



Chapter #3: Programming the J-Bot to Go Places

Chapter #3: Programming the J-Bot to Go Places

3

Chapter #3 is all about instructing the J-Bot where to go and how to get there. You'll write programs to make the J-Bot perform a variety of maneuvers instead of just going forward and backward. Some programs

can be used for navigating tight spaces, others for drawing shapes. Whatever the maneuver, this chapter presents the tools for programming the J-Bot to perform it. Here's what you'll learn how to do in Chapter #3:

- Move a specific distance forward and backward.
- Pivot and make turns.
- Program your J-Bot to go a variety of directions, all in the same program.
- Learn about the J-Bot class hierarchy used in the rest of this book (see Activity #3)
- Create a Java class to handle basic maneuvers
- Write programs that remember long lists of movement instructions.
- Write programs that make the J-Bot accelerate and decelerate during maneuvers.

The exercises in this chapter also offer lots of practice in using variables and flow control to accomplish a variety of tasks. Some essential math for converting program commands into distance and speed are also introduced. For some, this will be a first glimpse into elementary Dynamics.

Converting Instructions to Motion

In the previous chapter, you programmed the J-Bot to move forward and backward. Additionally, software calibration settings were determined for programming the J-Bot to move forward, backward, and to stop and stay still.

Each J-Bot navigation program in Chapter #2 focused on one direction. If the J-Bot was programmed to go forward, it had to be reprogrammed to go backward and reprogrammed again to turn in place. In this chapter, all the directions will be incorporated into a single program. By determining how many pulses it takes to make the J-Bot rotate a certain amount during a turn, you can program the J-Bot to perform a variety of more precise maneuvers. For example, the J-Bot can be programmed to draw a square, or a cross, or a triangle.

This level of programmed maneuverability is well and good, but programming long and involved lists can become a complicated

problem. Java also features a simple and efficient method of recording and accessing long lists of directions in the program memory. You'll notice that while the J-Bot is performing its programmed maneuvers that it comes to abrupt stops when it changes direction. Commands can also be added to make the J-Bot decelerate into and accelerate out of direction changes. This *ramping* process will solve the abrupt stops and extend the life of the J-Bot's servos.

FYI

The J-Bot will be able to run all the programs in this book when it is powered by the external power supply that plugs into a wall outlet. In some cases, the J-Bot will not operate properly when using battery power. This can be due to a combination of things from improper batteries, such as rechargeable nickel-cadmium (NiCd) batteries, or partially discharged batteries that cannot supply a sufficient current surge to the J-Bot when the servos start at top speed. In this case, the ramping support described at the end of chapter should be employed as it minimizes power surge. The ramping support will not address batteries that cannot supply sufficient current to run the J-Bot processor. In this case, new batteries will be required.

When programming the J-Bot, the goal is often to make it move a specific distance or to execute a particular turn. It is helpful to know how to figure out how far the J-Bot will travel or turn when it is given a specific command. Circumference is equal to pi (π) multiplied by the wheel diameter:

$$\text{circumference} = \pi \times \text{wheel diameter}$$

$$\text{circumference} = 3.14159 \times 6.67 \text{ cm} \approx 21 \text{ cm}$$

So, now we know that with one complete turn of the wheels, the J-Bot travels about 21 cm.

One way to determine how far the J-Bot will move is to determine how fast it moves. We can then calculate how far it will move. For example, if the servo turns at about 37.5 revolutions per minute (RPM), or 0.625 revolutions/sec then the speed would be about:

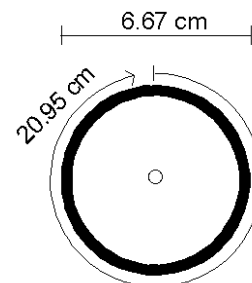


Figure 3.1: Wheel Diameter and Circumference

$$21 \text{ cm/revolution} \times 0.625 \text{ revolutions/sec} = 13.125 \text{ cm/s}$$

The time it takes to make the J-Bot travel 50 cm is:

$$t_{\text{travel}} = 50 \text{ cm} \div 13.125 \text{ cm/s} \approx \text{about 3.81 seconds}$$

Unfortunately, determining the rotational speed of the wheel can be somewhat difficult unless you have very good eyes or a light strobe. You can attempt to calculate the speed of rotation but we have an easier way of getting the J-Bot to move the distance we want.

The approach taken in Activity #1 measures linear distance instead of rotational distance. This is easy to measure with a ruler or tape measure. It also takes into account other factors such as possible slippage for a particular surface. Normally this should be very small but it depends on the surface you have available.

You can always compute the rotational speed from the results of Activity #1 if you want.

Activity #1: Maneuvers - Going the distance

Chapter 2 introduced wheel servo control for moving the J-Bot. The problem is that the J-Bot ran for a fixed amount of time. This is fine if you don't care how far the J-Bot moves but not if you want the J-Bot to perform in a predictable fashion. In this activity we move the J-Bot forward based on the desired number of inches to move instead of the number of seconds. This should prove move useful when we want the J-Bot to move in a square assuming we can get it to pivot in Activity #2.

First we need to figure out how long to run the J-Bot so it moves a fixed distance. Doing this could take a lot of trial and error because we do not know the relationship between the J-Bot's PWM frequency and a particular speed. If we did then it takes just a little math to compute distance moved for a given amount of time at a specified speed.

Before diving into making measurements we should take a look at a problem you may have encountered in the previous chapter. In particular, running the J-Bot with the serial cable attached. This is fine for short distances but impractical for longer distances because the cable can induce drag or even act as a tether preventing the J-Bot from moving as far as it is programmed to. To this end, the application in this activity is designed so that the J-Bot will not be tethered. This configuration also means that the J-Bot must be running on batteries.

The free roaming J-Bot operation works this way. First, you download the program when the J-Bot is connected to the PC using the serial cable. The power adapter can be used instead of battery power at this time. Second, the power and serial cables are removed. At this point the J-Bot is off and will not move. The program that was downloaded is contained in the flash memory of the Javelin Stamp. This information is retained even when power is removed. Third, the J-Bot is moved to where it will be tested. Fourth, battery power is applied by plugging the battery power cable into the J-Bot's circuit board. Hit the reset switch if necessary since the program will start running immediately. The reset switch starts it all over.

To start, enter the `BasicWheelServoDistanceTest1.java` program shown next.

```
import stamp.core.*;
import JBot.* ;

/**
 * Wheel servo distance test program
 * <p>
 * The program runs the J-Bot for fixed time periods.
 * Each time period is longer than the prior one.
 * A delay is included before each test run for measurements.
 * Distance measurements can be taken between each movement.
 *
 * @version 1.0 7/23/02
 * @author Parallax Inc.
 */

public class BasicWheelServoDistanceTest1 {
    public static void main () {
        BasicWheelServo leftWheel =
            new BasicWheelServo (
                CPU.pin13    // pin
                , 240         // forward
                , 175         // center
                , 110         // backward
                , 2000        // low
            ) ;
        BasicWheelServo rightWheel =
            new BasicWheelServo (
                CPU.pin12    // pin
                , 110        // forward
                , 175        // center
                , 240        // backward
                , 2000        // low
            ) ;

        for ( int i = 1 ; i <= 10 ; ++i ) {
            CPU.delay(20000);    // initial delay for measurements

            leftWheel.move ( 100 ) ;
            rightWheel.move ( 100 ) ;

            for ( int j = 0 ; j < i ; ++ j ) {
                CPU.delay(5000);    // run servos for fixed amount of time
            }
        }
    }
}
```

```

        leftWheel.stop () ;    // stop wheels
        rightWheel.stop () ;
    }
}
}

```

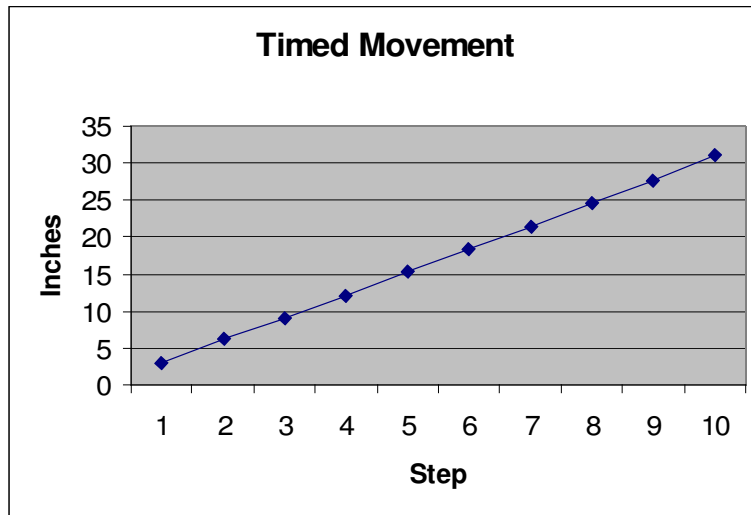
The BasicWheelServoDistanceTest1.java program will run the J-Bot forward for ten times. The time the J-Bot runs for each iteration is one timer interval greater than the last. You will need an area large enough to let the J-Bot run for this test. A hallway or large room works nicely. The top of a table will probably be insufficient. We present an alternative method for running the J-Bot with this program if a large area is not available.

For this test you will need some sort of small markers. Square pieces of paper about one square inch on a side are ideal. This is how the experiment works.

1. Load the J-Bot with the BasicWheelServoDistanceTest1.java program.
2. Place one marker on the ground next to the J-Bot. The marker, and subsequent markers, should be placed near the same spot on the J-Bot. The center of the J-Bot's wheel on either side is an easy spot to locate consistently.
3. Add power.
4. The J-Bot should wait two seconds before it starts moving so you have time to aim it in the proper direction.
5. The J-Bot rolls forward for its first timed movement. Place another marker next to the J-Bot.
6. Repeat steps 4 and 5 until all iterations are done.
7. Measure the distance between markers and put the data in the following table. The distance can be in inches (to the closest tenth or quarter inch) or millimeters depending upon the kind of ruler available.

Iterat ion	Time	Distance (in.)	Time/inch
1	5000	3.12	1602
2	10000	6.25	1600
3	15000	9.00	1666
4	20000	12.10	1652
5	25000	15.25	1639
6	30000	18.30	1639
7	35000	21.45	1631
8	40000	24.60	1626
9	45000	27.75	1621
10	50000	31.10	1607

Next we graph the results. The graph will look something like the following. This can be done easily using a spreadsheet program.



Don't worry if the J-Bot does not go in a straight line. The servo settings can be adjusted to improve this but finish testing before making major changes. The J-Bot movement should be relatively consistent as shown in the plot but there is some variation. This is due to a number of factors including startup and stopping times. Shorter movements will be affected more by this factor than longer movements. Minor deviations in the servos, wheel and servo alignment, and friction between the wheels and the surface the J-Bot is running on all contribute to movement deviations.

The program should be run multiple times to make sure the results are consistent. The results may be averaged for the next step that is the development of a wheel servo class that moves a fixed distance.

Using the results from the table and plot we get an average time of 1628 to move one inch. The following program turns the time around so the J-Bot moves a fixed number of inches.

```
import stamp.core.*;
import JBot.* ;

/**
 * Wheel servo distance test program
 * <p>
 * The program runs the J-Bot for a fixed number of inches.
 * Distance measurements can be taken between each movement.
 *
 * @version 1.0 7/23/02
 * @author Parallax Inc.
 */
```

```

public class BasicWheelServoDistanceTest2 {
    public static void main () {
        BasicWheelServo leftWheel =
            new BasicWheelServo (
                CPU.pin13 // pin
                , 240      // forward
                , 175      // center
                , 110      // backward
                , 2000     // low
            ) ;
        BasicWheelServo rightWheel =
            new BasicWheelServo (
                CPU.pin12 // pin
                , 110      // forward
                , 175      // center
                , 240      // backward
                , 2000     // low
            ) ;

        for ( int i = 1 ; i <= 10 ; ++i ) {
            CPU.delay(20000); // initial delay for measurements

            leftWheel.move ( 100 ) ;
            rightWheel.move ( 100 ) ;

            for ( int j = 0 ; j < i ; ++ j ) {
                CPU.delay(1628); // run servos for fixed amount of time
            }

            leftWheel.stop () ; // stop wheels
            rightWheel.stop () ;
        }
    }
}

```

The difference between BasicWheelServoDistanceTest1 and BasicWheelServoDistanceTest2 is the CPU.delay value. It is now the 1628 value computed as the average time to move one inch. The distance moved with this program will be significantly less than BasicWheelServoDistanceTest1.

Repeat the measurements with BasicWheelServoDistanceTest2. Note that the results will be close to an integral inch but not exactly. If the results are always farther than expected then reduce the CPU.delay value. Increase the value if the distances are less than expected.

Your Turn

- ❑ The BasicWheelServoDistanceTest1.java program is setup to move the J-Bot forward for a specific distance. Adjust the program to do the same thing when the J-Bot moves backwards.
- ❑ Change the sample program so the J-Bot moves forward for shorter and longer times. How do the plot results compare with original results?

- ❑ The sample programs run at 100%, the top speed of the J-Bot. Repeat the tests at different speeds such as 25%, 50% and 75%.

Activity #2: Maneuvers - Making Turns

Keeping the J-Bot on the straight and narrow is fine but this leads to one dimensional thinking and movement. While the J-Bot cannot climb steps and move in a third dimension it can move in two dimensions.

The J-Bot changes direction by turning or pivoting. Turning is accomplished by running the wheels in the same direction but the servos are run at different speeds. The turn will be towards the slower moving wheel and the radius of the turn will be based on the difference between the two.

Pivoting occurs when the wheels are run in opposite directions. The J-Bot will pivot in place if the wheels move at the same speed. The movement will be a mix of a turn and pivot if the speeds of the wheels differ. For our purposes, we want to pivot in place.

Pivoting In place

The BasicWheelServoPivotTest1 program shown next is designed to make the J-Bot rotate on its axis.

```
import stamp.core.*;
import JBot.* ;

/**
 * Wheel servo pivot test program
 * <p>
 * The program pivots the J-Bot.
 *
 * @version 1.0 7/23/02
 * @author Parallax Inc.
 */

public class BasicWheelServoPivotTest1 {
    public static void main () {
        BasicWheelServo leftWheel =
            new BasicWheelServo (
                CPU.pin13    // pin
                , 240        // forward
                , 175        // center
                , 110        // backward
                , 2000       // low
            ) ;
        BasicWheelServo rightWheel =
            new BasicWheelServo (
                CPU.pin12    // pin
                , 110        // forward
                , 175        // center
                , 240        // backward
                , 2000       // low
            ) ;

        for ( int i = 1 ; i <= 4 ; ++i ) {
            CPU.delay(20000);    // initial delay for measurements
        }
    }
}
```

```
leftWheel.move ( 100 ) ;
rightWheel.move ( -100 ) ;

for ( int j = 0 ; j < i ; ++ j ) {
    CPU.delay(1000);    // run servos for fixed amount of time
}

leftWheel.stop () ;    // stop wheels
rightWheel.stop () ;
}
}
```

The program is very close to the `BasicWheelServoDistanceTest` programs. The two major changes are the number of iterations, the direction of the wheels and the delay for the pivot action. The number of iterations is reduced to prevent the J-Bot from pivoting too much. This allows the test to be run with the power and serial cable attached although you need to make sure there is enough slack in the cable so it does not impede the movement of the J-Bot. Running on batteries with the serial cable disconnected eliminates this problem.

The direction of the left wheel remains the same but the right one goes in the opposite direction. This causes the J-Bot to pivot to the right. The delay time was reduced because it does not take much time for the J-Bot to pivot since it does not travel a great distance. Longer operation is possible but this just causes the J-Bot to go in circles.

The amount of time the J-Bot pivots controls the angle of rotation. We can determine the angle of rotation using the same method employed for the fixed distance programs. Again, the J-Bot's movements will be limited. Whereas the distance program used inches, the pivot programs will use increments of some fixed number of degrees.

The `BasicWheelServoPivotTest1` goes through four steps with increasing angles of rotation. As it turns out, the angles are close to values we intend to use in the long run including 45° and 90° . With a delay of 1000, four pivot steps (4000) will result in a rotation of about 45° . If we double this and use it in the `BasicWheelServoPivotTest2` then the J-Bot will rotation a full 90° on its last rotation and 45° after only two.

The `BasicWheelServoPivotTest2` uses this information. The only difference is that the rotation delay time has been increased to 4000. This is an approximation but it should be close. The J-Bot should pivot 45° , 90° , 135° and 180° . Be careful with this program if the cables are attached because the J-Bot will make more than one full rotation.

```

import stamp.core.*;
import JBot.* ;

/**
 * Wheel servo pivot test program
 * <p>
 * The program rotates the J-Bot in 45 degree increments.
 *
 * @version 1.0 7/23/02
 * @author Parallax Inc.
 */

public class BasicWheelServoPivotTest2 {
    public static void main () {
        BasicWheelServo leftWheel =
            new BasicWheelServo (
                CPU.pin13    // pin
                , 240        // forward
                , 175        // center
                , 110        // backward
                , 2000       // low
            ) ;
        BasicWheelServo rightWheel =
            new BasicWheelServo (
                CPU.pin12    // pin
                , 110        // forward
                , 175        // center
                , 240        // backward
                , 2000       // low
            ) ;

        for ( int i = 1 ; i <= 4 ; ++i ) {
            CPU.delay(20000);    // initial delay for measurements

            leftWheel.move ( 100 ) ;
            rightWheel.move ( -100 ) ;

            for ( int j = 0 ; j < i ; ++ j ) {
                CPU.delay(4000);    // run servos for fixed amount of time
            }

            leftWheel.stop () ;    // stop wheels
            rightWheel.stop () ;
        }
    }
}

```

We could create a class that would incorporate the distance movement and the pivot operations so we could simply use methods like forward, pivotRight and pivotLeft but we will wait until we have tested all the useful actions including the next programs for making turns.

Your Turn

- ❑ Try changing the speed of the wheels. Does the J-Bot pivot in place or turn?

- ❑ Make the J-Bot pivot to the left instead of the right.

Taking a Turn

3

Pivoting in place is handy but not always the most preferable method of changing direction. In particular, pivoting requires the wheels to turn in different directions. The J-Bot will skid if it does not stop for a short period of time. A turn, on the other hand, keeps the wheels moving in the same direction although there is a change in speed.

A left turn is accomplished by running both wheels in a forward direction but the left wheel runs slower than the right. The following BasicWheelServoTurnTest1 should look rather familiar by now.

```
import stamp.core.*;
import JBot.* ;

/**
 * Wheel servo turn test program
 * <p>
 * The program turn the J-Bot for a fixed amount of time.
 *
 * @version 1.0 7/23/02
 * @author Parallax Inc.
 */

public class BasicWheelServoTurnTest1 {
    public static void main () {
        BasicWheelServo leftWheel =
            new BasicWheelServo (
                CPU.pin13    // pin
                , 240        // forward
                , 175        // center
                , 110        // backward
                , 2000       // low
            ) ;
        BasicWheelServo rightWheel =
            new BasicWheelServo (
                CPU.pin12    // pin
                , 110        // forward
                , 175        // center
                , 240        // backward
                , 2000       // low
            ) ;

        for ( int i = 1 ; i <= 4 ; ++i ) {
            CPU.delay(20000);    // initial delay for measurements

            leftWheel.move ( 10 ) ;
            rightWheel.move ( 100 ) ;

            for ( int j = 0 ; j < i ; ++ j ) {
                CPU.delay(10000);    // run servos for fixed amount of time
            }

            leftWheel.stop () ;    // stop wheels
        }
    }
}
```

```
        rightWheel.stop () ;  
    }  
}  
}
```

The run time, the second CPU.delay call, has been increased because it takes the J-Bot a longer to circumscribe an arc. The leftWheel.forward method call cuts the speed down to 10%. This results in a relatively tight turn with an inside radius under one inch.

As with the first pivot program, the BasicWheelServoTurnTest1 program makes the J-Bot move but the amount of movement may not be what is desired. Although not random, we will want the J-Bot to move to known positions if we plan on having it do things like move around a rectangle.

Your Turn

- ❑ Determine time settings so turns the J-Bot's rotational movements are in increments of 45°. Use the same measurement and estimation techniques used for the straight and pivot movements.
- ❑ Make the J-Bot turn in the opposite direction. Hint: swap the speed values in the forward methods.
- ❑ Have the J-Bot make wider turns.
- ❑ Can the J-Bot make a turn going backwards? Try it.

Activity #3: Maneuvers - Basic J-Bot Class

Control of the J-Bot to this point has been done explicitly. Control of an individual wheel servo has been encapsulated in the BasicWheelServo class. While it is possible to extend this class to make actions like turning and pivot easier to work with it is better if another class is defined to provide J-Bot movement.

Before diving into a new class definition we need to determine what combination of methods will be useful. For now we will not be to ambitious. We need to move forward and backward a fixed number of inches, pivot left and right, turn left and right, and stop. The following class does just that.

```
package JBot;  
  
import stamp.core.*;  
import stamp.util.os.*;
```

```

/**
 * J-Bot wheel control interface
 * <p>
 * Defines methods that J-Bot wheel control classes must provide.
 *
 * @version 2.1 03/12/03 Use new Task status support
 * @version 2.0 12/12/02 Use Event synchronization
 * @version 1.0 10/2/02
 * @author Parallax Inc.
 */

public abstract class JBotInterface {
    protected Event startEvent ;
    protected Event nextEvent = Event.nullEvent ;
    protected Event oneTimeEvent = Event.nullEvent ;

    static final public int continuousForward = 32760 ;
    static final public int continuousBackward = -32760 ;
    static final public int continuousLeft = continuousForward ;
    static final public int continuousRight = continuousBackward ;

    /**
     * Set Event that is to be notified when a movement starts.
     *
     * @param event Event to be notified when movement starts. Can be null.
     */
    public JBotInterface ( Event event ) {
        startEvent = Event.checkEvent(event);
    }

    /**
     * Setup (current) task to wait for end of movement
     *
     * @param state next task state
     */
    public void wait ( int state ) {
        Task waitingTask = Task.getCurrentTask () ;
        waitingTask.nextState ( state ) ;

        if ( waitingTask.taskStatus()==Task.taskRunning) {
            waitingTask.suspend () ;
            oneTimeEvent = waitingTask ;
        }
    }

    /**
     * Set event to notify when movement is started. May be null.
     *
     * @param event Event to notify when movement is started
     *
     * @returns prior event
     */
    public Event setStartEvent ( Event event ) {
        Event resultEvent = startEvent ;
        startEvent = Event.checkEvent(event) ;
        return resultEvent ;
    }

    /**
     * Set event to notify when movement is done. May be null.
     *
     * @param event Event to notify when movement is done
     */

```

```

    * @returns prior event
    */
    public Event setNextEvent ( Event event ) {
        Event resultEvent = nextEvent ;
        nextEvent = Event.checkEvent(event) ;

        // Prime the pump if necessary
        if ( movementDone () ) {
            causeNextEvent () ;
        }

        return resultEvent ;
    }

    /**
     * Cause one time and next event.
     * Normally called by the matching multitasking object.
     */
    protected void causeNextEvent () {
        // Notify event that the next movement can be started
        nextEvent.notify ( this ) ;

        // Notify event that the next movement can be started
        oneTimeEvent.notify ( this ) ;
        oneTimeEvent = Event.nullEvent ;
    }

    /**
     * This method should be called at the beginning of a movement method.
     */
    protected void startMovement () {
        startEvent.notify (this) ;
    }

    /**
     * Check if movement is done. This should be called until it
     * returns true.
     *
     * @returns true if done waiting for movement
     */
    public abstract boolean movementDone () ;

    /**
     * Stop movement.
     */
    public void stop () {
        move ( 0 ) ;
    }

    /**
     * Set wheel speed to move forward/backward (negative).
     *
     * @param inches number of inches to move
     */
    public abstract void move ( int inches ) ;

    /**
     * Set wheel speed to pivot left (positive) or right (negative).
     *
     * @param steps number of steps to turn
     */
    public abstract void pivot ( int steps ) ;

```

```
/**
 * Turn left (positive) or right (negative).
 *
 * @param steps number of steps to turn
 */
public abstract void turn ( int steps ) ;
}
```

The first thing to notice with the JBotInterface class is that it is an abstract class. This means it cannot be used to create an object but it can be used as a super class to a class that is not an abstract class. The second thing to note is the use of the Event class. This is part of the stamp.util.os package which is why the statement:

```
import stamp.util.os.*;
```

is included at the start of the file. The Event class is relatively simple. It has a method called notify that is used to communicate with an Event object. In this case there are three events: startEvent, nextEvent and the oneTimeEvent. The first is notified when a movement is started. The other two are notified when a movement is completed. This allows an event driven system to be constructed in a multitasking environment covered in the next chapter. These events are discussed in more detail later in this section.

The Event.nullEvent is the only instance of the Event class that does nothing when it is notified. While this may sound useless it allows a reference like nextEvent to be used without having to check for a null value. Since another event is normally referenced by this variable it is more efficient to use a nullEvent instead of null. Only a single nullEvent object is needed since the response to the notify method is always the same. It does nothing.

Before covering the rest of the JBotInterface class definition we take a look at the class hierarchy for it and the events that will be used here and in the rest of the experiments in the book. First the JBotInterface hierarchy.

```
JBotInterface
    BasicJBot
        RampingJBot
            WheelEncoderJBot
```

The BasicJBot class, defined next, implements the abstract JBotInterface methods so a real object can be created. Normally a single object is created since there is only one J-Bot but the class allows an easy way to collect together the methods associated with its operation.

The BasicJBot provides support for forward and backward movement, pivoting and turning. It starts and stops the servos abruptly so it can be unsuitable for battery operation in some instances especially as additional sensors or add-on modules are employed.

The RampingJBot builds on the BasicJBot. It removes the abrupt change of speed with a gradual increase or decrease in speed. This minimizes the power surge necessary to start or stop a servo making it more suitable for battery operation.

The WheelEncoderJBot adds a closed feedback loop to the RampingJBot. It uses the wheel encoder IR detectors to track the color transitions on the inside of each wheel. It allows the J-Bot to move forward in a straight line by making sure the number of transitions for each wheel are the same over time.

The BasicJBot and RampingJBot are covered in this chapter. The WheelEncodingJBot is covered in a later chapter.

Much of the work for controlling the J-Bot is done in the movementDone method. This must be called repeatedly so the object can keep track of the servos. The servos actually run continuously but they must be turned off or the speed must change if the J-Bot is not running continuously in one direction.

The movementDone method can be polled in a number of ways. The simplest method is to have the part of the program that creates the JBot object but this can be rather tedious. Instead, two alternatives are available. One assumes a single tasking, fixed movement mode of operation where each movement will complete before control is returned to the calling program. The second uses a task in a multitasking environment allowing the calling program to do other things while the J-Bot is moving. This can include support for sensors that require multitasking support. Multitasking is covered in the next chapter while sensor support is covered in subsequent chapters for various sensor types.

A control object is required as a parameter to the constructors in the JBotInterface class hierarchy. This control object is based on the Event class and is assigned to the startEvent when an object is created. It is possible to change the object but in general it is set once when the JBot object is created. The control object hierarchy looks like this.

```
Event
    FixedMovementJBot
    Task
        MultitaskingJBot
```

In general, the creation of a BasicJBot with a fixed movement control object looks like:

```
JbotInterface jbot = new BasicJBot ( new FixedMovementJBot ());
```

Note that the type of `jbot` is `JBotInterface`. This can be done because `JBotInterface` is a super class of `BasicJBot`. It also means that `BasicJBot` can be replaced by any class, like `RampingJBot`, that has `JBotInterface` as a super class. If the `movementDone` method is called explicitly by the program that creates the object referenced by `jbot` then the parameter to the constructor can be `Event.nullEvent`.

The `FixedMovementJBot` is used throughout this chapter. The `MultitaskingJBot` class is defined in the next chapter.

The movement methods for the abstract `JbotInterface` class include `move`, `pivot` and `turn`. Each takes a single signed integer parameter. A positive value yields forward movement or pivoting and turning to the left. A negative value causes backward movement and pivoting and turning to the right. A value of 0 causes the J-Bot to stop. This is the same as the `stop` method.

It is also possible to initiate continuous movements. The J-Bot will proceed in the specified movement until a new movement is invoked. This mode is common when sensors are used to determine when a change in direction is required.

The movement methods like `move`, `pivot` and `turn` must be defined by a subclass and they need to take into account the three events: `startEvent`, `nextEvent` and `oneTimeEvent`. The `startMovement` method notifies the `startEvent`. It should be called after all the movement setup is complete. The event normally calls `movementDone` repeatedly. The `startEvent` is set when in the constructor but it can be changed using `setStartEvent`. This method returns the prior event reference. A null argument causes the event to be set to `Event.nullEvent`.

Once a movement is completed, the `causeNextEvent` method should be called. This method notifies the `nextEvent` and `oneTimeEvent`. The former is set using `setNextEvent`. This event is designed to be used all the time. The `oneTimeEvent` is set using the `wait` method and reset after the event is notified. The `wait` method assumes the system is utilizing a multitasking environment. The `wait` method should be called from a task. Waiting on an event is discussed in more detail in the next chapter on multitasking.

FYI Keep in mind that the <code>JBotInterface</code> and its subclasses are designed so that the program controlling the J-Bot will issue another movement command, including a <code>move(0)</code> or <code>stop</code> , immediately after the current command has finished. If not, the J-Bot will continue moving in the same direction and

speed. It is possible to create a control event that stops after each movement but this can result in jerky movements.

The BasicJBot class defined next extends the JBotInterface. The BasicJBot class implements all the abstract classes of its superclass so BasicJBot objects can be created. There are two constructors available. One uses the default servo objects while the other allows user defined objects to be used instead. Both require an event that will be used as the startEvent.

```
package JBot;

import stamp.core.*;
import stamp.util.os.*;

/**
 * Basic J-Bot wheel control class
 * <p>
 * Handles PWM support for a free running wheel servo on the J-Bot.
 * Start movement using move(), pivot() or turn().
 * It can stop the J-Bot using stop() or move(0).
 *
 * @version 2.0 03/20/03 Added getMovementSpeed
 * @version 1.0 12/23/02 Original version
 * @author Parallax Inc.
 */

public class BasicJBot extends JBotInterface {
    public BasicWheelServo leftWheel ;
    public BasicWheelServo rightWheel ;
    public Timer timer ;
    public int timeout ;
    public int msecPerInch ;
    public int msecPerPivot ;
    public int msecPerTurn ;

    protected static final int movementNone = 0 ;
    protected static final int movementMove = 1 ;
    protected static final int movementPivot = 2 ;
    protected static final int movementTurn = 3 ;

    protected int leftMovementSpeed ;
    protected int rightMovementSpeed ;
    protected int msecPerStep ;

    /**
     * Setup wheel servos for general movement.
     * Forward movement is measured in inches.
     * Pivots and turns are measured in steps that are normally 45 degrees.
     * Uses default servo settings.
     */
    public BasicJBot ( Event event ) {
        this ( event
            , 163 // ( 1628 * 100us ) / 1000us = 163 msec
            , 180 // ( 1800 * 100us ) / 1000us = 180 msec
            , 650 // ( 6500 * 100us ) / 1000us = 650 msec
            , new BasicWheelServo (
                CPU.pin13 // pin
                , 240 // forward
            )
        )
    }
}
```

```

        , 175          // center
        , 110          // backward
        , 2000         // low
    )
    , new BasicWheelServo (
        CPU.pin12      // pin
        , 110          // forward
        , 175          // center
        , 240          // backward
        , 2000         // low
    )
    ) ;
}

/**
 * Setup wheel servos for general movement.
 * Forward movement is measured in inches.
 * Pivots and turns are measured in steps that are normally 45 degrees.
 *
 * @param msecPerInch number of msec per inch for linear movement
 * @param msecPerPivot number of msec per pivot unit
 * @param msecPerTurn number of msec per turn unit
 * @param leftWheel BasicWheelServo for left wheel
 * @param rightWheel BasicWheelServo for right wheel
 */
public BasicJBot
( Event event
  , int msecPerInch
  , int msecPerPivot
  , int msecPerTurn
  , BasicWheelServo leftWheel
  , BasicWheelServo rightWheel
) {
    super(event);
    this.msecPerInch = msecPerInch ;
    this.msecPerPivot = msecPerPivot ;
    this.msecPerTurn = msecPerTurn ;
    this.leftWheel = leftWheel ;
    this.rightWheel = rightWheel ;
    timer = new Timer () ;
}

/**
 * Check if movement is done. This should be called until it
 * returns true.
 *
 * @returns true if done waiting for movement
 */
public boolean movementDone () {
    return timer.timeout ( timeout ) ;
}

/**
 * Set wheel speed to move forward.
 *
 * @param steps number of linear inches to move
 */
public void move ( int steps ) {
    movement ( movementMove, steps ) ;
}

/**
 * Set wheel speed to pivot left (positive) or right (negative).

```

```

*
* @param steps number of steps to turn
*/
public void pivot ( int steps ) {
    movement ( movementPivot, steps ) ;
}

/**
* Turn left (positive) or right (negative).
*
* @param steps number of steps to turn
*/
public void turn ( int steps ) {
    movement ( movementTurn, steps ) ;
}

/**
* Get wheel speeds for movement.
* Sets msecPerStep, leftMovementSpeed, rightMovementSpeed
*
* @param movement movement type
* @param steps number of steps to turn
*/
protected void getMovementSpeed ( int movement, int steps ) {
    switch ( movement ) {
    default:
    case movementMove:
        msecPerStep = msecPerInch ;

        if ( steps > 0 ) {
            leftMovementSpeed = 100 ;
            rightMovementSpeed = 100 ;
        } else {
            leftMovementSpeed = -100 ;
            rightMovementSpeed = -100 ;
        }
        break;

    case movementPivot:
        msecPerStep = msecPerPivot ;

        if ( steps > 0 ) {
            leftMovementSpeed = -100 ;
            rightMovementSpeed = 100 ;
        } else {
            leftMovementSpeed = 100 ;
            rightMovementSpeed = -100 ;
        }
        break;

    case movementTurn:
        msecPerStep = msecPerTurn ;

        if ( steps > 0 ) {
            leftMovementSpeed = 10 ;
            rightMovementSpeed = 100 ;
        } else {
            leftMovementSpeed = 100 ;
            rightMovementSpeed = 10 ;
        }
        break;
    }
}

```

```

    // Compute timeout when appropriate
    if ( ( steps != continuousForward )
        || ( steps != continuousBackward )
        || ( steps != 0 ) ) {
        timeout = msecPerStep * (( steps > 0 ) ? steps : - steps ) ;
    }
}

// ==== Private definitions follow ====

/**
 * Set wheel speed for movement.
 * Use movement 1 for positive values.
 * Use movement 2 for negative values.
 *
 * @param movement movement type
 * @param steps number of steps to move
 */
protected void movement ( int movement, int steps ) {
    // setup parameters
    getMovementSpeed ( movement, steps ) ;

    switch ( steps ) {
    case 0:
        setSpeed(0,0);
        startMovement();
        break;

    case continuousForward:
    case continuousBackward:
        setSpeed(leftMovementSpeed,rightMovementSpeed);
        break;

    default:
        timer.mark () ;
        setSpeed(leftMovementSpeed,rightMovementSpeed);
        startMovement();
        break;
    }
}

/**
 * Set speed for both wheels.
 * Settings are percentages.
 * Positive values are forward rotation.
 * Negative values are backward rotation.
 *
 * @param left speed settings for left wheel
 * @param right number speed setting for right wheel
 */
protected void setSpeed ( int left, int right ) {
    // Set real speed
    leftWheel.move(left);
    rightWheel.move(right);
}
}

```

If you made it this far then you have seen that the class definition is a bit long. Still, it is relatively simple. Up front are the variable definitions. Although they are defined as

public variables it might be better to make them protected. This prevents applications like the forthcoming BasicJBotTest1.java program.

The variables include two wheel object references and a Timer object. In prior programs the timing was provided by CPU.delay. In this class the timing will be done using a Timer object. The timeout variable is used to keep track of movement duration and is used in conjunction with the Timer.timeout method.

The three variables, msecPerInch, msecPerPivot and msecPerTurn are values that are based on the experiments done earlier in this chapter. For example, the BasicWheelServoDistanceTest2.java program was used to verify the time delay for moving in inches. This value was 1628 (your value may be slightly different) but it is in terms of 100usec units used by the CPU.delay function. The Timer has a 1msec (1msec is 1000usec) timeout method so we need to change the value by a factor of 10. This is 162.8 or 163 since the Javelin only deals with integers.

The pivot and turn values are designed for 45° steps. The J-Bot can be programmed for finer gradations but its accuracy and repeatability are limited due to the lack of feedback. Therefore, the 45° step increment should be more than adequate.

There are two class constructor definitions. The first uses the second and allocates the BasicWheelServo objects using predefined constants. This should be sufficient for a single J-Bot. The second definition is provided if the class is used on multiple J-Bots where the constants may have slightly different values. In this case an object can be created without modifying the BasicJBot class.

Next come the public method definitions. These are the ones a programmer will use after a BasicJBot object is created. We will go through these definitions next. These methods are followed by protected method definitions that are used within this class. They can be used by subclass definitions as well. We subclass the BasicJBot class for fixed movement and ramping covered later in the chapter. Back to the public methods.

The movementDone method checks the Timer object to determine if it is time to stop moving. It returns a value of true if this is the case. The startEvent stored in the superclass object variables can use this method in a while loop and stops both wheels when the movement is done. The stop method is defined later in the file but simply calls the stop method for both wheel objects.

Now why use these methods instead of CPU.delay? There are two reasons. First, it allows the movement timing to be centralized. Second, this type of polled architecture meshes well with the

multitasking system presented in the next chapter. The multitasking system polls tasks. The task controlling the wheels can use the `movementDone` method at this time. This will be covered in more detail after the multitasking system is covered. For now, just see how it works.

In terms of performance, using the `Timer` and `CPU.delay` are equivalent since the program is not doing anything else. In fact, the wheel movement is actually controlled by the background operation of the `PWM` object for each wheel.

We can quickly look at the `stop` method. As mentioned earlier, it just stops the two wheels. It uses a call to `move(0)` allowing movements to be coordinated by a single method. This turns out to be useful in the `RampingJBot` class that uses the `BasicJBot` as its superclass.

Next we take a close look at the `move`, `pivot` and `turn` methods. These call the `movement` method using different parameters. The first is the same as the argument to the method. The second is the timeout increment for each step. For the `move` method the `msecPerInch` this is the number of ticks that a servo needs to run to move one inch. The same is true for the other two methods.

The `movement` method sets up the timeout to be used with the timer object access in the `movementDone` method. The `movement` method calls the `getMovementSpeed` that determines the type of operation being performed. A 0 step value indicates the servos are to be stopped. This is done by setting the speed of each wheel to 0.

The `getMovementSpeed` method is used to so it can be enhanced by subclasses such as the `RampingJBot` class. The `getMovementSpeed` method sets the servo speed parameters and the timeout (duration) of the movement. The method takes into account the sign of the step variable.

Back in the `movement` method, the `startMovement` method is called to start of a fixed movement action. This includes stopping indicated by a step of 0 and non-continuous movements. If the step indicates a continuous movement then the `startMovement` method is not called. It is assumed that the calling program will initiate another movement when it wants to. The continuous movement step values are actually the upper and lower limits of an integer that would not normally be used to indicate the number of steps to move because these values are very large (i.e. in excess of 32,000).

The `setSpeed` method sets the speed of the servos.

FYI Remember that the <code>BasicWheelServo</code> move methods start the wheels moving if they were not already moving.

The BasicJBot class is a little more complex than it needs to be. As was mentioned earlier, this was done to accommodate subclasses. We had the advantage of 20-20 hindsight when designing the BasicJBot class because we knew about ramping and wheel encoder support. This makes these classes easier to write and to understand. It is possible to create these classes without this support but in general it results in a duplication of the BasicJBot class. Either approach is valid but this approach allows a more incremental presentation of the design while minimizing the duplication of support code within the subclasses.

So much for the class definition. The following program makes use of the BasicJBot class. This program is significantly shorter although it only takes advantage of some of the methods defined in the BasicJBot class.

```
import stamp.core.*;
import JBot.* ;

/**
 * Test BasicJBot class
 * <p>
 * The program runs the J-Bot using BasicJBot class methods.
 *
 * @version 1.0 7/23/02
 * @author Parallax Inc.
 */

public class BasicJBotTest1 {
    public static void main () {
        BasicJBot jbot = new BasicJBot (null) ;

        jbot.move ( 2 ) ;
        while ( ! jbot.movementDone () ) ;

        jbot.pivot ( -2 ) ;
        while ( ! jbot.movementDone () ) ;

        jbot.turn ( 1 ) ;
        while ( ! jbot.movementDone () ) ;

        jbot.stop () ;
        while ( ! jbot.movementDone () ) ;
    }
}
```

Three movements are performed by the main method. First the Jbot moves forward two inches. It then pivots 90° to the right followed by a 45° turn to the left. Notice how the movementDone is called after each movement call. This is necessary because there is no startEvent. The null argument to the BasicJBot constructor causes the startEvent to be set to Event.nullEvent.

It is possible to test the J-Bot when it is tethered by the power and serial cables for this simple program but it will be very difficult when more and longer movements are used. Instead, test by programming the J-Bot using debug mode while tethered. Then remove power and the serial cables. Provide power using the battery pack and the program will then run.

FYI

The J-Bot may not run properly using batteries if they are rechargeable or partially discharged. In this case the J-Bot may appear to start or run part of the program and then reset. This is not unusual. Simply run the J-Bot in tethered mode without the batteries. The RampingJBot described later in this chapter may eliminate this problem.

One way to test for this situation is to use the tone generator presented in Chapter 1 at the beginning of the main method. The tone should sound only once if the J-Bot is not being reset.

Your Turn

- ❑ Try out other BasicJBot movement methods to make sure they work properly. Use different distance values.
- ❑ Make the J-Bot move in a 3 x 4 inch rectangle.
- ❑ Make the J-Bot move in a circle. Hint: Try turning a lot.

Activity #4: Maneuvers - Fixed Movement Class

The BasicJBot class is handy but the test program shows that each action requires multiple method calls. These calls can be cut in half by eliminating the movementDone call. Instead of coming up with a completely new implementation we simply pass a different object to the BasicJBot class constructor. The following FixedMovementJBot class definition handles the movementDone calls instead.

```
package JBot;

import stamp.core.*;
import stamp.util.os.*;

/**
 * J-Bot wheel control class for fixed movements
 * <p>
```

```

* Combine with BasicJBot class for movement control.
*
* @version 2.0 12/12/02 Event version
* @version 1.0 7/23/02
* @author Parallax Inc.
*/

public class FixedMovementJBot extends Event {
    protected boolean idle = true ;

    /**
     * Called when movement started.
     */
    public void notify (Object jbot) {
        if ( idle && ( jbot != null )) {
            idle = false ;
            while ( ! ((JBotInterface)jbot).movementDone () ) {
            }
            idle = true ;
        }
    }
}

```

The FixedMovementJBot class is based on the Event class. It requires a redefinition of only one method: notify. The Object passed to the method should be the JBotInterface that controls the J-Bot's movements. It should not be null but a check is performed just in case. The method does nothing if the reference is null. Otherwise, it waits until movementDone returns true.

The result is that a call to the JBotInterface's move, pivot or turn methods will not return until the movement is complete. The following program exercises the new class definition.

```

import stamp.core.*;
import JBot.* ;

/**
 * Test FixedMovementJBot class
 * <p>
 * The program runs the J-Bot using BasicJBot class methods.
 *
 * @version 1.0 7/23/02
 * @author Parallax Inc.
 */

public class BasicJBotTest2 {
    public static void main () {
        BasicJBot jbot = new BasicJBot ( new FixedMovementJBot () ) ;

        jbot.move ( 2 ) ;    // forward
        jbot.pivot ( -2 ) ; // pivot right
        jbot.turn ( 2 ) ;    // turn left
        jbot.stop () ;
    }
}

```

The BasicJBotTest2 program is significantly shorter (ignoring the comments) than the prior test program. It is also easier to read. The calls the jbot movement methods like jbot.move will not return until the movement is completed due to the use of the FixedMovementJBot object.

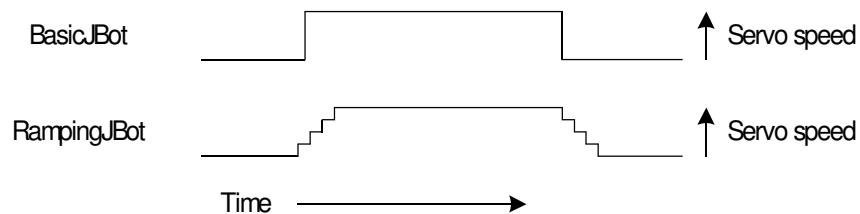
Activity #5: Maneuvers - Ramping Servo Class

Ramping is a way to gradually increase the speed of the servos instead of suddenly making them go the opposite direction. This technique can increase the life expectancies both of your J-Bot's batteries and servos. It also provides better control over movement. Without ramping the J-Bot goes from 0 to 100% power or at least it tries to. Do this with a car and it will leave skid marks. The J-Bot will not leave skip marks but it will have a jerky movement that is undesirable.

Programming for Ramping

The RampingJBot class adds some protected methods for internal use but requires no additional change to the movement method operation. Instead, it determines how to make incremental changes between movements based on the current speed of each wheel and the desired speed. This allows a move forward to be immediately followed by a backward movement. In this case, the J-Bot will slow down to a stop in the forward direction and then accelerate backward until it is running full speed backwards.

Ramping takes slightly longer to move the same distance because the JBot is slowing down or speeding up during the ramping period as shown below.



The full speed movement in the ramping implementation needs to be slightly shorter than the full speed movement without ramping because JBot is still moving during the ramping period. Ramping can utilize any number of steps but too many will either take a much longer period of time or be so short as to be limited by the performance of the Javelin.

The following class definition provides the ramping support.

```
package JBot;
```

```

import stamp.core.*;
import stamp.util.os.*;

/**
 * J-Bot wheel control class for fixed movements with ramping
 * <p>
 * Uses BasicJBot class for movement control.
 *
 * @version 2.0 3/20/03 Ramp up and down between movements
 * @version 1.0 7/23/02 Original version
 * @author Parallax Inc.
 */

public class RampingJBot extends BasicJBot {
    static final private int rampTimeout = 35 ; // milliseconds
    static final private int maxRampCount = 3 ;

    protected boolean decelerate = false ;
    protected int rampCount = 0 ;
    protected int rampLeftStep ;
    protected int rampRightStep ;

    protected int leftSpeed = 0 ;
    protected int rightSpeed = 0 ;

    protected int currentMovement = movementNone ;
    protected int currentSteps ;
    protected int nextMovement = movementNone ;
    protected int nextSteps ;

    /**
     * Setup wheel servos for general movement.
     * Forward movement is measured in inches.
     * Pivots and turns are measured in steps that are normally 45 degrees.
     * Uses default servo settings.
     *
     * @param event Event to be notified when movement starts. Can be null.
     */
    public RampingJBot ( Event event ) {
        super ( event ) ;
    }

    /**
     * Setup wheel servos for general movement.
     * Forward movement is measured in inches.
     * Pivots and turns are measured in steps that are normally 45 degrees.
     *
     * @param event Event to be notified when movement starts. Can be null.
     * @param msecPerInch number of msec per inch for linear movement
     * @param msecPerPivot number of msec per pivot unit
     * @param msecPerTurn number of msec per turn unit
     * @param leftWheel BasicWheelServo for left wheel
     * @param rightWheel BasicWheelServo for right wheel
     */
    public RampingJBot
    ( Event event
      , int msecPerInch
      , int msecPerPivot
      , int msecPerTurn
      , BasicWheelServo leftWheel
      , BasicWheelServo rightWheel
    ) {

```

```

    super ( event, msecPerInch, msecPerPivot, msecPerTurn, leftWheel,
rightWheel ) ;
}

/**
 * Check if next movement can be initiated.
 *
 * @returns true if next movement method can be called
 */
public boolean ready() {
    return nextMovement == movementNone ;
}

/**
 * Check if movement is done. This should be called until it
 * returns true.
 *
 * @returns true if done waiting for movement
 */
public boolean movementDone () {
    if ( currentMovement == movementNone ) {
        if ( nextMovement == movementNone ) {
            // nothing left to do
            return true;
        } else {
            // start up new movement from a dead stop
            currentMovement = nextMovement ;
            currentSteps     = nextSteps ;
            nextMovement     = movementNone ;

            // Startup ramping
            startRamping () ;
        }
    } else {
        // currentMovement is active
        if ( rampCount > 0 ) {
            if ( timer.timeout ( rampTimeout ) ) {
                // Ramp timeout occurred. Adjust speed.
                -- rampCount ;

                if ( rampCount == 0 ) {
                    if ( decelerate ) {
                        // should be stopped, setup to accelerate
                        decelerate = false ;
                        startRamping () ;
                    } else {
                        // should be up to speed
                        startFullSpeedMovement(true) ;
                    }
                } else {
                    // setup next ramp movement
                    updateRampSpeed() ;
                }
            }
        } else {
            // Not ramping. Check movement status
            if ( super.movementDone () ) {
                getNextMovement () ;
            }
        }
    }
    return false ;
}

```

```

/**
 * Get next movement setup
 */
protected void getNextMovement () {
    // Get nextMovement
    if ( nextMovement == movementNone ) {
        // check if anything more to do
        if ( currentMovement == movementNone ) {
            return ;
        }

        if ( ( currentMovement == movementMove )
            && ( currentSteps == 0 ) ) {
            // Already stopped. Do nothing more
            currentMovement = movementNone ;
            return ;
        }

        // setup to stop if no movement acquired
        nextMovement = movementMove ;
        nextSteps = 0 ;

        // Get next movement
        causeNextEvent () ;
    }

    // update steps, currentMovement must be updated later
    currentSteps = nextSteps ;

    // check if next movement or if need to decelerate
    if ( nextMovement == currentMovement ) {
        startFullSpeedMovement(false) ;
    } else {
        // decelerate first
        currentMovement = nextMovement ;
        rampDown();
    }
}

/**
 * Start ramping for currentMovement
 */
protected void startRamping () {
    if ( currentSteps == 0 ) {
        super.move(0); // stop servos
    } else {
        getMovementSpeed ( currentMovement, currentSteps ) ;

        rampTo ( leftMovementSpeed, rightMovementSpeed ) ;
    }
}

/**
 * Start ramping for deceleration
 */
protected void rampDown () {
    decelerate = true ;
    rampTo ( 0, 0 ) ;
}

/**
 * Start movement, ramp up already complete

```

```

*
* @param ramping true if ramping time must be subtracted
*/
protected void startFullSpeedMovement ( boolean ramping ) {
    int timeoutAdjust = 0 ;

    switch ( currentMovement ) {
    case movementMove:
        super.move ( currentSteps ) ;
        timeoutAdjust = msecPerStep - 2 ;
        break;

    case movementPivot:
        super.pivot ( currentSteps ) ;
        timeoutAdjust = msecPerStep - 2 ;
        break;

    case movementTurn:
        super.turn ( currentSteps ) ;
        timeoutAdjust = ( currentSteps == 1 )
            ? (( 3 * msecPerStep ) / 8 )
            : ( msecPerStep / 2 ) ;

        break;
    }

    // adjust timeout for ramping
    if ( ramping ) {
        timeout -= timeoutAdjust ;
    }

    // Reset next movement
    nextMovement = movementNone ;
}

/**
 * Setup to ramp to desired speed.
 */
* @param left  final left speed
* @param right final right speed
*/
protected void rampTo ( int left, int right ) {
    rampCount      = maxRampCount - 1 ; // ramp down count less current step
    rampLeftStep   = ( left - leftSpeed ) / maxRampCount ;
    rampRightStep  = ( right - rightSpeed ) / maxRampCount ;
    updateRampSpeed() ;
}

/**
 * Update speed using ramp parameters
 */
protected void updateRampSpeed () {
    // set timeout mark for next ramp speed update
    timer.mark();

    // Set new speed
    setSpeed(leftSpeed+rampLeftStep,rightSpeed+rampRightStep) ;
}

/**
 * Set wheel speed to move forward.
 */
* @param inches number of linear inches to move
*/

```



```

public void move ( int inches ) {
    setNextMovement ( movementMove, inches ) ;
}

/**
 * Set wheel speed to pivot left.
 *
 * @param steps number of steps to turn
 */
public void pivot ( int steps ) {
    setNextMovement ( movementPivot, steps ) ;
}

/**
 * Set wheel speed to turn left.
 *
 * @param steps number of steps to turn
 */
public void turn ( int steps ) {
    setNextMovement ( movementTurn, steps ) ;
}

/**
 * Setup next movement to occur.
 * The movement will be started via a call to movementDone()
 *
 * @param movement next movement to perform
 * @param steps number of steps to turn
 */
protected void setNextMovement ( int movement, int steps ) {
    nextMovement = movement ;
    nextSteps     = steps ;

    // start moving if currently stopped
    if ( currentMovement == movementNone ) {
        startMovement();
    }
}

/**
 * Set speed for both wheels.
 * Settings are percentages.
 * Positive values are forward rotation.
 * Negative values are backward rotation.
 *
 * @param left speed settings for left wheel
 * @param right speed setting for right wheel
 */
protected void setSpeed ( int left, int right ) {
    // Set real speed
    super.setSpeed(left,right);
    leftSpeed  = left ;
    rightSpeed = right ;
}

/**
 * Adjust speed for both wheels.
 * Settings are percentages.
 * Positive values are forward rotation.
 * Negative values are backward rotation.
 *
 * @param left speed adjustment for left wheel
 * @param right speed adjustment for right wheel

```

```
*/
protected void adjustSpeed ( int left, int right ) {
    super.setSpeed
        ( leftSpeed + (( leftSpeed * -left ) / 100 )
        , rightSpeed + (( rightSpeed * -right ) / 100 ));
}
}
```

How the Ramping Class Works

The RampingJBot class makes use of the services provided by the BasicJBot but it adjusts the way the movementDone and setSpeed methods operate. It does this so the object can change the speed in small increments instead of large jumps that can occur with the BasicJBot. For example, a move(1) method call to a BasicJBot object followed by a move(-1) will result in the J-Bot moving forward for about one inch and then reversing direction and moving backward an inch. The J-Bot's wheels will jerk to a start and then in reverse as each movement starts. In the latter case, the relative speed change is 200% (100% to -100%). This can generate a significant power surge causing problems when the J-Bot is battery operated.

Instead, the RampingJBot takes a look at the J-Bot's current speed (remember that it was saved in the setSpeed method). It then compares it to the desired speed and sets things up so the speed change will occur in short steps. This does result in a minor distance change but it eliminates the jerky movements and power surges. The class also tries to take the change in distance into account when computing how far it should run.

There are some extra variables defined in the RampingJBot object. The rampCount controls the number of steps to take between speed changes. The rampLeftStep and rampRightStep are the percentage speed changes for each wheel. Two variables are needed because the two servos must be controlled independently. Finally, the final, desired speed is saved. This is necessary because stepping to that point can result in a difference since integer values are used and there can be rounding errors.

The variables currentMovement and nextMovement, along with currentSteps and nextSteps, keep track of movements. It may seem odd to keep the next movement around but it is necessary to allow a proper transition between movements and to allow two identical movements to be combined into one. For example, moving forward 5 and then 10 should be the same as moving forward 15.

The setSpeed method calls the BasicJBot's setSpeed and stores the current speed. This allows the adjustSpeed method to change the speed. This method IS NOT used by the BasicJBot object. It is

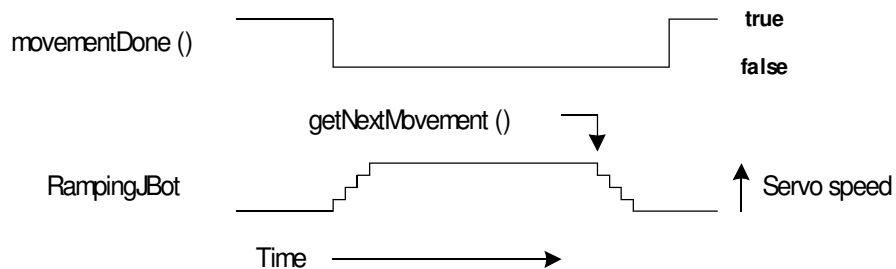
included to support the `WheelEncoderJBot` subclass that needs to adjust the speed based on the wheel encoder feedback.

The bulk of the work done is done in the `movementDone` method. This method takes into account the `currentMovement` and `nextMovement` variables. If both are set to `movementNone` then the method exits with a value of `true`. If the `currentMovement` is set to `movementNone` then the current movement has completed and it is time to transition to the `nextMovement`. This transition will occur when the prior movement is different from the `nextMovement` and ramping at the end of the prior movement has brought the JBot to a stop for a fraction of a second.

If it is not time to check the movement transitions then it may be time to check the ramping status maintained by `rampCount`. This variable is used for both the ramp up to the desired speed and movement as well as ramping down to a stop. Ramping is completed when the `rampCount` is zero. The `decelerate` variable indicates whether the ramping is up or down. If it is down then the JBot will be stopped and it is time to ramp up for the next movement. This is done by calling the `startRamping` method. If the JBot is ramping up then it is time to start the movement at full speed. This is done by calling `startFullSpeedMovement`.

If there is no transition then the ramping speed is updated by calling `updateRampSpeed`. This uses the object variables tracking the ramping support. Of course, if all else fails to be true then the movement is handled by the superclass', `BasicJBot`, `movementDone` method. If the movement is done then it is time to call `getNextMovement`.

The following figure shows when `getNextMovement` is called for a single movement.

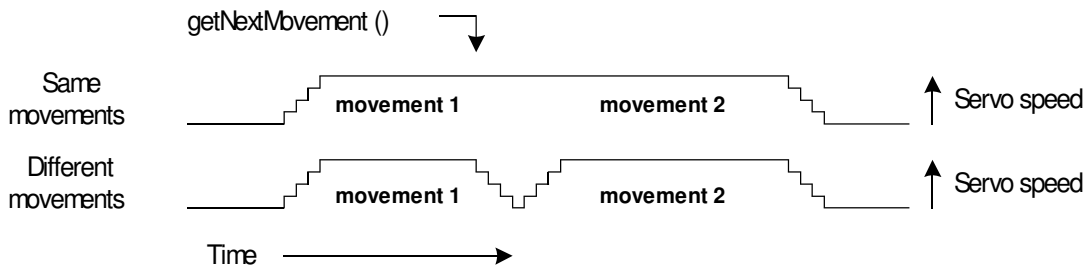


The `getNextMovement` method is a bit convoluted. It tries to see if the next movement has already been provided. If not then `causeNextEvent` is called. This support goes way back to the `JBotInterface` class. It is typically used with the `MultitaskingJBot` support. The task controlling the `RampingJBot` class is notified by the `causeNextEvent` call. The task can initiate a movement so when the call returns the `nextMovement` variable will be set. If the next movement is the same as the

prior one then `startFullSpeedMovement` is called. This time the parameter is false indicating that the ramping time should not be subtracted from the movement time because it was already subtracted from the initial movement.

The ramping methods include `startRamping`, `rampDown`, and `rampTo`. These manipulate the ramping variables to handle the up or down ramping process.

The `startFullSpeedMovement` method is next since we are looking at the code in the order it occurs in the source file. This is actually where the `BasicJBot` movement methods are called. The `BasicJBot` assumes all movements are running at full speed. The `startFullSpeedMovement` method either runs a movement between ramps or when two of the same movements occur sequentially as shown below. In the upper timing diagram, the `startFullSpeedMovement` method is called a second time at the `getNextMovement` point.



The switch statement selects the type of movement to perform next. Each case calls the `BasicJBot` super class method to setup a full speed movement. The variable `timeoutAdjust` is then set as shown below.

```
case movementMove:
    super.move ( currentSteps ) ;
    timeoutAdjust = msecPerStep - 2 ;
    break;
```

The `timeoutAdjust` value is the change necessary to take into account for ramping up and down. This value is subtracted from the object's timeout value used to control the movement if the ramping is part of this movement. Remember, if two of the same type of movement are executed back-to-back then this adjustment is only done with respect to the first movement's timeout.

The `move`, `pivot` and `turn` methods are defined in the `JBotInterface` superclass. They are redefined in this class because the method of handling movements is different. In the `RampingJBot` class, these methods simply store the requested movement in the `nextMovement` and `nextSteps` variables. As noted earlier, these variables are used when the `JBot` transitions from one movement to another.

The following program exercises the RampingJBot class.

```
import stamp.core.*;
import JBot.* ;

/**
 * Test RampingJBot class
 * <p>
 * The program runs the J-Bot using BasicJBot class methods.
 *
 * @version 1.0 7/23/02
 * @author Parallax Inc.
 */
public class BasicJBotTest3 {
    public static void main () {
        RampingJBot jbot = new RampingJBot ( new FixedMovementJBot () ) ;

        jbot.move ( -2 ) ;
        jbot.stop () ;

        jbot.move ( 2 ) ;
        jbot.pivot ( -2 ) ;
        jbot.turn ( 1 ) ;
        jbot.stop () ;
    }
}
```

The program adds a short backward movement first but then performs the same movement sequence as the prior test programs. Note how this program differs only by the change to the RampingJBot and the additional reverse movement at the beginning.

Activity #6: Driving the J-Bot

The test program in Activity #5 can be easily changed to move the J-Bot in other patterns but the amount of source code gets rather verbose. It would be easier to make patterns if the only thing to define was the movement itself. In this activity, we look at how to encode and decode movements to simplify programming.

In this example the movements are stored in a byte array. It is easy to define constant byte arrays in Java as shown in the following sample program.

```
import stamp.core.*;
import JBot.* ;

/**
 * J-Bot movements using commands stored in an array
 * <p>
 * The program runs the J-Bot using RampingJBot class methods
 * and a table of commands.
 *
 * @version 1.0 7/23/02
 * @author Parallax Inc.
 */

public class BasicJBotTest4 {
    static final byte move = -1 ;
    static final byte pivot = -2 ;
    static final byte turn = -3 ;

    RampingJBot jbot = new RampingJBot (new FixedMovementJBot ()) ;

    static byte movements [] = { move, 2, pivot, -2, move, 5 } ;

    public void performMovements ( byte movements [] ) {
        for ( int i = 0 ; i < movements.length ; i += 2 ) {
            switch ( movements [ i ] ) {
                case move:
                    jbot.move ( movements [ i + 1 ] ) ;
                    break ;

                case pivot:
                    jbot.pivot ( movements [ i + 1 ] ) ;
                    break ;

                case turn:
                    jbot.turn ( movements [ i + 1 ] ) ;
                    break ;
            }
        }

        jbot.stop () ;
    }

    // ---- Main program ----

    public static void main () {
        BasicJBotTest4 myJBot = new BasicJBotTest4 () ;
    }
}
```

```

    myJBot.performMovements ( movements ) ;
}
}

```

This program shows off a little trick. It contains both a class definition and a static main method normally associated with a main program. The methods in the class other than the static methods are available to an object of this class.

The main method is `performMovements`. It takes an array of bytes as its argument. The number of elements in the array is obtained using `movements.length`. The array is assumed to be pairs of numbers with the first being a movement that will be one of the following: `fw`, `bk`, `pl`, or `pr`. The array will not contain these letters but rather the values of the constants defined at the beginning of the class definition. The names are arbitrary.

The `performMovements` method assumes the J-Bot is at rest and will be stopped after completing the movements in the array. The ramping methods are used for starting and stopping. The start method is chosen based on the first action. It is assumed that `startForward` will be used for any action except a backward movement.

The movements array is defined as the following:

```
static byte movements [] = { move, 2, pivot, 2, move, 5 } ;
```

This should cause the J-Bot to accelerate forward then move 2 more inches forward, pivot 90° to the right, move forward 5 inches and decelerate to a full stop.

The `performMovements` method can be called using any byte array.

Your Turn

- ❑ Change the movements array so the J-Bot moves once around a 3 x 4 inch rectangle.
- ❑ Add the turn movements to the array interpretation. This means additional constants must be defined as well such as `tr` and `tl`.
- ❑ Split the `BasicJBotTest4` class definition into a separate class called `MovementJBot` and `BasicJBotTest5`. The `MovementJBot` class should be self contained allowing it to be reused in the `BasicJBotTest5` main method as well as with other classes.
- ❑ Change the movement list contents to move the J-Bot in different patterns.

- ❑ The movement list consists of a pair of numbers for each action. The movement value is always negative so it is possible to differentiate it from the setp value. Change the movement list scanning support so it assumes that the step value is 1 if it is not included.



Summary and Applications

Coordinated J-Bot wheel control was introduced in this chapter. Examples include controlling the J-Bot's distance and turns, along with methods for programming the J-Bot to travel measured distances. Examples of speed ramping as well as an example of integrating the navigation algorithms introduced in this chapter also were provided.

Real World Example

Micro-controlled motion is also all around us. Although there may not be that many autonomous rolling robots in your household yet, there are many other gizmos with micro-controlled moving parts. Printer heads and computer disk drives are two examples that use stepper motors. Servos controlled by microcontrollers are also used in a variety of places. Many automobile systems rely on servos to control small moving parts in various engine and emission systems. Industrial servos maintain many factory processes, often in conjunction with the level sensors discussed earlier.

J-Bot Applications

Programmed navigation is the foundation for a variety of other J-Bot activities. In the Projects section, you'll work on programming the J-Bot to navigate a variety of shapes and on fine tuning some of the navigational algorithms developed in this chapter. In subsequent chapters, we'll use some of these classes to respond to sensor inputs. The navigation routines developed in this chapter will be especially helpful in getting around obstacles that are detected. One of the most popular occupations for autonomous robots is solving mazes, which are full of obstacles. Responding automatically to certain situations can increase the J-Bot's ability to navigate features within the maze. For example, when a corner is detected, instead of trying to navigate the corner based solely on sensor.

Questions and Projects

Questions

1. Describe the CPU.delay was replaced with a Timer object in the BasicJBot class.
2. Why was ramping support added?

3. Describe how a turn is accomplished. How must the sample programs change to make a wider turn?
4. Why were arrays used in the BasicJBotTest4 program in Activity #6?

Exercises

1. Incorporate stop and ramping commands in the array approach used in the BasicJBotTest4 program in Activity #6.
2. Change one or more of the test programs to make the J-Bot go in a circle, a 5 inch square, or a figure 8 instead of the simple movements initially presented.

Projects

Need some project ideas.

1. Make the J-Bot move forward through a figure such as a rectangle and then backward. See how close to the starting point the J-Bot finishes.
2. Create source code for the following movement patterns:

Figure 2.8: J-Bot paths to program.