Threads vs Processes

- A process has its own address space
  - In a multitasking operating system, each program is run as a separate process
  - Process switching has overhead
- A thread shares the address space of the program that created it
  - Minimal overhead with thread switching
the Thread class

- `System.out.println("hello");`
- `Thread.sleep(1000);`
- `System.out.println("world");`

puts currently executing thread to sleep for 1000 ms

static method

actually....

```java
try {
    Thread.sleep(1000);
} catch (InterruptedException e) { }
```
Creating your own thread

- Two ways to create a Java thread:
  - Extend the Thread class and override the run() method
  - Implement the Runnable interface
    - define run()

Whatever code is in run() will be run as a separate thread.

Subclass Thread

Thread class inherits MyThread

Thread class has run() method that does nothing

MyThread override run() - that's what your thread will do
Subclassing Thread

class SimpleThread extends Thread {
    private String internalName;
    
    SimpleThread (String name) {
        internalName = name;
    }

    public void run() {
        for (int i=0; i<5; i++) {
            System.out.println(internalName);
        }
    }
}

Activating your thread

The Java runtime system sends the run message

the programmer sends the start message to the thread object
public class SimpleThreadTest1 {
    public static void main (String argv[]) {
        SimpleThread t1 = new SimpleThread("sun");
        SimpleThread t2 = new SimpleThread("java");
        SimpleThread t3 = new SimpleThread("beans");
        t1.start();
        t2.start();
        t3.start();
    }
}

class SimpleThread extends Thread {
    String internalName;
    public void run() {
        for (int i=0; i<5; i++) {
            System.out.println(internalName);
        }
    }
}

Output:
sun
sun
java
java
java
sun
beans
beans
sun
beans
beans
beans
Don’t try to out-guess me!

The “Runnable” pool of threads
Option 2. Runnable interface

class SayName implements Runnable {
    private String internalName;

    SayName (String name) {
        internalName = name;
    }

    public void run() {
        for (int i=0; i<8; i++) {
            System.out.println(internalName);
        }
    }
}

Working with Runnable objects

SayName say1 =
    new SayName("sun");
Thread thread1 =
    new Thread(say1);

class SayNameTest {
    public static void main (String [ ] args ) {
        SayName say1 =   new SayName("sun");
        SayName say2 =   new SayName("java");

        Thread thread1 =  new Thread(say1);
        thread1.start();

        Thread thread2 =  new Thread(say2);
        thread2.start();
    }
}

java SayNameTest
sun..java..java..java..java..java..java..java
..java..sun..sun..sun..sun..sun..sun..sun..

Thread Creation RULE

• Always create an instance of Thread
• Rule:
  - either use new with a subclass of Thread
    • new MyThread( );
  - or use new with Thread (anObject)
    • new Thread (anObject);  
      must implement
      Runnable interface
Thread Execution

```java
SimpleThread t1 = new SimpleThread("sun");
SimpleThread t2 = new SimpleThread("java");
SimpleThread t3 = new SimpleThread("beans");
t1.start();
t2.start();
t3.start();
```

Hog Prevention with yield()

- Use yield to prevent a thread from hogging the CPU

```java
public void run() {
    for (int i=0; i<8; i++) {
        System.out.println(internalName);
        Thread.yield();
    }
}
```

```
> java SayNameTest
sun..java..sun..java..sun..java..sun..java..
sun..java..sun..java..java..sun..java..sun..
sun..java..sun..java..java..sun..java..sun..
sun..java..sun..java..java..sun..java..sun..
sun..java..sun..java..java..sun..java..sun..
sun..java..sun..java..java..sun..java..sun..
sun..java..sun..java..java..sun..java..sun..
```
Life Cycle of a Thread

New Thread

Runnable

Dead

Not Runnable

Thread t =
new Thread()
Thread t =
new Thread(myObj)

Runnable

How to make a thread NOT Runnable -
1. invoke its sleep method
2. thread blocks on IO
3. a thread executes a wait() method

Partitioning Work with Threads
Three Work Threads

t1.start(); \(\rightarrow\) returns immediately
t2.start();
t3.start();
System.out.println("done");

will print before threads complete their work

---------

Three Work Threads

t1.start(); \(\rightarrow\) returns immediately
t2.start();
t3.start();
t1.join(); \(\rightarrow\) main thread will block until t1 is no longer alive
t2.join();
t3.join();
System.out.println("done");

will print after all threads complete their work
Threads II
Synchronization

Multiple Thread Problem

t1:ReservationThread

1. locate free seat
2. assign seat to customer

↓

t2:ReservationThread

1. locate free seat
2. assign seat to customer
Solution

1. locate free seat
2. assign seat to customer

only one thread allowed to execute critical code at a time

synchronized methods lock other threads out

object is LOCKED - only one thread can execute synchd code at a time

1. locate free seat
2. assign seat to customer
class Account {
    private double balance;

    public Account (double initialDeposit) {
        balance = initialDeposit;
    }
    public synchronized double getBalance ( ) {
        return balance;
    }
    public synchronized void deposit (double amount) {
        balance += amount;
    }
}

what is locked?

class Account {
    synchronized double getBalance() { 
        return balance;
    }
}

Account a1 = new Account(200);  
Account a2 = new Account(500);  

the object instance is locked

when a thread enters synchronized code --
all other threads trying to access other
synchronized methods of that object, are blocked!
class Agent {
    a = new Agent();
    synchronized boolean foo() {
        //code
    }
    synchronized boolean bar() {
        //code
    }
}

Thread t1
void run () {
    a.foo();
}

Thread t2
void run () {
    a.bar();
}

Object is locked until the synchronized block has completed

Thread t1
void run () {
    a.foo();
}

Thread t2
void run () {
    a.bar();
}
synchronized boolean foo() {
    //... code
}

synchronized boolean bar() {
    //... code
}

boolean boz() {
    //... code
}

Objects & Synchronized Methods

foo1

- synchronized methods
- non-synchronized methods

- data

instance of Foo

foo2

- synchronized methods
- non-synchronized methods

- data

instance of Foo

other threads wait to execute a.foo or a.bar until t1 completes

ok to execute non-synchronized
"i own the lock"

foo1

synchronized methods

non-synchronized methods

instance of Foo

data

all synchronized methods of this object are unavailable to other threads

foo2

synchronized methods

non-synchronized methods

instance of Foo

data

"i own the lock to this object"

foo1

synchronized methods

non-synchronized methods

instance of Foo

data

foo2

synchronized methods

non-synchronized methods

instance of Foo

data

other objects have their own locks
What about variables?

• Variables cannot be synchronized
• However you can control access if you:
  - declare variables private
  - provide accessor methods

```java
class Account {
    private double balance;
    synchronized void updateBalance (double amt) {
        balance += amt;
    }
}
```
**Synchronized statements**

- Allows locking an object without a synchronized method
- Form:

  ```java
  synchronized (<some object>) {
    statements;
  }
  
  some object to lock
  ```

**Control array access**

```
public static void abs (int[] values) {
  synchronized (values) {
    for (int i=0; i<values.length; i++) {
      if (values[i] < 0)
        values[i] = -values[i];
    }
  }
}
```

locks the array values

any object may serve as the lock object
Synchronized Summary

• Synchronized methods or code blocks
  LOCK an object
• Synchronization permits multiple
  threads to act concurrently without
  interfering with each other

END