread is called with no arguments. This may result in throwing a `SecurityException`.

Otherwise, the priority of this thread is set to the smaller of the specified `newPriority` and the maximum permitted priority of the thread's thread group.

Methods of Thread

**join**

```java
public final void join(long millis)
```

Waits at most `millis` milliseconds for this thread to die. A timeout of 0 means to wait forever.

**Runnable**

```java
Runnable
```

*Known Implementing Classes:* `Thread`, `TimerTask`

The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called `run`.

This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, `Runnable` is implemented by class `Thread`.

Being active simply means that a thread has been started and has not yet been stopped.

In addition, `Runnable` provides the means for a class to be active while not subclassing `Thread`. A class that implements `Runnable` can run without subclassing `Thread` by instantiating a `Thread` instance and passing itself as the target. In most cases, the `Runnable` interface should be used if you are only planning to override the `run()` method and no other `Thread` methods.

This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.