CENG 1004
Introduction to Object Oriented Programming

Spring 2016

WEEK 12

Concurrency

Concurrency

- Doing more than one thing at a time.
- You can continue to work in a word processor, while other applications download files, and stream audio.
- Even a single application is often expected to do more than one thing at a time.

Concurrency in Java

- Basic concurrency support in the Java programming language and the Java class libraries.

- High-level APIs in the **java.util.concurrent** packages.

Processes and Threads

- In concurrent programming, there are two basic units of execution:
  - processes and
  - threads

- In the Java programming language, concurrent programming is mostly concerned with threads.

Processes and Threads

- Today, most computer systems have multiple processors or processors with multiple execution cores.

- But concurrency is possible even on simple systems, without multiple processors or execution cores.

Processes

- Often seen as synonymous with programs or applications.

- Most implementations of the Java virtual machine run as a single process.

Threads

- Threads are sometimes called lightweight processes.
- Threads exist within a process — every process has at least one.
- You start with just one thread, called the main thread. This thread has the ability to create additional threads.

Defining and Starting a Thread

- An application that creates an instance of Thread must provide the code that will run in that thread.

- There are two ways to do this:
  - Provide a Runnable object.
  - Subclass Thread.

Runnable Interface

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

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

}
```

Thread class

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```

- Notice that both examples invoke Thread.start in order to start the new thread.

Pausing Execution with Sleep

- **`Thread.sleep`** causes the current thread to suspend execution for a specified period.

- Sleep times are not guaranteed to be precise, because they are limited by the facilities provided by the underlying OS.

- You cannot assume that invoking sleep will suspend the thread for precisely the time period specified.

Interrupts

- An indication to a thread that it should stop what it is doing and do something else.
- How to respond to an interrupt is up to the programmer
  - it is very common for the thread to terminate.
- A thread sends an interrupt to another thread.
- For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.

Supporting Interruption

- How does a thread support its own interruption?
  - This depends on what it's currently doing.
- If the thread is frequently invoking methods that throw InterruptedException, it simply returns from the run method after it catches that exception.
  - Many methods that throw InterruptedException, such as sleep, are designed to cancel their current operation and return immediately when an interrupt is received.

Supporting Interruption

- If a thread goes a long time without invoking a method that throws InterruptedException
  - Then it must periodically invoke Thread.interrupted, which returns true if an interrupt has been received.

- When a thread checks for an interrupt by invoking the static method Thread.interrupted, interrupt status is cleared.

Joins

- The join method allows one thread to wait for the completion of another.

```
t.join();
```

- causes the current thread to pause execution until t's thread terminates.

- Like sleep, join responds to an interrupt by exiting with an InterruptedException.

Synchronization

- Threads communicate primarily by sharing access to fields and the objects reference fields refer to.

- This form of communication is extremely efficient, but makes two kinds of errors possible:
  - thread interference and
  - memory consistency errors.

- The tool needed to prevent these errors is synchronization.

Thread Interference

- Consider a simple class called Counter

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

Thread Interference

- Counter is designed so that
  - each invocation of increment will add 1 to c,
  - and each invocation of decrement will subtract 1 from c.

- However, if a Counter object is referenced from multiple threads, interference between threads may prevent this from happening as expected.

Thread Interference

- Interference happens when two operations, running in different threads, but acting on the same data, interleave.
- This means that the two operations consist of multiple steps, and the sequences of steps overlap.
- What are the overlapping steps in this example?

Thread Interference

- Even simple statements can translate to multiple steps by the virtual machine.

- c++ can be decomposed into three steps:
  - Retrieve the current value of c.
  - Increment the retrieved value by 1.
  - Store the incremented value back in c.

- The expression c-- can be decomposed the same way, except that the second step decrements instead of increments.

Thread Interference

- Suppose Thread A invokes increment at about the same time Thread B invokes decrement. If the initial value of c is 0, their interleaved actions might follow this sequence:
  - Thread A: Retrieve c.
  - Thread B: Retrieve c.
  - Thread A: Increment retrieved value; result is 1.
  - Thread B: Decrement retrieved value; result is -1.
  - Thread A: Store result in c; c is now 1.
  - Thread B: Store result in c; c is now -1.

Synchronized Methods

- The Java programming language provides two basic synchronization idioms:
  - synchronized methods and
  - synchronized statements.

```
public synchronized void increment() {
    c++;
}

public synchronized void decrement() {
    c--;
}
```

Synchronized Methods

- It is not possible for two invocations of synchronized methods on the same object to interleave.

- When one thread is executing a synchronized method for an object,
  - all other threads that invoke synchronized methods for the same object block (suspend execution)
  - until the first thread is done with the object.

Synchronized Statements

- Unlike synchronized methods, synchronized statements must specify the object that provides the lock:

```
public void increment() {
 synchronized(this){
        c++;
 }
}

public void decrement() {
 synchronized (this) {
        c--;
        }

}
```

Synchronized Statements

```
public class MsLunch {
    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

- Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.

- When Deadlock runs, it's extremely likely that both threads will block when they attempt to lock each other's object

Guarded Blocks

- Threads often have to coordinate their actions.
- The most common coordination idiom is the guarded block.
- Such a block begins by polling a condition that must be true before the block can proceed.
- There are a number of steps to follow in order to do this correctly.

Guarded Blocks

- Looping until the condition is satisfied, but that loop is wasteful, since it executes continuously while waiting.

```
public void guardedJoy() {
    // Simple loop guard. Wastes
    // processor time. Don't do this!
    while(!joy) {}
    System.out.println("Joy has been achieved!");
}
```

Guarded Blocks

- A more efficient guard invokes Object.wait to suspend the current thread.

```
public synchronized void guardedJoy() {
    // This guard only loops once for each special event,
   which may not
    // be the event we're waiting for.
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been
   achieved!");
}
```

Guarded Blocks

- The invocation of wait does not return until another thread has issued a notification that some special event may have occurred

- Always invoke wait inside a loop that tests for the condition being waited for. Don't assume that the interrupt was for the particular condition you were waiting for

- When wait is invoked, the thread releases the lock and suspends execution.

Guarded Blocks

- At some future time, another thread will acquire the same lock and invoke Object.notifyAll

```
public synchronized notifyJoy() {
    joy = true;
    notifyAll();
}
```

- Some time after the second thread has released the lock, the first thread reacquires the lock and resumes by returning from the invocation of wait.

Enum Types

Enum Types

- A special data type that enables for a variable to be a set of predefined constants.
- The variable must be equal to one of the values that have been predefined for it.
- Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week.

Enum Types

- You define an enum type by using the enum keyword.

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

Enum Types

```
public class EnumTest {
    Day day;

    public EnumTest(Day day) {
        this.day = day;
    }

    public void tellItLikeItIs() {
        switch (day) {
            case MONDAY:
                System.out.println("Mondays are bad.");
                break;

            case FRIDAY:
                System.out.println("Fridays are better.");
                break;

            case SATURDAY: case SUNDAY:
                System.out.println("Weekends are best.");
                break;

            default:
                System.out.println("Midweek days are so-so.");
                break;
        }
    }
```

Enum Types

```
public static void main(String[] args) {
        EnumTest firstDay = new EnumTest(Day.MONDAY);
        firstDay.tellItLikeItIs();
        EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);
        thirdDay.tellItLikeItIs();
        EnumTest fifthDay = new EnumTest(Day.FRIDAY);
        fifthDay.tellItLikeItIs();
        EnumTest sixthDay = new EnumTest(Day.SATURDAY);
        sixthDay.tellItLikeItIs();
        EnumTest seventhDay = new EnumTest(Day.SUNDAY);
        seventhDay.tellItLikeItIs();
    }
}
```

Enum Types

- The enum class body can include methods and other fields.
- The compiler automatically adds some special methods when it creates an enum.
- For example, they have a static **values** method that returns an array containing all of the values of the enum in the order they are declared.

```
for (Planet p : Planet.values()) {
    System.out.printf("Your weight on %s is %f%n",
                      p, p.surfaceWeight(mass));
}
```

Enum Types

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);

    private final double mass;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    private double mass() { return mass; }
    private double radius() { return radius; }
```

Enum Types

```
    // universal gravitational constant  (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage: java Planet <earth_weight>");
            System.exit(-1);
        }
        double earthWeight = Double.parseDouble(args[0]);
        double mass = earthWeight/EARTH.surfaceGravity();
        for (Planet p : Planet.values())
           System.out.printf("Your weight on %s is %f%n",
                             p, p.surfaceWeight(mass));
    }
}
```

Enum Types

- The constructor for an enum type must be package-private or private access.
  - It automatically creates the constants that are defined at the beginning of the enum body. You cannot invoke an enum constructor yourself.

- In addition to its properties and constructor, Planet has methods that allow you to retrieve the surface gravity and weight of an object on each planet.

- All enums implicitly extend java.lang.Enum. An enum cannot extend anything else.

Nested Classes

Nested Classes

- The Java programming language allows you to define a class within another class. Such a class is called a nested class.

```
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

Nested Classes

- Nested classes are divided into two categories:
  - static and
  - non-static.

- Nested classes that are declared static are called **static nested classes**.

- Non-static nested classes are called **inner classes.**

Nested Classes

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
    class InnerClass {
        ...
    }
}
```

Nested Classes

- Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private.

- Static nested classes do not have access to other members of the enclosing class.

- As a member of the OuterClass, a nested class can be declared private, public, protected, or package private. (Recall that outer classes can only be declared public or package private.)

Static Nested Classes

- Static nested classes are accessed using the enclosing class name:

```
OuterClass.StaticNestedClass
```

- To create an object for the static nested class, use this syntax:

```
OuterClass.StaticNestedClass nestedObject =
     new OuterClass.StaticNestedClass();
```

Inner Classes

- Associated with an instance of its enclosing class and has direct access to that object's methods and fields.

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

Inner Classes

- An inner class cannot define any static members itself.
- Objects that are instances of an inner class exist within an instance of the outer class.
- An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance.

Inner Classes

- To instantiate an inner class, you must first instantiate the outer class.

- Then, create the inner object within the outer object with this syntax:

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Inner Classes

- We can create a long hierarchy of inner classes as long as we want to :

```
private class OuterClass {
  public class InnerClassA {
   public class InnerClassB {
   }
  }
 }
```

Inner Classes

- Outer class can create as many numbers of instances of inner class inside its code.

```
public class OuterClass {

 class InnerClass {
  public void printMe() {
   System.out.println("I am inner class !");
  }
 }

 void callInner(){
  InnerClass inner = new InnerClass();
  inner.printMe();

  InnerClass inner1 = new InnerClass();
  inner1.printMe();
 }
}
```

Inner Classes

- To create an instance of inner class in static method of outer class, you should have the instance of outer class

```
public class OuterClass {

 class InnerClass {
 }

 static void callInner() {
  /*
   * way of creating an inner class object in static method of outer class
   */
  OuterClass outClass = new OuterClass();
  OuterClass.InnerClass inner = outClass.new InnerClass();

  /*
   * another way of creating an inner class object in static method of
   * outer class
   */
  InnerClass inner1 = new OuterClass().new InnerClass();
 }
}
```

Inner Classes

- To create an instance of inner class in another class, you should have the instance of outer class

```
class AnotherClass {
 void callInner() {
  /*
   * way of creating an inner class object in static method of outer class
   */
  OuterClass ouetClass = new OuterClass();
  OuterClass.InnerClass inner = ouetClass.new InnerClass();

  /*
   * another way of creating an inner class object in static method of
   * outer class
   */
  OuterClass.InnerClass inner1 = new OuterClass().new InnerClass();
 }
}
```

Inner Classes

- An inner class have free access to all members of its outer class, no matter what the access level of outer class members has.

```
public class OuterClass {

 String def = "default";
 public String pub = "public";
 private String pri = "private";
 protected String pro = "protected";

 class InnerClass {
  void printMe() {
   System.out.println(def + " " + pub + " " + pri + " " + pro);
  }
 }

}
```

Inner Classes

- In case the inner class have same variable name as the outer class, than outer class variable can be called as follows:

```
public class OuterClass {

 public String pub = "Outer - public";

 class InnerClass {
  public String pub = "Inner - public";

  void printMe() {

   // i am calling local vaiable
   System.out.println(pub);

   // i am calling outer class variable
   System.out.println(OuterClass.this.pub);
  }
 }
}
```

Local and Anonymous Classes

- There are two additional types of inner classes.
- You can declare an inner class within the body of a method.
  - These classes are known as **local classes.**
- You can also declare an inner class within the body of a method without naming the class.
  - These classes are known as **anonymous classes.**

Local Classes

```
public class LocalClassExample {
   ...
    public static void validatePhoneNumber(
         String phoneNumber1, String phoneNumber2) {
       ...
       class PhoneNumber {
             ...
       }
       PhoneNumber myNumber1 = new
             PhoneNumber(phoneNumber1);
       PhoneNumber myNumber2 = new
             PhoneNumber(phoneNumber2);
   }
}
```

Local Classes

- A local class has access to the members of its enclosing class.

- In addition, a local class has access to local variables. However, a local class can only access local variables that are declared final.

Anonymous Classes

- Enable you to declare and instantiate a class at the same time.
- They are like local classes except that they do not have a name.
- Use them if you need to use a local class only once.

Anonymous Class

```
 interface HelloWorld {
    public void greet();
    public void greetSomeone(String someone);
}


 HelloWorld frenchGreeting = new HelloWorld() {
        String name = "tout le monde";
        public void greet() {
            greetSomeone("tout le monde");
        }
        public void greetSomeone(String someone) {
            name = someone;
            System.out.println("Salut " + name);
        }
    };
```

Anonymous Class

- The anonymous class expression consists of the following:
  - The new operator
  - The name of an interface to implement or a class to extend.
  - Parentheses that contain the arguments to a constructor, just like a normal class instance creation expression.
    - Note: When you implement an interface, there is no constructor, so you use an empty pair of parentheses, as in this example.
  - A body, which is a class declaration body. More specifically, in the body, method declarations are allowed but statements are not.

References

- http://math.hws.edu/javanotes/
- https://docs.oracle.com/javase/tutorial/essential/concurrency/
- http://www.beingjavaguys.com/2013/10/inner-class-example-in-java.html