Getting Started with Java for FRC

Worcester Polytechnic Institute Robotics Resource Center

Brad Miller, Ken Streeter, Beth Finn, Jerry Morrison, Dan Jones, Ryan O'Meara, Derek White, Stephanie Hoag, Eric Arseneau

Rev 5.0 January 5, 2012

Table of Contents

Welcome to Robot Programming with Java

Starting with the 2010 competition season teams have the option to write Java programs for their robots, including a full suite of tools to ease program development and debugging.

The tools include:

- The NetBeans Integrated Development Environment (IDE) which is available for download from http://www.netbeans.org. You can install the components needed for robot development by simply adding an update site to NetBeans and installing a plugin. (Eclipse integration is coming, for another IDE choice.)
- Sun SPOT Java SDK for FRC includes the Java virtual machine and tools necessary to compile, deploy, and run Java code on the cRIO.
- The WPILib Application Programming Interface (API) for Java provides a programming interface to the cRIO. It is almost identical to the C++ interface. Converting existing C++ code to Java is simple and straightforward, and will let you reuse code developed in previous years.

The development tools run on most common platforms: **Windows**, **Mac OSX**, and **Linux**.

The complete source code for everything including the NetBeans IDE, Sun SPOT Java SDK for FRC, and the WPILib API is available to teams wishing to look at any aspect of the implementation.

Installing Java and Tools

Required Software

In order to setup your machine to program in Java, the following software components are required:

- Java SE Development Kit (JDK) version 6.
- NetBeans IDE version 6.7 or later (there may be issues running NetBeans V7, please install V6.9.1 until we've had a chance to review this).
  You can use other IDEs if desired but the focus for this document is NetBeans.
- SunSPOT Java SDK for FRC, which includes WPILib.

All these components can be installed on your platform of choice. Each platform requires slightly different installation procedures.

Install the Java tools in three steps, downloading the components from the Internet for each step:

1. Install the Java SE Development Kit (JDK) version 6 available from http://java.sun.com. Your development system may already have the JDK installed, for example on Mac OS X.
2. Install NetBeans version 6.9.1. This is available from http://netbeans.org/downloads.
3. Add the *FRC* plugins to NetBeans. These plugins can be downloaded from the WPILib project on http://firstforge.wpi.edu or installed via the NetBeans built in downloader as described in the following sections of this document.

Note: *The details of each step vary by operating system and browser.*
On 64-bit Windows the SunSPOT tools still need a 32-bit JDK so download a JDK for platform "Windows" not "Windows x64." You can install the JDK in C:\Java\32-bit\ even if you also have a 64-bit JDK in, say, C:\Java\. Give this SDK location to the NetBeans installer wizard.

Besides the tools for Java programming you'll also need:

- The FRC cRIO Imaging Tool to format/initialize your cRIO for Java programming. Be sure to use the 2012 FRC cRIO Imaging tool for updating the image on your cRIO. Previous versions will not work with the 2012 image file format and the libraries will not work with older images.
- Use the 2012 FRC Driver Station software to control your robot.

All of these tools are available from National Instruments (NI.com).

Labview update	http://joule.ni.com/nidu/cds/view/p/id/2261
Utilities update	http://joule.ni.com/nidu/cds/view/p/id/2262
Driver station update	http://joule.ni.com/nidu/cds/view/p/id/2263

Installing the NetBeans Plugins: Sun SPOT Java SDK for FRC and WPILib

The FRC Plugins add the FRC specific components to your standard NetBeans installation. The NetBeans plugins contain everything needed to extend your Java development environment to program your cRIO. The FRC plugins enable NetBeans to directly download and debug code on the NI cRIO controller. The plugins include project types and sample programs to help you get started developing robot programs.

There are two ways of installing the plugins:

1. From the FRC NetBeans update site. In this case the URL of the update site is entered into NetBeans and the plugins are installed automatically and as updates are published, you will automatically be prompted to install them provided your computer is connected to the Internet.

2. From the FIRSTForge WPILib project site. In this case you download a zip file that contains the plugins, unzip it, and enter the location of the unzipped files into the NetBeans plugin manager. In this case, you will be responsible for manually installing updates as they become available.

Installing the plugins from the update site

You get the Sun SPOT Java SDK for FRC as a NetBeans plug-in from a NetBeans update site.

To install the plugins from the update site follow these steps (see below for development computers that are not connected to the Internet):

1. Run NetBeans using the Start menu or the desktop shortcut.
2. Select "Tools" then "Plugins" from the main menu in NetBeans.
3. Select the "Settings" tab, and then press the "Add" button to add a new Update Center.
4. For the name, enter "FRC Java" and for the URL enter:
   http://first.wpi.edu/FRC/java/netbeans/update/updates.xml and press the OK button.
5. Select the "Available Plugins" tab, select all the plugins in the "FRC Java" category, and click the "Install" button. There should be 6 plugins displayed.
6. Advance by clicking the "Next" button, accept the agreements, and install the plugins. Ignore the "Validation Warning" dialog where it informs you "The following plugins are not signed:" and press the "Continue" button.
7. In the "Restart NetBeans IDE to complete the installation" window, use the "Restart IDE Now" option and click the "Finish" button.
8. After restarting NetBeans you should notice the *FIRST* logo button in the toolbar. This confirms that the module has been installed properly.

NetBeans will periodically check for new updates and offer to install them when they become available. Be sure to keep your installation current to get the latest bug fixes and improvements.

Installing Sun SPOT Java SDK for FRC without Internet Access

NetBeans is designed to automatically update its plugins on computers that are connected to the Internet. If your development system does not have Internet access and then follow this procedure:

1. Using a computer that is connected to the Internet open a browser and enter the update site URL: http://first.wpi.edu/FRC/java/netbeans/update.
2. Download the 6 NetBeans module files all with the file type of .nbm.
3. Copy the files to a USB drive. You should see 6 .nbm files.
4. Connect the USB drive that has the downloaded files to your development computer.
5. Run NetBeans using the Start menu or the desktop shortcut.
6. Select "Tools" then "Plugins" from the main menu in NetBeans.
7. Select the "Downloaded" tab and press "Add Plugins…"
8. Enter then location of the 6 .nbm files just downloaded.
9. Select "Install" to install the plugins into your NetBeans installation.

You'll need to repeat these steps when new updates are available. Be sure to watch the *FIRSTForge beta* forums for announcements of new versions of the tools. Be sure to keep your installation current to get the latest bug fixes and improvements.

Configuring Your NetBeans Installation for your team

The plugins are installed. A little configuration is required:

1. Select the "Tools" menu and choose the "Preferences" menu options from the NetBeans menu bar.
2. Select the "Miscellaneous" tab. Then select the "FRC Configuration" tab and enter your team number into the text field. Then press OK.

Installing the 2012 cRIO Imaging Tool on your development computer

Before you can download and run code on your cRIO, you must install the right operating system files onto it. To do that, you get the 2012 cRIO Imaging tool. This is available from the LabVIEW installation DVD and the LabVIEW tools updates. You can find those tools on the FIRST web site here: http://usfirst.org/roboticsprograms/frc/content.aspx?id=450.

Installing the 2012 cRIO Image for Java on your cRIO

Now that you have the cRIO Imaging tool, follow the 2012 instructions for using it to install the new image onto your cRIO.

Be sure to:

- Get the latest available software image. It is supplied as part of the Java Netbeans plugins.
- Select Java as the image type.
- Check the "Reformat" box in the imaging tool to ensure the image is installed afresh.

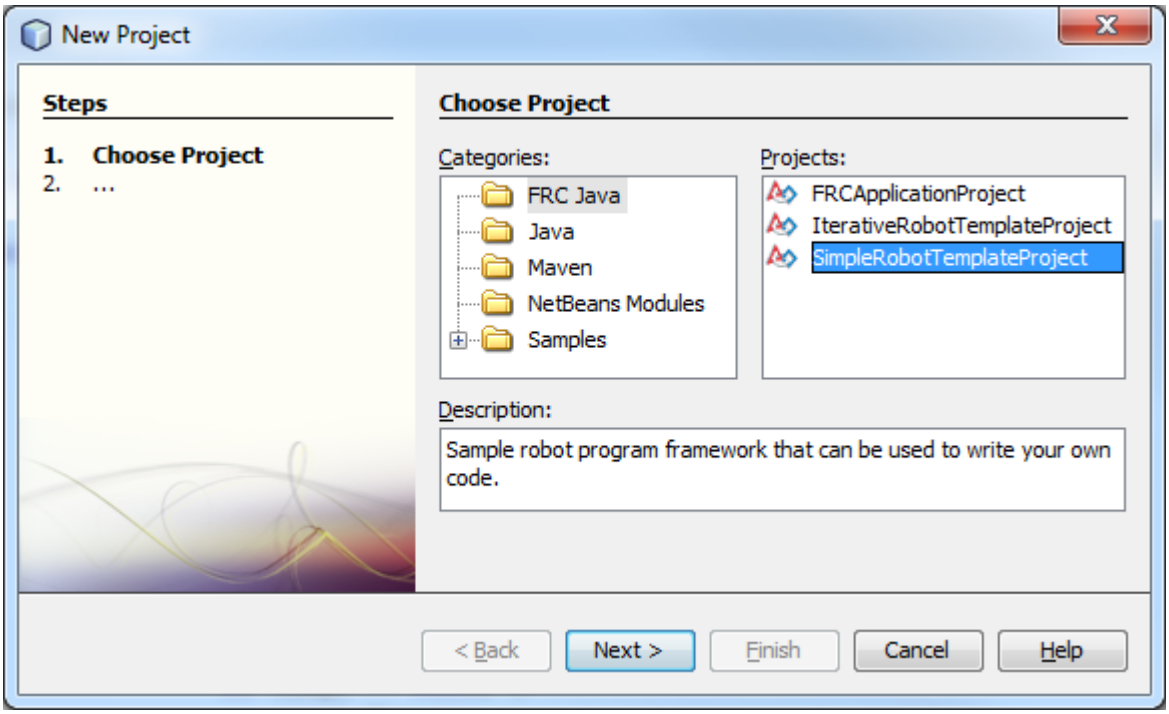
Note: There is one cRIO image per language and this is different than prior years. The image comes from the Java NetBeans plugins installation, so you won't be able to update the cRIO image until you've installed the plugins and started NetBeans at least once.

Creating a Robot Project

Note: The following steps describe creating a very simple robot program and make a good starting point for simple projects. For more complex robots with multiple subsystems, please refer to the WPILib Programming Cookbook posted in the Documents section of the WPILib project on FIRSTForge.wpi.edu.

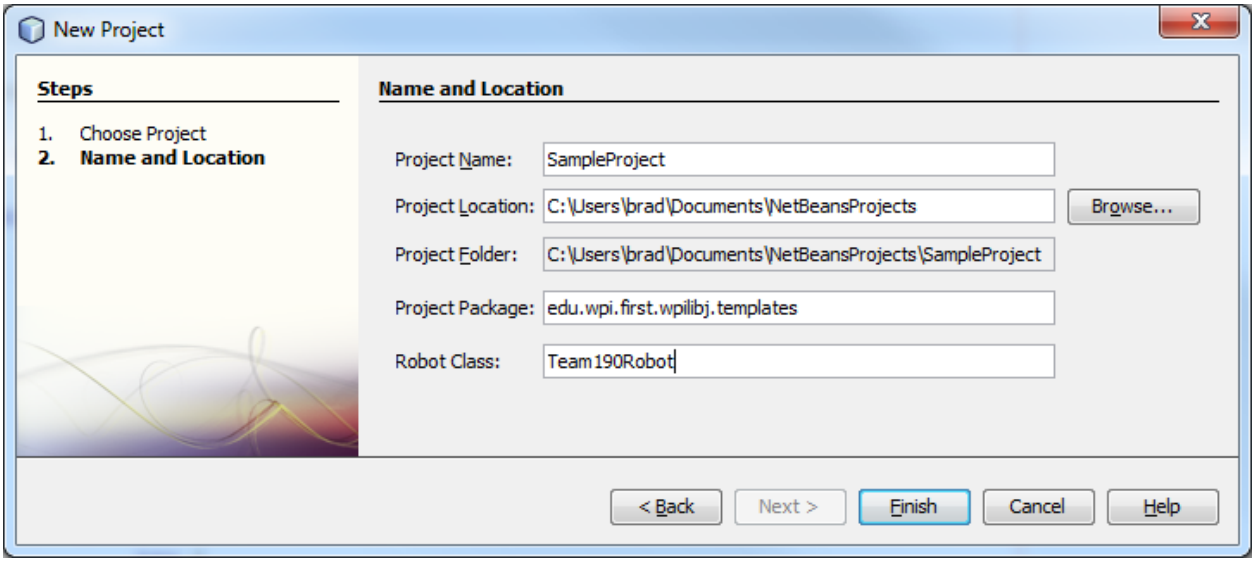
To create your first Java project, perform these steps:

1. Right click in the projects pane on the left side of NetBeans, and then select "New Project."

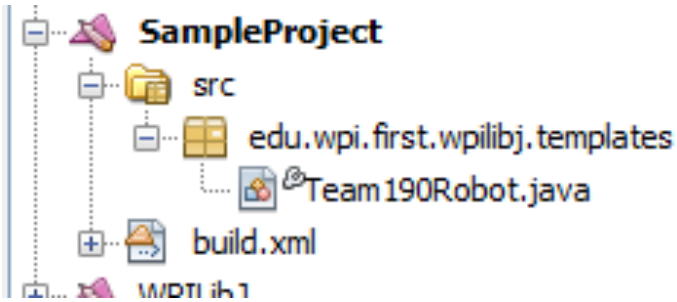


2. Select "FRC Java" and "SimpleRobot Template Project," then click "Next."

3. Type a project name and a class name. The named class will contain your robot program including its methods for the autonomous and operator control periods. In this example, we choose "SampleProject" for the project name and "Team190Robot" for the class name. Then click "Finish."



4. Close the output.xml window. Look in the project tab for the files generated by the New Project wizard:



5. The source file "Team190Robot.java" has the same name as the class, "Team190Robot". Java requires the class name to match the file name. The generated file looks like this (plus comments left out for brevity):

```
package edu.wpi.first.wpilibj.templates;

import edu.wpi.first.wpilibj.SimpleRobot;

public class Team190Robot extends SimpleRobot {

    public void autonomous() {

    }

    public void operatorControl() {

    }
}
```

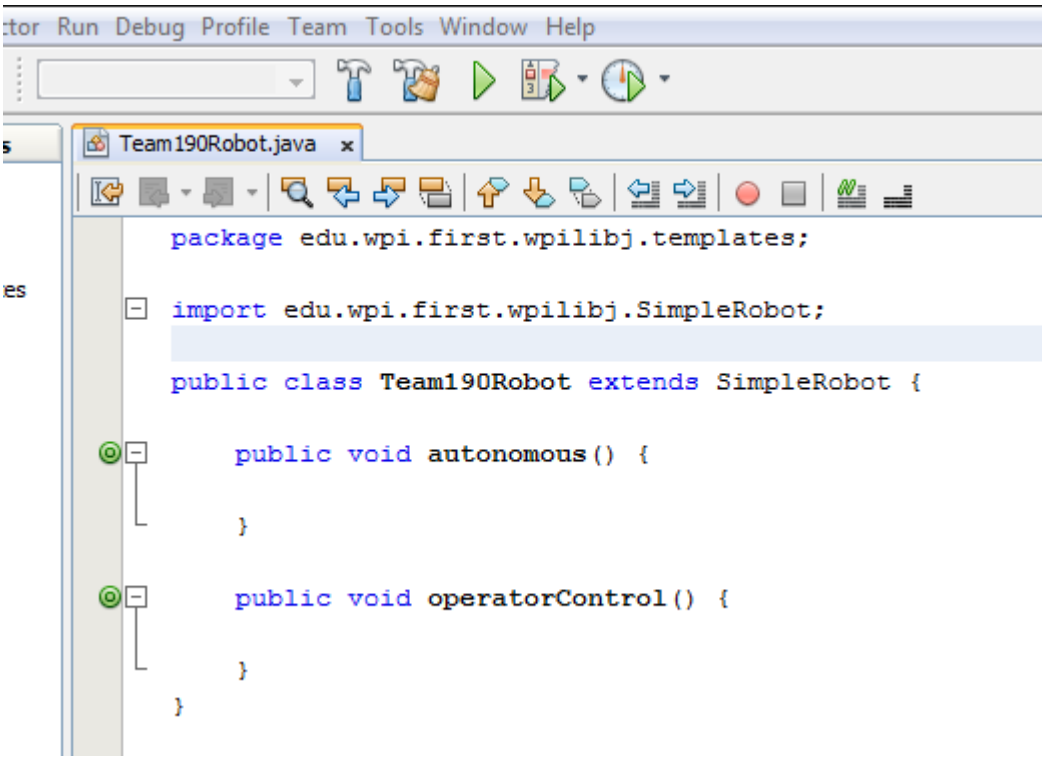
Notice that the wizard generated empty `autonomous()` and `operatorControl()` methods. The next step is to fill these methods in with the code you want to run for the autonomous and tele-operated field states, respectively. The `SimpleRobot` base class will automatically call these methods at the appropriate times.

Building the project

Be sure that the project you want to build is designated as the NetBeans "main project" by right clicking on the project in the Project pane and selecting "Set main project." The main project name will appear in bold text. Build the project by selecting the "Build main project" command in the Run menu or use the F11 shortcut. You'll see any build errors in the lower window under the source code.

Downloading and running the robot program

You can download the program to the robot by using the "Run main project" arrow in the toolbar or the "Run main project" item in the Run menu.



The Run command will do these steps:

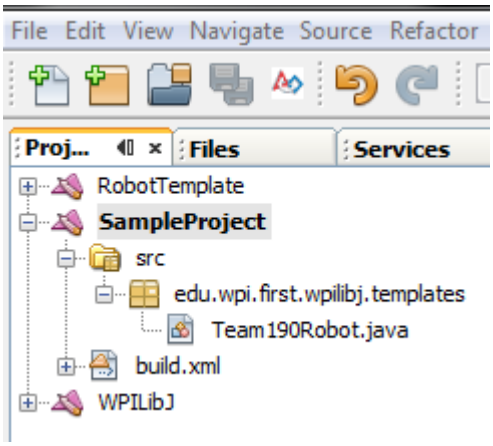
1. Connect to the cRIO and verify that the correct version of the FRC Java environment is loaded. If not, it will be updated before copying your robot program.
2. Copy your robot program to the cRIO and set it up to run on reboot.
3. Reboot the cRIO.
4. Wait for the cRIO to finish rebooting, and then connect to it so that console messages will appear in the NetBeans console window.

Be sure to enable the robot in either Autonomous or Tele-op mode using the driver station to see the program run.

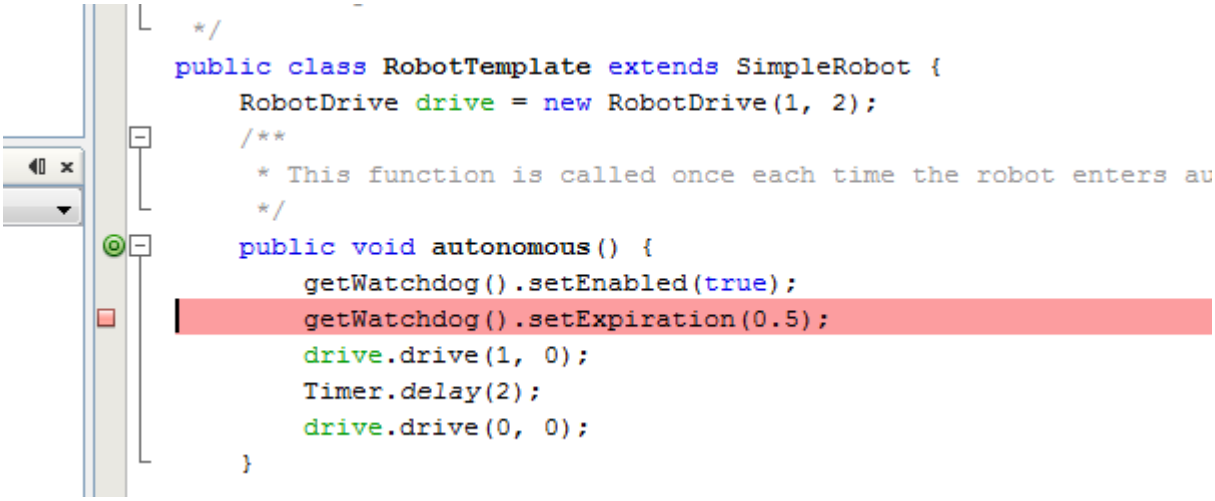
Debugging the robot program

Debugging the robot program is slightly more complex. The program has to start, and then you must attach the NetBeans debugger to the running program. The procedure is:

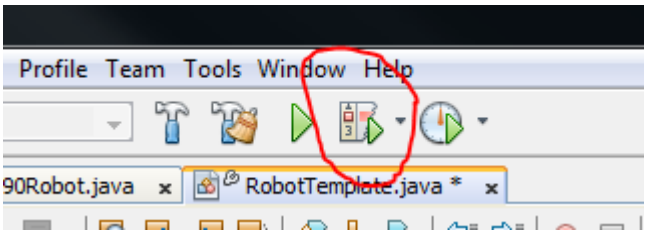
1. Make sure the project you want to debug is the main project (it will be bold in the Project pane)



2. Place a breakpoint that you expect to hit by clicking in the gray area to the left of the desired source code line. You can set more breakpoints, too.

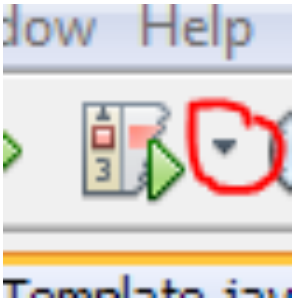


3. Click on the debug button in the toolbar.



4. Wait until the output window displays "Waiting for connection from debugger on serversocket://:2900. This is when the program will try to connect to the debugger.

5. Click on the down-arrow button adjacent to the debug toolbar icon and select "Attach debugger."



6. Make sure the debugger settings are as shown then hit the OK button:

The program will start running then pause at the first breakpoint it hits. You can then examine variables and set more breakpoints.

Creating a Robot Program

Consider a very simple robot program that has these characteristics:

Mode	Description
Autonomous period	Drives in a square pattern by driving at half speed for 2 seconds to make each side then turn 90 degrees. Do this 4 times.
Operator Control period	Uses two joysticks to provide tank steering for the robot.

Robot specifications:

Method	Port Location
Left drive motor	PWM port 1
Right drive motor	PWM port 2
Joystick	Driver station joystick ports 1 and 2

Starting with the simple template for a robot program:

```
package edu.wpi.first.wpilibj.templates;

import edu.wpi.first.wpilibj.RobotDrive;
import edu.wpi.first.wpilibj.SimpleRobot;
import edu.wpi.first.wpilibj.Timer;

public class RobotTemplate extends SimpleRobot {

    public void autonomous() {
    }

    public void operatorControl() {
    }
}
```

Now define a robot drive object for motors in ports 1 and 2 and joystick objects for joystick ports 1 and 2:

```
package edu.wpi.first.wpilibj.templates;

import edu.wpi.first.wpilibj.Joystick;
import edu.wpi.first.wpilibj.RobotDrive;
import edu.wpi.first.wpilibj.SimpleRobot;

public class RobotTemplate extends SimpleRobot {

    RobotDrive drive = new RobotDrive(1, 2);
    Joystick leftStick = new Joystick(1);
    Joystick rightStick = new Joystick(2);

    public void autonomous() {
    }

    public void operatorControl() {
    }
}
```

To simplify this example, we'll disable the watchdog timer. (The watchdog is a safety feature in the WPI Robotics Library that helps keep your robot from running away out of control if the program malfunctions. You don't really want to disable it. If necessary, give it a longish timeout.) In general you

should leave the watchdog enabled but for the sake of this first example, we'll disable it. This can be done in a constructor for your `RobotDemo` object:

```
public RobotDemo()
{
      drive.setSafetyEnabled(false);
}
```

Now write the autonomous part of the program to drive in a square:

```
    public void autonomous() {
        for (int i = 0; i < 4; i++) {
            drive.drive(0.5, 0.0);  // drive 50% forward speed with 0% turn
            Timer.delay(2.0);       // wait 2 seconds
            drive.drive(0.0, 0.75); // drive 0% forward with 75% turn
            Timer.delay(0.75);      // wait for the 90 degree turn to complete
        }
        drive.drive(0.0, 0.0);      // drive 0% forward with 0% turn (stop)
    }
```

Now write the operator control part of the program:

```
    public void operatorControl() {
        while (isOperatorControl() && isEnabled())    // loop during enabled teleop mode
        {
            drive.tankDrive(leftStick, rightStick);   // drive with the joysticks
            Timer.delay(0.005);
        }
    }
```

This applies joystick control values to the drive motors, every 5 milliseconds.

Putting it all together we get this very short program that accomplishes an autonomous task and provides operator control tank steering:

```
package edu.wpi.first.wpilibj.templates;

import edu.wpi.first.wpilibj.Joystick;
import edu.wpi.first.wpilibj.RobotDrive;
import edu.wpi.first.wpilibj.SimpleRobot;
import edu.wpi.first.wpilibj.Timer;

public class RobotTemplate extends SimpleRobot {

    RobotDrive drive = new RobotDrive(1, 2);
    Joystick leftStick = new Joystick(1);
    Joystick rightStick = new Joystick(2);

    public void autonomous() {
        for (int i = 0; i < 4; i++) {
            drive.drive(0.5, 0.0);  // drive 50% forward speed with 0% turn
            Timer.delay(2.0);       // wait 2 seconds
            drive.drive(0.0, 0.75); // drive 0% forward speed with 75% turn
            Timer.delay(0.75);      // wait for the 90 degree turn to complete
        }
        drive.drive(0.0, 0.0);      // drive 0% forward speed with 0% turn (stop)
    }

    public void operatorControl() {
        while (isOperatorControl() && isEnabled())    // loop during enabled teleop mode
        {
            drive.tankDrive(leftStick, rightStick);   // drive with the joysticks
            Timer.delay(0.005);
        }
    }
}
```

Some details:

- In this example **`drive`**, **`leftStick`**, and **`rightStick`** are member objects of the **`RobotDemo`** class.
- The **`drive.drive()`** method takes two parameters, a speed and a turn rate. See the documentation about the **`RobotDrive`** object for details on how the speed and direction parameters work.

Using objects

The WPI Robotics Library accesses all sensors, motors, driver station elements, and more through objects. Most objects correspond to the physical things on your robot like sensors. Objects have the code and the data needed to operate that physical thing. Let's look at a gyro. The operations (or methods) you can perform on a gyro are:

- Create the gyro object – this sets up gyro communications and calibrates the gyro
- Configure the gyro parameters, i.e. its sensitivity
- Get the current heading, or angle, from the gyro
- Reset the current heading to zero
- Delete the gyro object when you're done using it

Creating a gyro object is done like this:

```
Gyro robotHeadingGyro = new Gyro(1);
```

The variable **`robotHeadingGyro`** refers to a **`Gyro`** object that operates a gyro module connected to analog port 1. That's all you have to do to make an instance of a **`Gyro`** object.

Note: An instance of an object has a block of memory for that instance's data. When you create an object, that memory block gets allocated and when you delete the object that memory block gets deallocated.

To get the current heading from the gyro, you simply call the **`getAngle`** method of the gyro object. Calling this method is really just calling a function that works on the data specific to this gyro instance.

```
float heading = robotHeadingGyro.getAngle();
```

This sets the variable **`heading`** to the current heading of the gyro on analog channel 1.

WPI Robotics Library Conventions

This section documents some conventions used throughout the library to standardize its use and make things more understandable. Knowing these should make your programming job much easier.

Class, Method, and Variable Naming

Names follow these conventions:

Type of name	Naming rules	Examples
Class name	Initial upper case letter then camel case (mixed upper/lower case) except acronyms which are all upper case	`Victor, SimpleRobot, PWM`
Method name	Initial lower case letter then camel case	`isAutonomous, getAngle`
Member variable	"`m_`" followed by the member variable name starting with a lower case letter then camel case	`m_deleteSpeedControllers, m_sensitivity`
Local variable	Initial lower case	`targetAngle`

Constructors with Slots and Channels

In past releases you might have had **slot numbers in your code**. To make programs compatible between the 4-slot and 8-slot cRIO we have moved to a module numbering scheme. Rather than using the actual slot number a module is loaded in, the number is the instance of the module type. For example the first digital module would be 1 and the second one would be 2. If you only had a single module of each type in your robot and you used the short form of the constructors when creating devices (where the slot number argument was left out and defaulted to the first module) then your code doesn't have to change. The library will continue to default the numbers to the first module of a given type. If you explicitly passed the slot number to the constructors, then it will likely have to change. The module(s) in slot 4 of the 4-slot cRIO-FRC II or slots 5-7 of the 8-slot cRIO FRC will be referred to as Module 2.

For example:

```
Jaguar wristMotor = new Jaguar(2); // first digital module, pwm port 2
```

Since the slot number was implicit this won't have to change. But if you did this:

```
Jaguar wristMotor = new Jaguar(4, 2); // digital module slot 4, pwm 2
```

Then this will have to change to: new Jaguar(1, 2); // 1st digital module

Module Ordering

All teams will now be working with the new cRIO images which require the new module order. There is a new module order for 2012 to create some symmetry between the 8-slot and 4-slot cRIOs. The new module order is as follows:

Physical Slot number	Module number in your program	8-slot cRIO	4-slot cRIO
1	1	Analog Module 9201	Analog Module 9201
2	1	Digital Module 9403	Digital Module 9403
3	1	Solenoid Module 9472	Solenoid Module 9472
4	2	empty	Any module type
5	2	Analog Module 9201	NA
6	2	Digital Module 9403	NA
7	2	Solenoid Module 9472	NA
8		empty	NA

Examples are:

```
Jaguar(int channel)                 // channel with default slot (2)
Jaguar(int slot, int channel)       // channel and slot
Gyro(int slot, int channel)         // channel with explicit slot
Gyro(int channel)                   // channel with default slot (1)
```

Sharing inputs between objects

WPILib object constructors generally use port number(s) to select and reserve cRIO input and output channels. e.g. when you instantiate an encoder object, it reserves a digital input channel.

Built-in Robot classes

There are several built-in robot classes to help you quickly create a robot program. Subclass the one that best fits your requirements and preferences.

Note: The IterativeRobot template is the basis for Command-based programs that we'd like to see the beta teams explore. See the WPILib Programming Cookbook located in the Documents section of the 2012 WPILib project on FIRSTForge.

Table 1: Built-in robot base classes to create your own robot program.

Class name	Description
`SimpleRobot`	This template is the easiest to use and is designed for writing a straight-line autonomous routine without complex state machines. Pros: • There are only three places to put your code: the constructor for initialization, the `Autonomous` method for autonomous code and the `OperatorControl` method for teleop code. • Sequential robot programs are trivial to write. Just code each step one after another. • No state machine is required for multi-step operations. The program can simply do each step sequentially. Cons: • Switching between autonomous and operator control code may require rebooting the controller if your program gets stuck in a loop. • The `Autonomous` method will not quit running until it exits, so it will continue to run during the operator control period. You don't want that. So be sure to make your loops check if the field state is still in the autonomous period.
`IterativeRobot`	This template gives additional flexibility in the code for responding to various field state changes (autonomous, operator control, disabled) in exchange for additional complexity in your program. `IterativeRobot` repeatedly calls your methods depending on the current field state. The intent is that each method will do some processing for that field state and then return. That way, as soon as the field state changes, `IterativeRobot` starts calling a different method. Pros: • You get fine-grain control of field state changes, especially if you're practicing and retesting the same field state over and over. Cons: • It's more difficult to write action sequences that unfold over time, e.g. an autonomous sequence. That requires state variables to remember what the robot was doing from one call to the next.

`RobotBase`	The base class for the above classes. This provides all the basic functions for field control, the user watchdog timer, and robot status. Extend this class if you need more flexibility.

SimpleRobot class

The `SimpleRobot` class is designed to be the base class for a robot program with straightforward transitions from Autonomous to Operator Control periods. There are three methods to fill in to complete a `SimpleRobot` program.

Table 2: SimpleRobot class methods that are called as the field state progresses through the match.

Method	Description
the Constructor (method with the same name as the robot class)	Put code in the constructor to initialize sensors, motors, pneumatics, and your robot program variables. This code runs as soon as the robot is turned on, before it is enabled, that is, before it can operate motors and other actuators. When the constructor exits, the program will wait until the robot is enabled.
`autonomous()`	Put code in the `autonomous` method to run during the autonomous period of the match. When this method exits, the program will wait until the start of the operator control period. This method had better detect the end of the autonomous period since it won't be interrupted when it's time for operator control. If this method has an infinite loop, it won't stop until the entire match ends.
`operatorControl()`	Put code in the `operatorControl` method to run the robot during the operator control part of the match. This method will be called after the `autonomous` method has exited and the field has switched to the operator control part of the match. If your program exits from the `operatorControl` method, it will not resume until the robot is reset.

IterativeRobot class

The `IterativeRobot` class organizes your robot program (your subclass) into methods that it calls according to the match state. For example, it calls your `autonomousContinuous` method repeatedly during the autonomous period. When the playing field (or Driver Station setting) changes to operator control, it calls the `teleopInit` method then starts calling the `teleopContinuous` method repeatedly.

When basing a robot program on the `IterativeRobot` base class, you implement these methods:

Table 3: IterativeRobot calls these methods of your robot program as the match proceeds:

Method name	Description
`robotInit()`	Called when the robot is first turned on. You put initialization code here or in the constructor. This method is only called once.
`disabledInit()`	Called once each time the robot becomes disabled.
`autonomousInit()`	Called once when the match enters the autonomous period from any other state.

`teleopInit()`	Called once when the match enters the teleoperated period from any other state.
`disabledPeriodic()`	Called periodically during the disabled part of the match, using a periodic timer.
`autonomousPeriodic()`	Called periodically during the autonomous part of the match, using a periodic timer.
`teleopPeriodic()`	Called periodically during the teleoperated part of the match, using a periodic timer.
`disabledContinuous()`	Called continually during the disabled part of the match. When this method returns, it gets called again if the match state hasn't changed.
`autonomousContinuous()`	Called continually during the autonomous part of the match. When this method returns, it gets called again if the match state hasn't changed.
`teleopContinuous()`	Called continually during the teleoperated part of the match. When this method returns, it gets called again if the match state hasn't changed.

The three Init methods are called on transition into the relevant field state. The Continuous methods are called repeatedly while in that state, after calling the appropriate Init method. The Periodic methods are called periodically while in a given state. Call the `IterativeRobot` class's `setPeriod` method to set the period interval. The periodic methods are intended for time-based algorithms like PID control. During each match state, its periodic and continuous methods will both be called, at different rates.

RobotBase class

The `RobotBase` class is the superclass of the `SimpleRobot` and `IterativeRobot` classes. If you decide not to build on `SimpleRobot` or `IterativeRobot` , then subclass `RobotBase` directly. `RobotBase` has all the methods to determine the field state, set up the watchdog timer, handle communications, and do other housekeeping work.

Create a subclass of `RobotBase` and implement at least the `startCompetition` method, much like the `SimpleRobot` class does.

For example, the **SimpleRobot** class definition looks approximately like this:

```
public class SimpleRobot extends RobotBase {

    private boolean m_robotMainOverridden;

    public SimpleRobot() {
        super();
        m_robotMainOverridden = true;
    }

    public void autonomous() {      // supplied default autonomous()
        System.out.println("Provided autonomous() method running");
    }

    public void operatorControl() { // suppled default operatorControl()
        System.out.println("Provided operatorControl() method running");
    }

    public void robotMain() {       // supplied default robotMain()
        System.out.println("Information: No user-supplied robotMain()");
        m_robotMainOverridden = false;
    }

    public void startCompetition() {
        if (!m_robotMainOverridden) {
            // this is where the match sequencing happens
        }
    }
}
```

It overrides the **startCompetition** method that controls the running of the other methods and it adds the **autonomous**, **operatorControl**, and **robotMain** methods. The **startCompetition** method looks approximately like this:

```
    public void startCompetition() {
        robotMain();
        if (!m_robotMainOverridden) {
            while (true) {
                // Wait for robot to be enabled
                while (isDisabled()) {
                    Timer.delay(.01);
                }
                // Now enabled - check if we should run Autonomous code
                if (isAutonomous()) {
                    autonomous();
                    while (isAutonomous() && !isDisabled()) {
                        Timer.delay(.01);
                    }
                } else {
                    operatorControl();          // run the operator control method
                    while (isOperatorControl() && !isDisabled()) {
                        Timer.delay(.01);
                    }
                }
            }
        }
    }
```

It uses the **isDisabled** and **isAutonomous** methods of **RobotBase** to determine the field state, then calls the correct methods as the match progresses.

Similarly the **IterativeRobot** class calls a different set of methods as the match progresses.

MotorSafety and the Watchdog timer class

MotorSafety timers are allocated on each speed controller object and the RobotDrive class. You can individually enable and set expiration times for each motor and for the entire robot drive implemented with the RobotDrive object. Every time you send a value, even the same value as the previous value, the motor safety timer is reset. If no values are sent for a period longer than the expiration time for that device, that motor is disabled until a new value is sent. The RobotDrive class has the motor safety enabled by default and individual motors do not. If you want to use the motor safety option for individual motors, you must enable it on each one and set the expiration time. You can enable or disable motor safety for an individual device by calling the `setSafetyEnabled(enabled)` method, where enabled is replaced with either a true or false Boolean value.

The Watchdog timer class has been deprecated since 2011 but is still available. The watchdog timer is turned off by default and must be enabled explicitly to make use of it. The Watchdog timer will stop the robot's motors and pneumatics if the program goes into an infinite loop or crashes. A watchdog object is created inside the `RobotBase` class (the base class for all robot programs). Your robot program is responsible for "feeding" the watchdog periodically by calling the `feed()` method on the Watchdog. If you don't feed the Watchdog often enough, it will stop all of the robot's motors and pneumatics.

The default expiration time for the Watchdog is 500ms (0.5 second). Programs can override the default expiration time by calling the `setExpiration(expiration-time-in-seconds)` method on the Watchdog.

Using the Watchdog timer is recommended for safety, but it can be disabled. For example, during the autonomous period of a match the robot needs to drive for drive for 2 seconds then make a turn. The easiest way to do this is to start the robot driving, and then call `wait` to wait for 2 seconds. During the 2-second wait, it can't feed the Watchdog. In this case you could disable the Watchdog at the start of the `autonomous()` method and re-enable it at the end. *A better approach is to set a longer watchdog timeout period so you still get most of the watchdog protection.*

You can call `getWatchdog()` from any of the methods in a `RobotBase` subclass.

Waiting for 2 seconds has another problem: This robot program's teleoperated code will start up to 2 seconds late if autonomous mode ends near the start of those 2 wait seconds since the teleoperated code won't start until the `autonomous` method returns.

Advanced Programming Topics

Using Subversion with NetBeans

Subversion is a free source code management tool that is designed to track changes to a project as it is developed. You can save each revision of your code in a repository, go back to a previous revision, and compare revisions to see what changed. You should install a Subversion client if:

- You need access to the WPI Robotics Library source code installed on a Subversion server.
- You have your own Subversion server for your team projects. This is especially useful if your team has more than one programmer.

Getting the WPILib Source Code

The Java source code for WPILib is included with the NetBeans plugins (the Java update). Teams can access the source code by opening the WPILib source code project here: c:\Users\<username>\sunspotfrcsdk\WPILibJ.

This source code will always be updated to match the current installation of the plugins. The source code is always rebuilt as part of a FRC Java project so any changes made to that directory will impact the files downloaded to the cRIO. With that said, the recommended method of modifying the source code is to make a copy of the new class and add it to your project. That way updates to the Java plugins won't overwrite your customizations. Even better is to write a subclass of a WPILib class rather than changing WPILib.

Differences between C++ and Java

C++ and Java are very similar languages. In fact Java has its roots in C++. If you looked at a C++ or Java program from a distance, it would be hard to tell them apart. You'll find that if you can write a WPILib C++ program for your robot, then you can probably also write a Java program.

Language differences

There is a good detailed list of differences between the two languages on Wikipedia: http://en.wikipedia.org/wiki/Comparison_of_Java_and_C++. Here's a summary of the main differences that affect WPILib robot programs.

- C++ memory is allocated and freed manually, that is the programmer is required to allocate objects and delete them. In Java, you allocate objects the same way (via the `new` operator), but they get freed automatically when there are no more references to them. This greatly simplifies memory management for programmers.
- Java does not have pointers; only references to objects. All objects must be allocated with the `new` operator and are always referenced using the dot (.) operator, for example `gyro.getAngle()`. In C++ you have to manage the difference between pointers, references, and local instances of objects.
- C++ uses header files and a preprocessor for including declarations in parts of the program where they are needed. In Java this happens automatically and with much less trouble.
- C++ supports multiple inheritance where a class can be derived from several other classes combining the behavior of all of the base classes. In Java only single inheritance is supported, but it has interfaces to get most of the benefits of multiple inheritance without the complications.
- Java programs will check for array subscripts out of bounds, uninitialized references to objects, and other runtime errors that might occur a program. C++ will just crash if you make these goofs.
- C++ programs will run the fastest since it compiles to machine code for the cRIO's PowerPC processor. The Java virtual machine interprets byte codes.

WPILib differences

We made every attempt to make the transition between C++ and Java as easy as possible in both directions. All the classes and methods have the same names. There are some differences due to the differences in the languages and language conventions:

Item	C++	Java
Method name convention	Methods are named with an upper case first letter and then camel case after that, for example, `GetDistance().`	Methods are named with a lower case first letter then camel case after that, for example `getDistance().`
Utility functions	Call global utility functions like `delay(1.0)` (to wait for one second).	Java has no functions outside of classes so for example you call `Timer.delay(1.0)`.

Our version of Java

The Java virtual machine and implementation we are using is the Squawk platform based on the Java ME (micro edition) standard. Java ME is simplified version of Java designed for the limitations of embedded devices like the cRIO. As a result it doesn't have classes such as GUI classes that aren't useful in embedded programs. If you've done any Java programming in the past it was probably with the Java Standard Edition (SE). Some of the differences between SE and ME are:

- Dynamic class loading is not supported. No class loading or unloading at run time.
- Reflection (a way of manipulating classes while the program is running) is not supported.
- The Java compiler generates a series of byte codes for the virtual machine. Building a Java ME program runs compiler then does a "pre-verification" step. Pre-verification speeds up program loading process and keeps the Java virtual machine (JVM) smaller.
- Finalization is not implemented, that is, the JVM will not automatically call `finalize()` methods. If you need to run cleanup code, you must explicitly call a cleanup method.
- Java Native Interface (JNI) is not supported. JNI is a standard way means for Java programs to call C programs. The JVM does support a similar mechanism called JNA.
- Serialization and Remote Method Invocation (RMI) are not supported.
- User interface libraries (Swing and AWT) are absent.
- Threads are supported by thread groups are not.
- Since Java ME is based on an earlier version of Java SE (1.3), it doesn't include newer features such as generics, annotations, enums, varargs, and autoboxing.