

An exercise in concurrent programming

- ▶ A narrow North-South bridge can accommodate traffic only in one direction at a time.

An exercise in concurrent programming

- ▶ A narrow North-South bridge can accommodate traffic only in one direction at a time.
- ▶ When a car arrives at the bridge
 1. Cars on the bridge going in the same direction \Rightarrow can cross
 2. No other car on the bridge \Rightarrow can cross (implicitly sets direction)
 3. Cars on the bridge going in the opposite direction \Rightarrow wait for the bridge to be empty

An exercise in concurrent programming

- ▶ A narrow North-South bridge can accommodate traffic only in one direction at a time.
- ▶ When a car arrives at the bridge
 1. Cars on the bridge going in the same direction \Rightarrow can cross
 2. No other car on the bridge \Rightarrow can cross (implicitly sets direction)
 3. Cars on the bridge going in the opposite direction \Rightarrow wait for the bridge to be empty
- ▶ Cars waiting to cross from one side may enter bridge in any order after direction switches in their favour.

An exercise in concurrent programming

- ▶ A narrow North-South bridge can accommodate traffic only in one direction at a time.
- ▶ When a car arrives at the bridge
 1. Cars on the bridge going in the same direction \Rightarrow can cross
 2. No other car on the bridge \Rightarrow can cross (implicitly sets direction)
 3. Cars on the bridge going in the opposite direction \Rightarrow wait for the bridge to be empty
- ▶ Cars waiting to cross from one side may enter bridge in any order after direction switches in their favour.
- ▶ When bridge becomes empty and cars are waiting, yet another car can enter in the opposite direction and makes them all wait some more.

An example . . .

- ▶ Design a class `Bridge` to implement consistent one-way access for cars on the highway synchronization primitives
 - ▶ Should permit multiple cars to be on the bridge at one time (all going in the same direction!)

An example . . .

- ▶ Design a class `Bridge` to implement consistent one-way access for cars on the highway synchronization primitives
 - ▶ Should permit multiple cars to be on the bridge at one time (all going in the same direction!)

- ▶ `Bridge` has a public method

```
public void cross(int id, boolean d, int s)
```

- ▶ `id` is identity of car
- ▶ `d` indicates direction
 - ▶ `true` is `North`
 - ▶ `false` is `South`
- ▶ `s` indicates time taken to cross (milliseconds)

An example . . .

```
public void cross(int id, boolean d, int s)
```

► Method `cross` prints out diagnostics

1. A car is stuck waiting for the direction to change

```
Car 7 going North stuck at Thu Mar 13 23:00:11 IST  
2009
```

2. The direction changes

```
Car 5 switches bridge direction to North at Thu  
Mar 13 23:00:14 IST 2009
```

3. A car enters the bridge.

```
Car 8 going North enters bridge at Thu Mar 13  
23:00:14 IST 2003
```

4. A car leaves the bridge.

```
Car 16 leaves at Thu Mar 13 23:00:15 IST 2003
```

An example . . .

```
public void cross(int id, boolean d, int s)
```

- ▶ Method `cross` prints out diagnostics
 1. A car is stuck waiting for the direction to change
`Car 7 going North stuck at Thu Mar 13 23:00:11 IST 2009`
 2. The direction changes
`Car 5 switches bridge direction to North at Thu Mar 13 23:00:14 IST 2009`
 3. A car enters the bridge.
`Car 8 going North enters bridge at Thu Mar 13 23:00:14 IST 2003`
 4. A car leaves the bridge.
`Car 16 leaves at Thu Mar 13 23:00:15 IST 2003`

- ▶ Use `java.util.Date` to generate time stamps

Analysis

- ▶ The “data” that is shared is the [Bridge](#)

Analysis

- ▶ The “data” that is shared is the `Bridge`
- ▶ State of the bridge is represented by two quantities
 - ▶ Number of cars on bridge — an `int`
 - ▶ Current direction of bridge — a `boolean`

Analysis

- ▶ The “data” that is shared is the `Bridge`
- ▶ State of the bridge is represented by two quantities
 - ▶ Number of cars on bridge — an `int`
 - ▶ Current direction of bridge — a `boolean`
- ▶ The method

```
public void cross(int id, boolean d, int s)
```

changes the state of the bridge

Analysis

- ▶ The “data” that is shared is the `Bridge`
- ▶ State of the bridge is represented by two quantities
 - ▶ Number of cars on bridge — an `int`
 - ▶ Current direction of bridge — a `boolean`
- ▶ The method

```
public void cross(int id, boolean d, int s)
```

changes the state of the bridge

- ▶ Concurrent execution of `cross` can cause problems ...

Analysis

- ▶ The “data” that is shared is the `Bridge`
- ▶ State of the bridge is represented by two quantities
 - ▶ Number of cars on bridge — an `int`
 - ▶ Current direction of bridge — a `boolean`
- ▶ The method

```
public void cross(int id, boolean d, int s)
```

changes the state of the bridge

- ▶ Concurrent execution of `cross` can cause problems ...
- ▶ ...but making `cross` a synchronized method is too restrictive
 - ▶ Only one car on the bridge at a time
 - ▶ Problem description explicitly disallows such a solution

Analysis ...

- ▶ Break up `cross` into a sequence of actions

Analysis . . .

- ▶ Break up **cross** into a sequence of actions
 - ▶ **enter** — get on the bridge
 - ▶ **travel** — drive across the bridge
 - ▶ **leave** — get off the bridge

Analysis . . .

- ▶ Break up `cross` into a sequence of actions
 - ▶ `enter` — get on the bridge
 - ▶ `travel` — drive across the bridge
 - ▶ `leave` — get off the bridge
 - ▶ `enter` and `leave` can print out the diagnostics required

Analysis . . .

- ▶ Break up `cross` into a sequence of actions
 - ▶ `enter` — get on the bridge
 - ▶ `travel` — drive across the bridge
 - ▶ `leave` — get off the bridge
 - ▶ `enter` and `leave` can print out the diagnostics required
- ▶ Which of these affect the state of the bridge?

Analysis . . .

- ▶ Break up `cross` into a sequence of actions
 - ▶ `enter` — get on the bridge
 - ▶ `travel` — drive across the bridge
 - ▶ `leave` — get off the bridge
 - ▶ `enter` and `leave` can print out the diagnostics required
- ▶ Which of these affect the state of the bridge?
 - ▶ `enter` : increment number of cars, perhaps change direction
 - ▶ `leave` : decrement number of cars

Analysis . . .

- ▶ Break up `cross` into a sequence of actions
 - ▶ `enter` — get on the bridge
 - ▶ `travel` — drive across the bridge
 - ▶ `leave` — get off the bridge
 - ▶ `enter` and `leave` can print out the diagnostics required
- ▶ Which of these affect the state of the bridge?
 - ▶ `enter` : increment number of cars, perhaps change direction
 - ▶ `leave` : decrement number of cars
- ▶ Make `enter` and `leave` synchronized

Analysis . . .

- ▶ Break up `cross` into a sequence of actions
 - ▶ `enter` — get on the bridge
 - ▶ `travel` — drive across the bridge
 - ▶ `leave` — get off the bridge
 - ▶ `enter` and `leave` can print out the diagnostics required
- ▶ Which of these affect the state of the bridge?
 - ▶ `enter` : increment number of cars, perhaps change direction
 - ▶ `leave` : decrement number of cars
- ▶ Make `enter` and `leave` synchronized
- ▶ `travel` is just a means to let time elapse — use `sleep`

Analysis ...

Code for `cross`

```
public void cross(int id, boolean d, int s){  
  
    // Get onto the bridge (if you can!)  
    enter(id,d);  
  
    // Takes time to cross the bridge  
    try{  
        Thread.sleep(s);  
    }  
    catch(InterruptedException e){}  
  
    // Get off the bridge  
    leave(id);  
}
```

Analysis . . .

Entering the bridge

- ▶ If the direction of this car matches the direction of the bridge, it can enter

Analysis . . .

Entering the bridge

- ▶ If the direction of this car matches the direction of the bridge, it can enter
- ▶ If the direction does not match but the number of cars is zero, it can reset the direction and enter

Analysis . . .

Entering the bridge

- ▶ If the direction of this car matches the direction of the bridge, it can enter
- ▶ If the direction does not match but the number of cars is zero, it can reset the direction and enter
- ▶ Otherwise, `wait()` for the state of the bridge to change

Analysis . . .

Entering the bridge

- ▶ If the direction of this car matches the direction of the bridge, it can enter
- ▶ If the direction does not match but the number of cars is zero, it can reset the direction and enter
- ▶ Otherwise, `wait()` for the state of the bridge to change
- ▶ In each case, print a diagnostic message

Code for enter

```
private synchronized void enter(int id, boolean d){
    Date date;

    // While there are cars going in the wrong direction
    while (d != direction && bcount > 0){

        date = new Date();
        System.out.println("Car "+id+" going "+direction_name(d)+"

        // Wait for our turn
        try{
            wait();
        }
        catch (InterruptedException e){}
    }

    ...

}
```

Code for `enter`

```
private synchronized void enter(int id, boolean d){
    ...
    while (d != direction && bcount > 0){ ... wait() ...}
    ...

    // Switch direction, if needed
    if (d != direction){
        direction = d;
        date = new Date();
        System.out.println("Car "+id+" switches bridge direction
            to "+direction_name(direction)+" at "+date);
    }

    // Register our presence on the bridge
    bcount++;

    date = new Date();
    System.out.println("Car "+id+" going "+direction_name(d)+"
        enters bridge at "+date);
}
```

Analysis . . .

Leaving the bridge is much simpler

- ▶ Decrement the car count

Analysis . . .

Leaving the bridge is much simpler

- ▶ Decrement the car count
- ▶ `notify()` waiting cars

Analysis ...

Leaving the bridge is much simpler

- ▶ Decrement the car count
- ▶ `notify()` waiting cars
... provided car count is zero

Analysis ...

Leaving the bridge is much simpler

- ▶ Decrement the car count
- ▶ `notify()` waiting cars
... provided car count is zero

```
private synchronized void leave(int id){
    Date date = new Date();
    System.out.println("Car "+id+" leaves at "+date);

    // "Check out"
    bcount--;

    // If everyone on the bridge has checked out, notify the
    // cars waiting on the opposite side
    if (bcount == 0){
        notifyAll();
    }
}
```