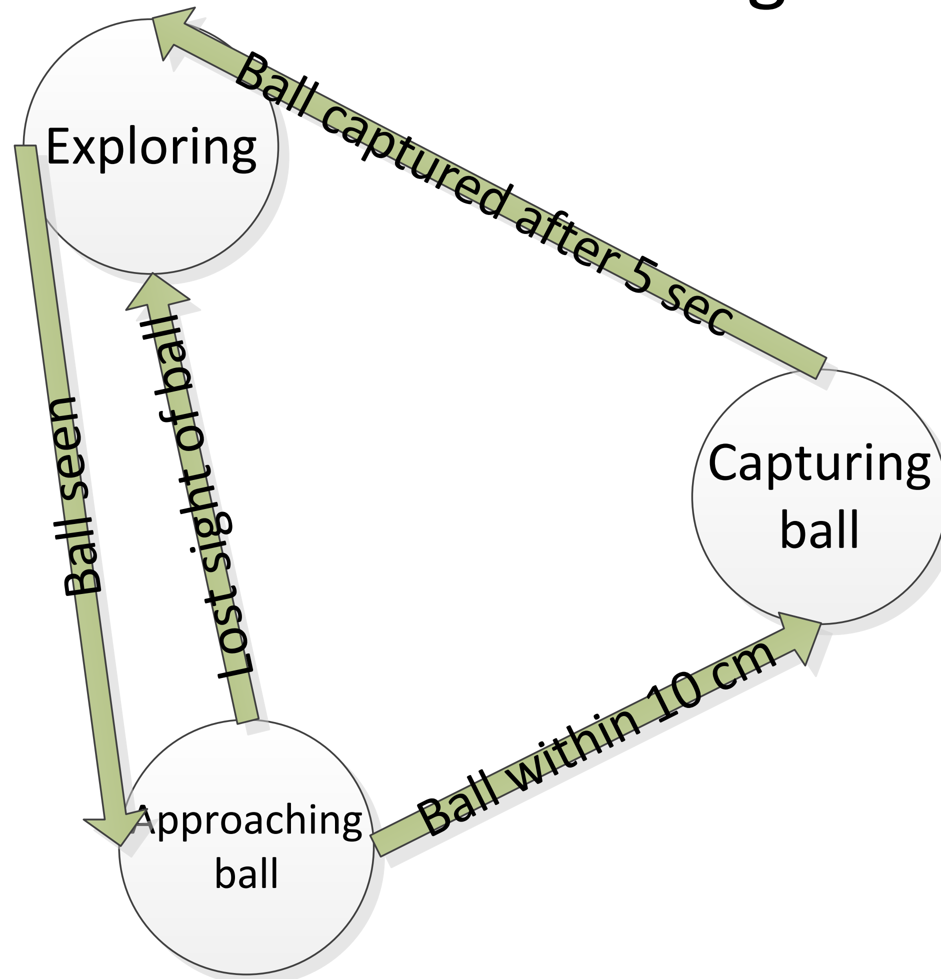
Software and Threading

Geza Kovacs

Maslab 2011

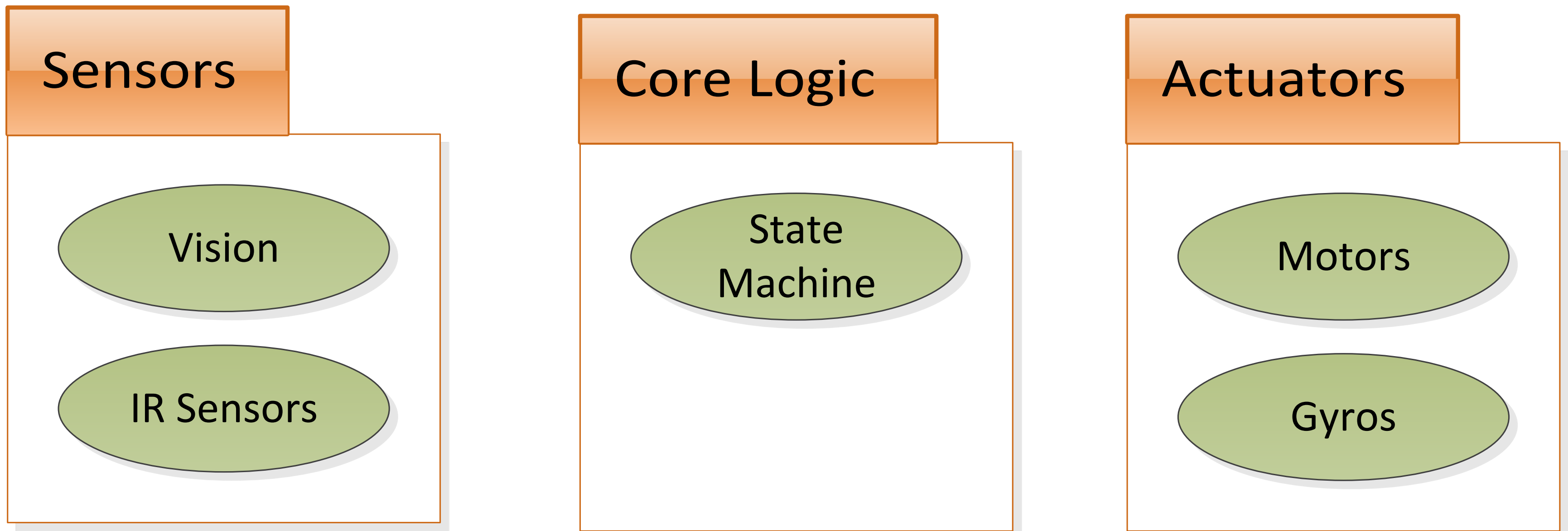
Abstract Design: State Machines

- By using state machine diagrams, you can find flaws in your behavior without needing to implement it



Modular Design

- Split independent parts of your application into modules (packages, classes)
- Each module can be independently implemented and tested

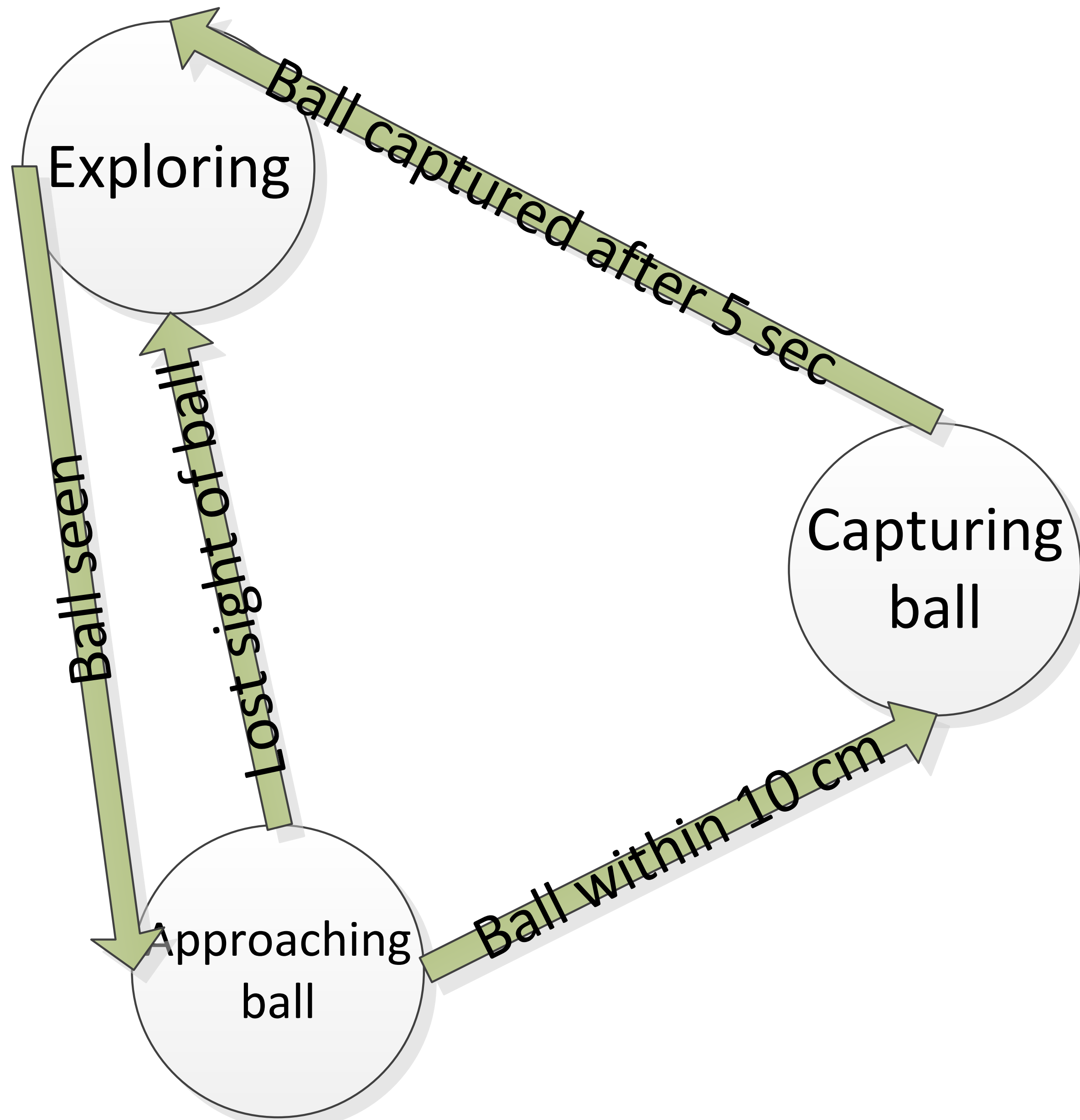


Agree on Specifications Before Coding

```
class BallPosition {
    /** angle of ball relative to the camera,
     *  in radians. + is right, - is left */
    double angle;
    /** distance in cm, from camera to the ball */
    double distance;
}

abstract class Vision { // image processing code
    /** ball positions from nearest to furthest.
     *  empty if no balls are detected */
    BallPosition[] ballLocations;
    /** captures image, detects balls
     * and populates ballLocations */
    abstract void detectBalls();
}
```

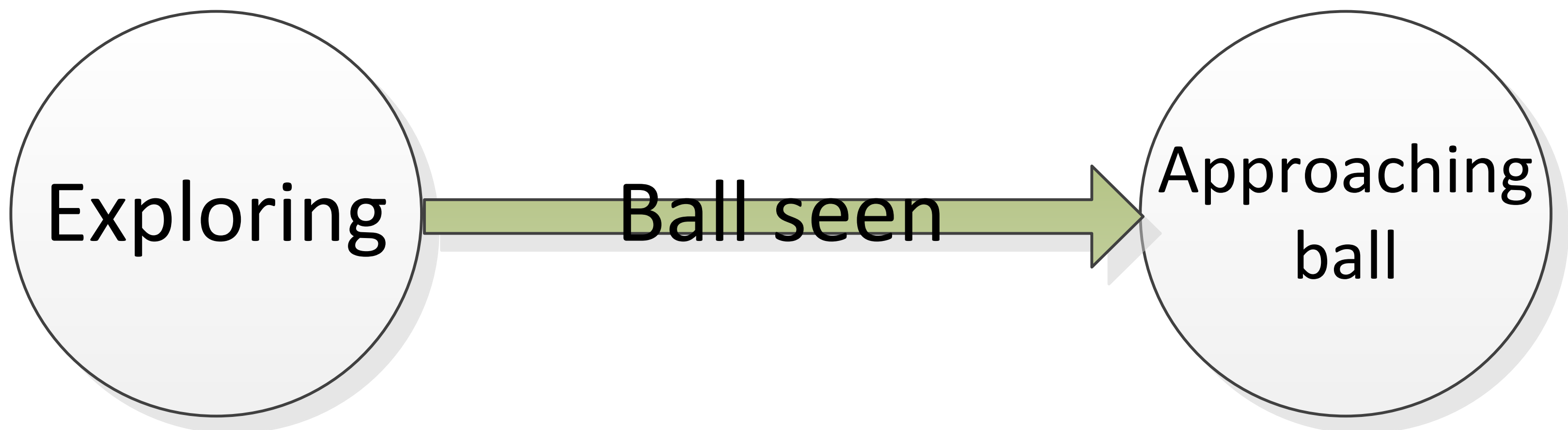
From State Machines to Java Code



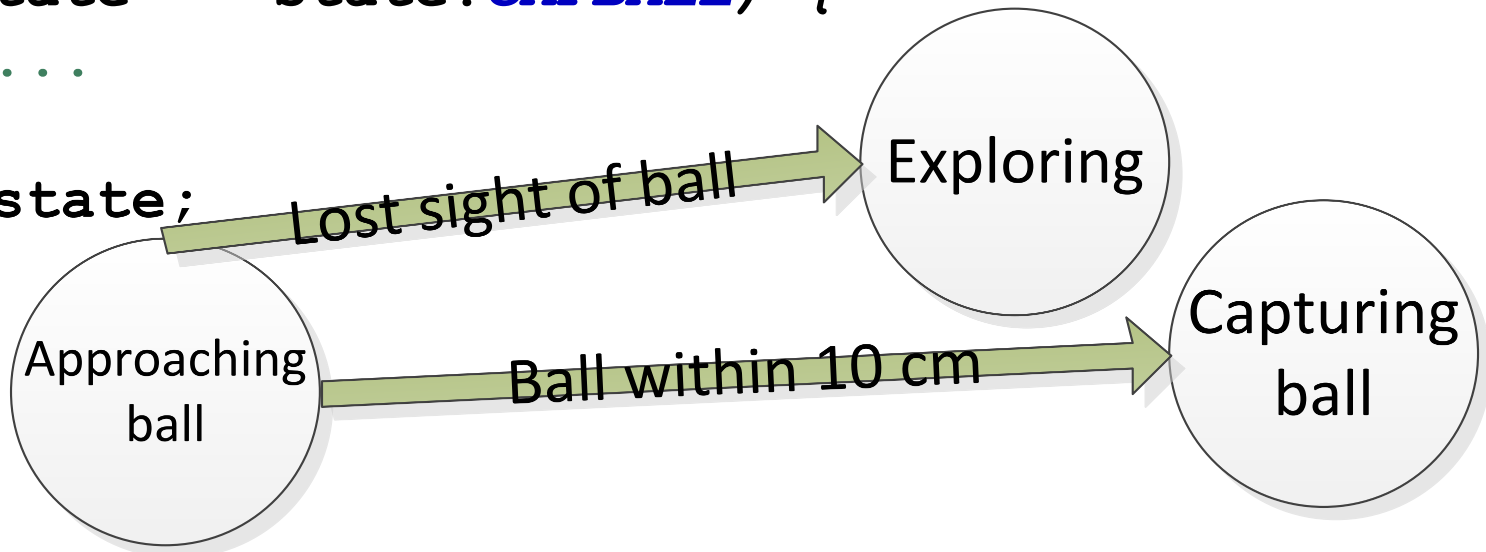
```java
enum State {
    EXPLORE,TOBALL,CAPBALL
}

class RobotSM { // state machine
    Vision vis;
    Actuator act;
    long captureBallStartTime = 0;
/** @param state Current state of state machine
 *  @return Next state of state machine */
    State nextState(State state) {…}
    void runSM() {
        State state = State.EXPLORE;
        while (true) {
            vis.detectBalls();//capture+process image
            state = nextState(state);
        }
    }
}
```

```java
State nextState(State state) {
    if (state == State.EXPLORE) {
        if (vis.ballLocations.length > 0)
            state = State.TOBALL;
        else
            act.rotateLeft();
    } if (state == State.TOBALL) {
        // ...
    } if (state == State.CAPBALL) {
        // ...
    }
    return state;
}
```



```
State nextState(State state) {
    if (state == State.EXPLORE) {
        // ...
    } if (state == State.TOBALL) {
        if (vis.ballLocations.length == 0)
            state = State.EXPLORE;
        else if (vis.ballLocations[0].distance < 10.0)
            captureBallStartTime = System.currentTimeM
            state = State.CAPBALL;
        } else
            act.moveForward(vis.ballLocations[0].angle
    } if (state == State.CAPBALL) {
        // ...
    }
    return state;
}
```

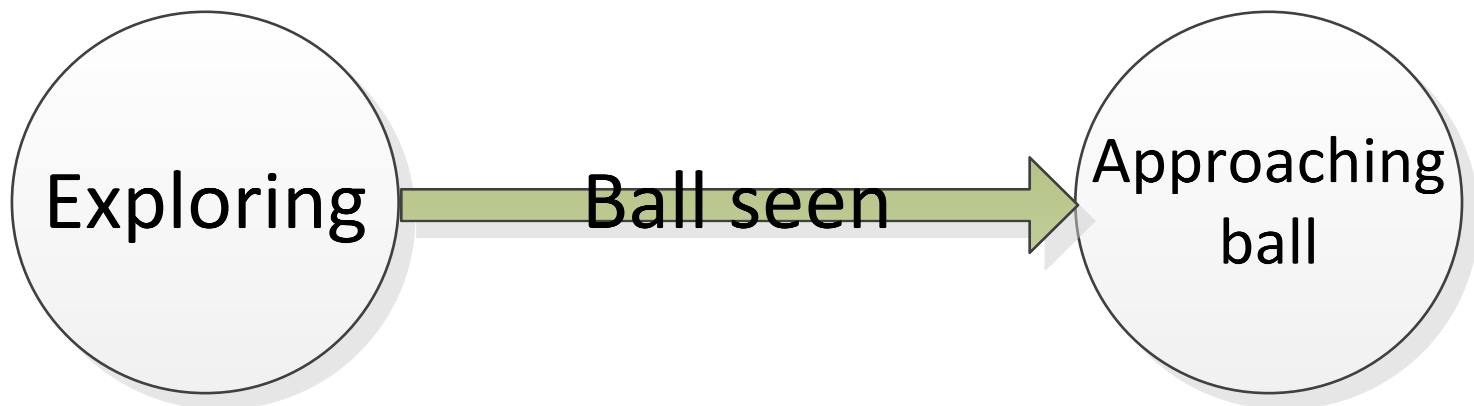


```
State nextState(State state) {
    if (state == State.EXPLORE) {
        // ...
    } if (state == State.TOBALL) {
        // ...
 } if (state == State.CAPBALL) {
        if (System.currentTimeMillis() >
captureBallStartTime + 5000) {
            act.stopCaptureBall();
            state = State.EXPLORE;
        } else
            act.captureBall();
    }
    return state;
}
```

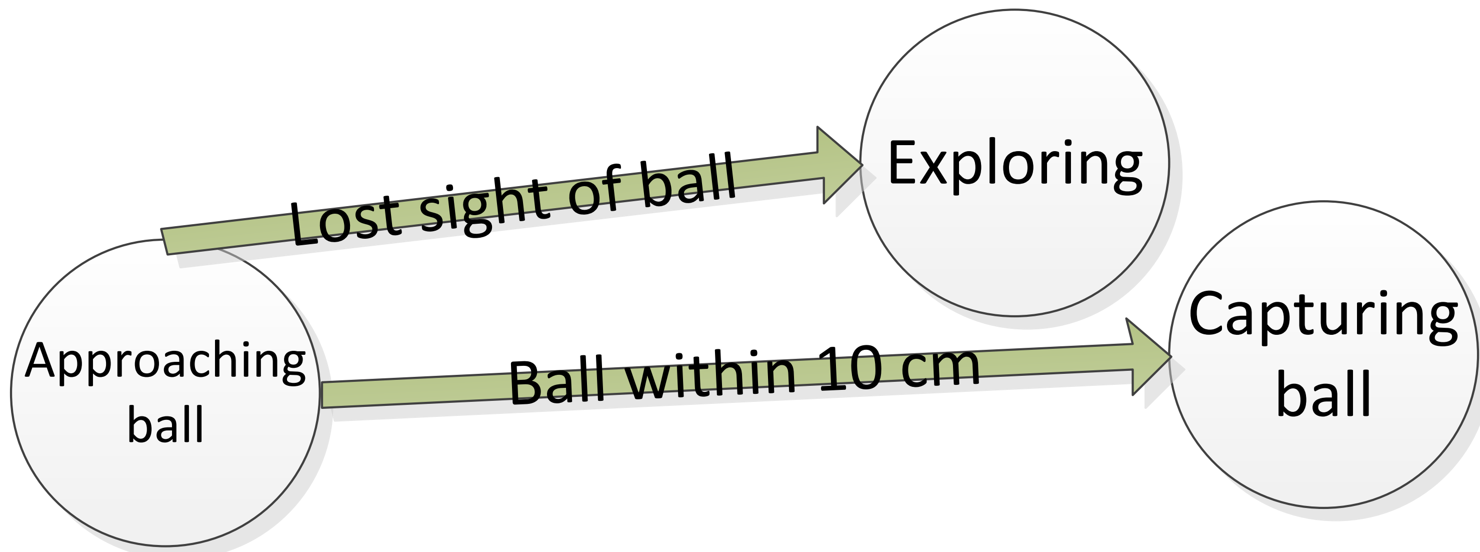
- Alternatively, you can represent states as objects

```
interface IState {
    IState nextState();
}
class RobotSM2 {
    Vision vis;
    Actuator act;
    void runSM() {
        IState state = new ExploreState(this);
        while (true){
            vis.detectBalls();
            state = state.nextState();
        }
    }
}
```

```java
class ExploreState implements IState {
    private RobotSM2 sm;
    ExploreState (RobotSM2 sm) {
        this.sm = sm;
    }
    public IState nextState() {
        if (sm.vis.ballLocations.length > 0)
            return new ToBallState(sm);
        else {
            sm.act.rotateLeft();
            return this;
        }
    }
}
```



```java
class ToBallState implements IState {
    private RobotSM2 sm;
    ToBallState (RobotSM2 sm) {
        this.sm = sm;
    }
    public IState nextState() {
        if (sm.vis.ballLocations.length == 0)
            return new ExploreState(sm);
        else if (sm.vis.ballLocations[0].distance < 10.0) {
            return new CapBallState(sm, System.currentTimeMi
        } else {
            sm.act.moveForward(sm.vis.ballLocations[0].angle
            return this;
        }
    }
}
```



```java
class CapBallState implements IState {
    private RobotSM2 sm;
    private long captureBallStartTime;
    CapBallState (RobotSM2 sm, long captureBallStartTime) {
        this.sm = sm;
        this.captureBallStartTime = captureBallStartTime;
    }
    public IState nextState() {
        if (System.currentTimeMillis() >
captureBallStartTime + 5000) {
            sm.act.stopCaptureBall();
            return new ExploreState(sm);
        } else {
            sm.act.captureBall();
            return this;
        }
    }
}
```

Unit Testing

- Ensures your code does what you think it does
  - Saves you debugging time later on
- Prevents regressions
  - Ensures that performance tweaks don't break your code
- JUnit: Unit testing for Java
- Denote a method as a test method with the **@Test** annotation, use **Assert.assertEquals()** , **Assert.assertTrue()**, etc, to do tests
- Run via Eclipse's GUI (Run -> Run As -> JUnit Test) or with **java org.junit.runner.JUnitCore TestClass.class**

Writing Unit Tests with JUnit

```java
import org.junit.*;

public class RobotSMTests {
    @Test
    public void testTransitions() {
        RobotSM sm = new RobotSM();
        sm.vis = new Vision() {
            void detectBalls() {}
        };
        sm.vis.ballLocations = new BallPosition[] {
            new BallPosition(1.0, 0.0),
        };
        State expected = State.TOBALL;
        State actual = sm.nextState(State.EXPLORE);
        Assert.assertEquals(expected, actual);
    }
}
```

Writing Unit Tests with JUnit

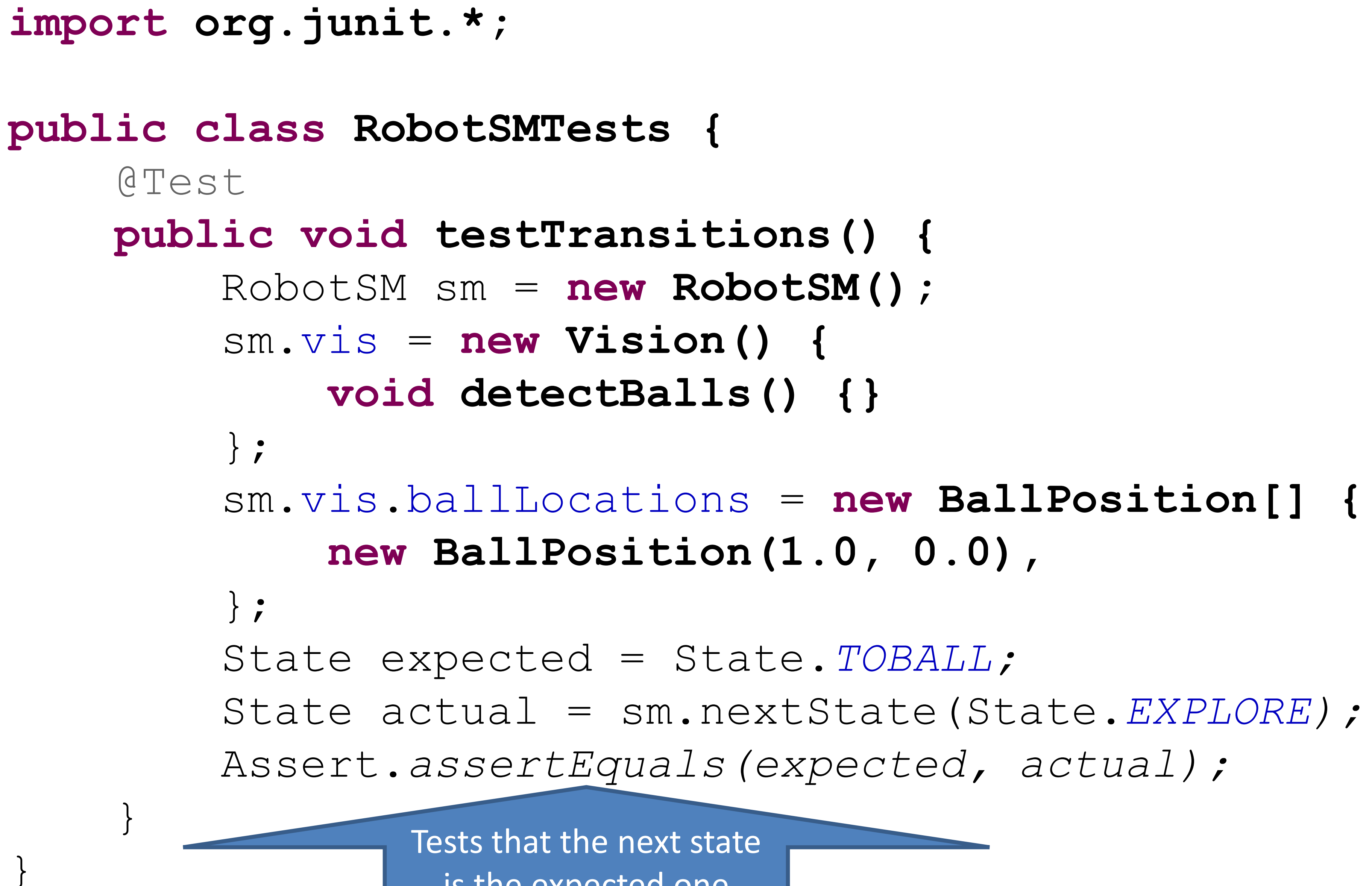
```
import org.junit.*;

public class RobotSMTests {
    @Test
    public void testTransitions() {
        RobotSM sm = new RobotSM();
        sm.vis = new Vision() {
            void detectBalls() {}
        };
        sm.vis.ballLocations = new BallPosition[] {
            new BallPosition(1.0, 0.0),
        };
        State expected = State.TOBALL;
        State actual = sm.nextState(State.EXPLORE);
        Assert.assertEquals(expected, actual);
    }
}
```

Vision code doesn't need to be implemented to test the state machine!

Simulates detection of a ball

Writing Unit Tests with JUnit



Revision Control – svn

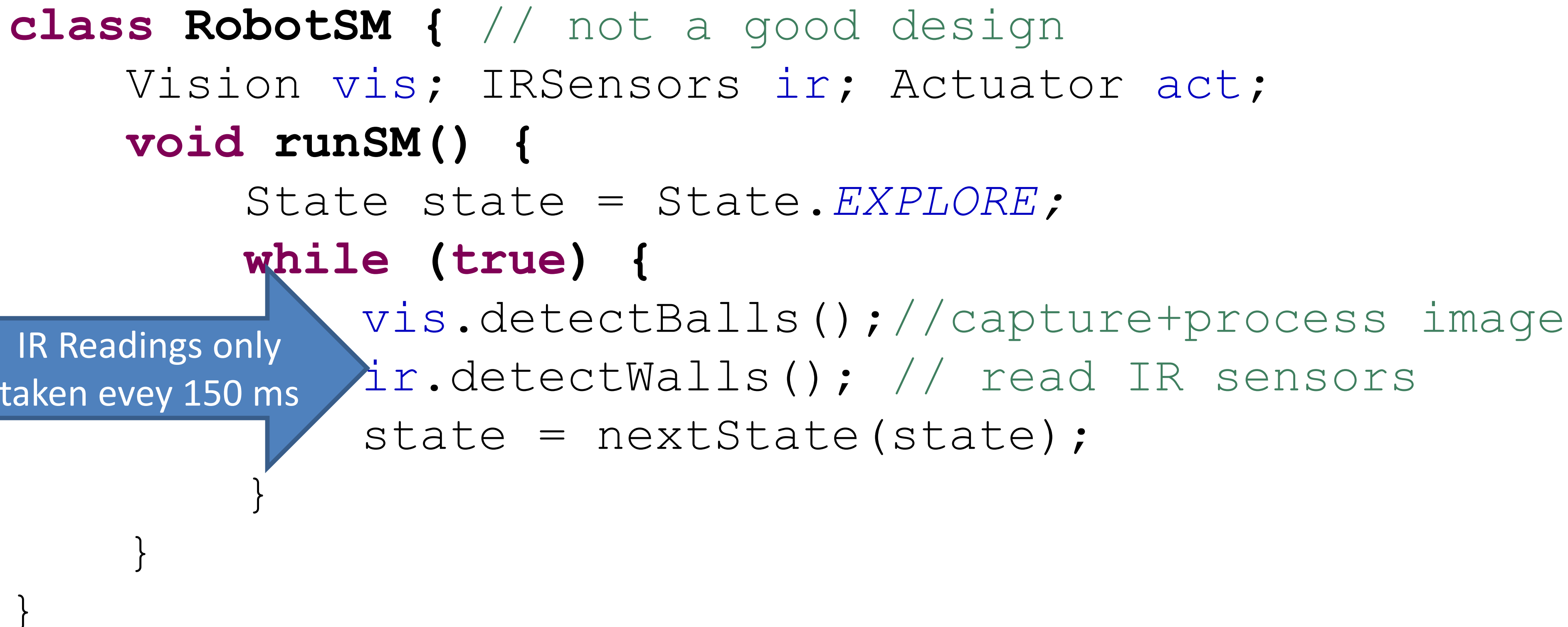
- Keeps track of the history of changes to your code, and keeps it backed up it safely on a server
- Lets you collaborate with others.
- Did something recently stop working? Revert to a working revision, and see what changed.
- Commit often, but don’t commit broken code
- Subclipse: graphical svn integration for Eclipse http://subclipse.tigris.org/

Other Useful Resources

- Java API docs: http://download.oracle.com/javase/6/docs/api/
- Maslab API docs: http://web.mit.edu/maslab/2011/doc/maslab/api/
- uORC API docs: http://web.mit.edu/maslab/2011/doc/uorc/api/
- Eclipse: Code editing, JUnit integration, debugger, etc
- BotClient: Displaying info from your robot
- OrcSpy: Testing your orcboard
- SSH: Uploading code to the robot
- http://maslab.mit.edu/2011/wiki/Software

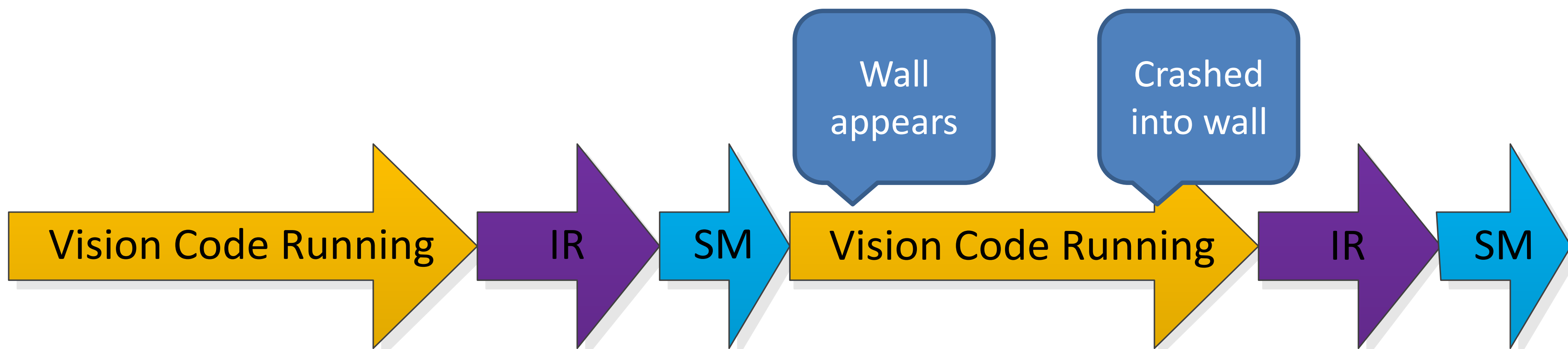
Using Multiple Sensors

- Readings from different sensors may be taken and processed at different frequencies
  - IR: every few ms
  - Vision: more CPU intensive, every 150 ms

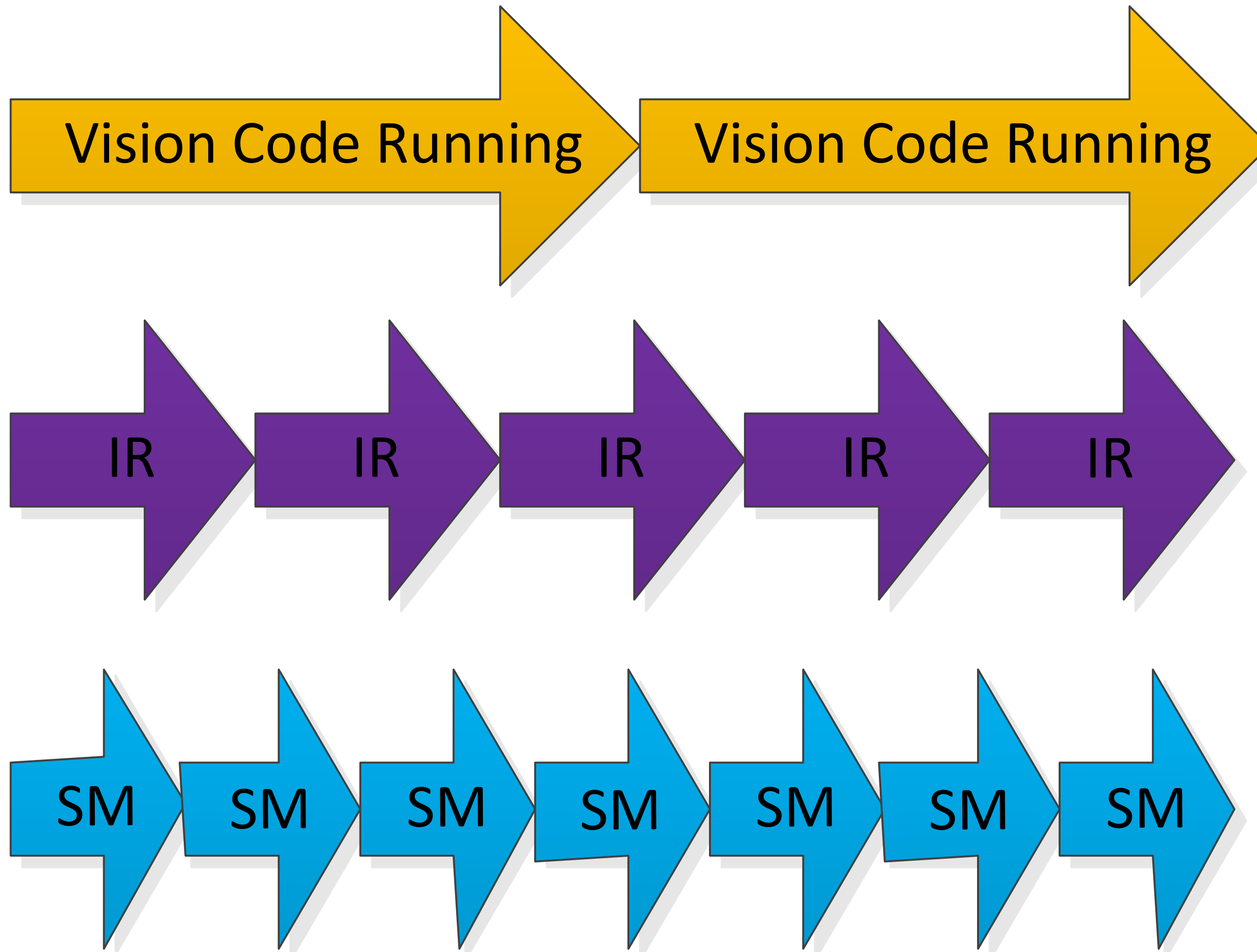


Problem with Sequential Execution

- Robot cannot capture or react to data from other sensors (like IR) while long, CPU-intensive tasks like image processing are running



What we want (Parallel Execution)



What are Threads?

- Can do tasks in parallel
  - Does this essentially by having the CPU swap between tasks (frequently yet unpredictably)
- Threads can access the same region of memory
  - If only one thread is writing to that region of memory (and others are reading from it) -> simple
  - If multiple threads need to write to the same region of memory -> more complex

- To make code run in a separate thread, implement the **Runnable** interface, and add a **run()** method

```
abstract class Vision implements Runnable {
    BallPosition[] ballLocations;
    abstract void detectBalls();
    public void run() {
        while (true) {
            detectBalls();
        }
    }
}
```

- Use **Thread.yield()** to let other threads run (to avoid one thread hogging all the CPU time)

```
abstract class Vision implements Runnable {
    BallPosition[] ballLocations;
    abstract void detectBalls();
    public void run() {
        while (true) {
            detectBalls();
            Thread.yield();
        }
    }
}
```

- Start threads with **new Thread(Runnable).start()**
- Program exits when all threads terminate
  - Note that, as written, this program never exits because it never terminates its threads

```
public class Main {
    public static void main(String[] args) {
        RobotSM sm = new RobotSM();
        sm.vis = new VisionImpl();
        sm.ir = new IRSensorsImpl();
        sm.act = new ActuatorImpl();
        new Thread(sm).start();
        new Thread(sm.vis).start();
        new Thread(sm.ir).start();
    }
}
```

- A thread terminates when **run()** returns
- Ensure that threads can be terminated by other threads
- Mark variables written to by other threads as **volatile** to ensure compiler doesn't optimize them out

```
abstract class Vision implements Runnable {
    BallPosition[] ballLocations;
    abstract void detectBalls();
    volatile boolean running = true;
    public void run() {
        while (running) {
            detectBalls();
            Thread.yield();
        }
    }
}
```

- Thread can be suspended with **Thread.sleep(millis)**
- This application runs for 3 minutes then exits

```
public class Main {
    public static void main(String[] args) throws
        RobotSM sm = new RobotSM();
        sm.vis = new VisionImpl();
        sm.ir = new IRSensorsImpl();
        sm.act = new ActuatorImpl();
        new Thread(sm).start();
        new Thread(sm.vis).start();
        new Thread(sm.ir).start();
        Thread.sleep(180000); // wait 3 minutes
        sm.running = false;
        sm.vis.running = false;
        sm.ir.running = false;
    }
}
```

Writes Across Threads

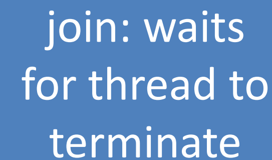
- In example so far, many threads read a given variable, but only a single thread writes to it
- What if many threads write to a single variable?

```java
class Counter {
    int value = 0;
    void increment() { ++value; }
    void decrement() { --value; }
}

class Main { // sequential version
    public static void main(String[] args) {
        final Counter c = new Counter();
        for (int i = 0; i < 1000; ++i) {
            c.increment();
        }
        for (int i = 0; i < 1000; ++i) {
            c.decrement();
        }
        System.out.println(c.value); // always 0
    }
}
```

- Sequential version: Result after 1000 increments and 1000 decrements is 0, as expected

```java
class Main { // multi-threaded version, doesn't (yet) work
    public static void main(String[] args) throws Interrupte
        final Counter c = new Counter();
        Thread a = new Thread(new Runnable() {
            public void run() {
                for (int i = 0; i < 1000; ++i) {
                    c.increment();
                    Thread.yield();
                }
            }
        });
        Thread b = new Thread(new Runnable() {
            public void run() {
                for (int i = 0; i < 1000; ++i) {
                    c.decrement();
                    Thread.yield();
                }
            }
        });
        a.start(); b.start(); a.join(); b.join();
        System.out.println(c.value); // not always 0!
    }
}
```



Problem with Multi-threaded Counter

- The following sequence of operations might occur:
  - Counter's value is 0
  - Thread a gets counter value (0)
  - Thread b gets counter value (0)
  - Thread a writes back incremented value (1)
  - Thread b writes back decremented value (-1)
- Final counter value is -1!

Problem with Multi-threaded Counter

- This sequence is also possible (remember, threads interleave operations in unpredictable order)
  - Counter's value is 0
  - Thread a gets counter value (0)
  - Thread b gets counter value (0)
  - Thread b writes back decremented value (-1)
  - Thread a writes back incremented value (1)
- Final counter value is 1!

Want: Atomic operations

- We want to get and increment, or get and decrement the counter without having it be written to by another thread in the meantime
  - Counter’s value is 0
  - Thread a gets counter value (0) and writes back incremented value (1)
  - Thread b gets counter value (1) and writes back decremented value (0)
- Or:
  - Counter’s value is 0
  - Thread b gets counter value (0) and writes back decremented value (-1)
  - Thread a gets counter value (-1) and writes back incremented value (0)

- **synchronized methods** are one way of accomplishing this this, since only one synchronized method of an instance can run at once.

```
class SynchronizedCounter { // thread-safe versio
    int value = 0;
    synchronized void increment() {
        ++value;
    }
    synchronized void decrement() {
        --value;
    }
}
```

- **synchronized blocks** are another way, which requires an object instance to be “locked” as the block of code is entered. This prevents other code that is synchronized on that object from executing.

```
class SynchronizedCounter { // thread-safe versio
    int value = 0;
    void increment() {
        synchronized (this) {      // Counter instance locked
            ++value;
        }                          // Counter instance unlocked
    }
    void decrement() {
        synchronized (this) {      // Counter instance locked
            --value;
        }                          // Counter instance unlocked
    }
}
```

Thread-safe objects

- The SynchronizedCounter example is **thread-safe**. That is, multiple threads can call its methods and they will still behave as expected
  - Immutable objects like String are inherently thread-safe
  - Some mutable objects, like BlockingQueue, have been made thread-safe
- Others, like ArrayList, are not thread-safe
- If modifying an object from multiple threads, read its specifications to see whether it's thread-safe
  - If not, add synchronized statements in your own code

Not thread-safe

External synchronization

External synchronization

```
class Main { // multi-threaded version, works
    public static void main(String[] args) throws Interrupte
        final Counter c = new Counter();
        Thread a = new Thread(new Runnable() {
            public void run() {
                for (int i = 0; i < 1000; ++i) {
                    synchronized (c) {c.increment();}
                    Thread.yield();
                }
            }
        });
        Thread b = new Thread(new Runnable() {
            public void run() {
                for (int i = 0; i < 1000; ++i) {
                    synchronized (c) {c.decrement();}
                    Thread.yield();
                }
            }
        });
        a.start(); b.start(); a.join(); b.join();
        System.out.println(c.value);
    }
}
```

Final Notes on Threading

- Do:
  - Avoid writing to objects from multiple threads if possible
  - Make the objects thread-safe, or synchronize externally, if you must
- Don't
  - Synchronize everything (makes your code execute sequentially, and might even lead to deadlocks)
- For further info, see Java's Concurrency Tutorial: http://download.oracle.com/javase/tutorial/essential/concurrency/