Introduction to **synchronized**

An int variable `x` shared by two threads may be involved in a race condition because the assignment statement

```
x= x + 1;
```

is not an atomic, indivisible, action —it is executed as a sequence of three steps: (1) load `x` into a register, (2) add 1 to the register; (3) store the register into `x`.

Using an analogy, we show how to use keyword **synchronized** to make such a statement into an atomic action.

Locking the outhouse

The first *outhouse*[1] to the right is open. Someone wanting to use the outhouse goes inside, locks the door from the inside, and does their business. At any one time, at most one person is in the outhouse.

Anyone wanting to use the locked outhouse waits outside. Several people may come and wait. They are on what we call the *locklist* of people waiting outside the locked door.

The person inside unlocks the door and comes out. Then there is a free-for-all. The people waiting on the *locklist* fight to get in. One person manages to go inside and locks the door. The others continue to wait.

The synchronized statement

Java keyword **synchronized** is used to implement this process. Suppose `business` is what one does inside object `outhouse`. Then, the statement

synchronized (`outhouse`) { `business` }

is executed exactly as discussed above: Wait until the door of object `outhouse` is open, go inside, lock the door, and execute `business`. When execution is complete, unlock the door and come out.

Synchronizing on a primitive value is not possible; it has to be an object (i.e. a pointer to an object)

An example of the use of synchronized

```
public class X {
    public int x;
}
```

Class `X` to the right is used to wrap an **int** variable `x`, which is initially 0. We can create instance `wrapper` of `X` using

`X wrapper=` **new** `X`();

Suppose several different threads have access to variable `wrapper` and want to change `wrapper.x`, perhaps adding 1 or doubling `x`. These changes must be done atomically in order to avoid race conditions. So, the operations are synchronized like this:

synchronize(`wrapper`) { `wrapper.x= wrapper.x + 1;` }

synchronize(`wrapper`) { `wrapper.x= 2 * wrapper.x;` }

Here, object `wrapper` is the `outhouse`. Inside the `outhouse`, one thread's `business` is to add 1 to `x`; the other thread's `business` is to double `x`.

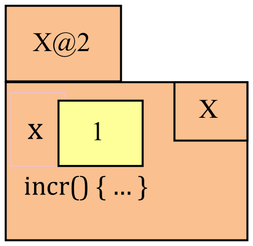
Problem

In what we have written, preventing race conditions is left up to the user. But the user shouldn't have to worry about that. Class `X` take on that task. We show how to do this on the next page.

[1] An outhouse is a separate, small building with a toilet inside. Outhouses were used in many countries well into the second half of the twentieth century. One can still find outhouses in the U.S in some rural areas. The toilet was a hole in a bench with a deep pit underneath. Some outhouses were two-holers —with places for two people to sit at a time. To the right above is an image of Thomas Jefferson's outhouse, taken from Wikipedia. Try as we might, we couldn't get the image into the footnote (using Microsoft Word).

Introduction to **synchronized**

A better way to handle synchronization



To the left is an object of class `X`, showing wrapped variable `x` and only one method, `incr`, which increments `x`. No other thread should be able to change `x` while a thread is executing a call on `incr`, so the body of method `incr` should be synchronized. But on what? *On this object itself*. No other thread should be able to look in this object.

You know that within method `incr`, keyword **this** refers to this object, so it makes sense to synchronize on **this**, using

`synchronize(`**`this`**`)` {…}.

```
public class X {
   private int x= 0;

   public int getX() {return x;}

   public void incr() {
      synchronized(this) {x= x + 1;}
   }

   public synchronized void doubble() {
      x= 2*x;
} }
```

Now look in class `X` to the right (we have removed comments to save space). Field `x` is private, so it can't be referenced outside class `X`. The body of method `incr` synchronizes on **this**, as just discussed. No other thread synchronizing on this object can touch anything in the object while the body of `incr` is being executed.

Synchronizing a method —more syntactic sugar

It's important to realize that the whole body of a method may not have to be synchronized. We should allow as much concurrent execution as possible and synchronize only that part that may otherwise be involved in a race condition.

But often, the whole method body must be synchronized. In that case, *synchronize the method*. Method `doubble` in class `X` (above and to the right) is an example of synchronizing a method. This seems like a new, deep concept, but it's just syntactic sugar. A synchronized method `m`:

public synchronized void `m` (…) {body}

is equivalent to a method `m` with the body synchronized on **this**:

public void `m` (…) {**synchronized(this)** {body}}

Friends get in the back door

Not many people know that your outhouse has an *unlocked* back door, and you tell your best friends that they don't have to wait at the front locked door. You tell them to sneak around the back and go in whenever they want. You tell them to be careful: don't disrupt whatever is going on inside. Be mindful and polite.

In Java, **synchronized** works in the same way, as we now explain.

Suppose variable `wrapper` contains (a pointer to) an object of class `X`, above. Suppose thread `T1` is executing a call `wrapper.incr`(), so that the door to outhouse `wrapper` is locked. Look at method `wrapper.getX`(). Since no synchronization is associated with it, another thread can go in the back door by calling `wrapper.getX`(). That call will return one value, the value of `wrapper.x` before or after thread `T1` increments `wrapper.x`.

Why allow friends to get in the back door? With more than one thread, one wants as much concurrency as possible. The more threads have to wait to get into an outhouse, the less concurrency there is. Therefore, the ability to sneak in the back door can be a good thing.

However, it is *extremely* difficult to prove correctness of a concurrent program (i.e. to understand that a concurrent program is correct) when processes are continually processing shared variables in a non-atomic fashion. Don't use this back-door approach unless you know fully what you are doing. Leave it to the experts.