Java™-enabled Robotics Primer

Overview

This primer provides a very brief overview of the Java programming language. If you are familiar with programming, particularly in C or C++, but have not programmed using Java this primer will help you become familiar with Java. If you haven't done much programming before you may want to purchase one of the many available books to help you master Java programming.

The `SimpleBot` and `SimpleBotPlus` examples included with RoboJDE™ are used to illustrate basic Java programming concepts applied to robotics.

Classes and Objects

Two fundamental constructs in object oriented programming and the Java language are the class and the object.

A class is a <u>type of thing</u>. A class is the software implementation of the thing. When you write a Java program, you are creating classes. `SimpleBot` and `SimpleBotPlus` are examples of classes. These classes represent a robot controller. Their implementation is the logic that controls the robot. Each class is usually contained in its own Java source file, for example, SimpleBot.java. Classes may be implemented inside of other classes (Inner Classes). However, it is convenient to initially think of a class as the software implemented in one Java source file.

An object is an <u>instance of a class</u>. Objects contain the run-time data for a specific instance of a class. Objects are created and destroyed as a Java program executes. In the `SimpleBot` example the class uses two `Motor` objects to control the robot, one for each of the robot's wheels. These objects behave similarly but refer to different motor ports on the robot controller board. Each of these `Motor` objects consists of a few bytes of memory containing data relevant to the object, including its class and the port it uses.

New objects are created by executing the constructor method. This is done by using the `new` keyword. In `SimpleBot`, the main method creates a `SimpleBot` object by using the statement:

```
SimpleBot simpleBot = new SimpleBot(…)
```

Member Variables and Methods

Classes define member variables and methods. Member variables contain data relevant to a particular object. Methods are functions that are relevant to the class and its objects.

Frequently member variables are kept "`private`" to the class so the only way for methods in other classes to interact with an object is through the object's methods. This keeps code outside of the class from knowing to many details about the inner workings of the class, making it easier to make changes that don't ripple throughout the code. In the `SimpleBot` class, the member variables which keep track of the `Motor` objects – `mLeftMotor` and `mRightMotor` – are private and not accessible to other classes. The only way for other classes to interact with a `SimpleBot` object's `Motor` objects is through its methods – `forward(), backward(), stop() and go()`.

Any program is much easier to maintain if changes and enhancements can be isolated to a small portion of the program. By hiding the internal structure of a class, changes to the inner workings of a class to extend it or add new features are isolated to the class itself. So long as the class's methods continue to do the same thing, other classes will not need to change. Other classes will be unaware of the internal structure of the modified class so they will not need to change. Additionally, by only allowing an object's methods to modify its member variables, it isn't possible for other parts of the program to improperly modify an object due to a programming error. Because all modifications are controlled by an object's own methods, it is in complete control of how and when it can be modified.

You will find your programs are easier to debug and maintain if you declare member variables `private`.

Static Methods and Member Variables

Methods and member variables are declared `static` if they are per-class rather than per-object.

A static member variable is a single variable per-class shared by all objects of the class. If a member variable is not static, it is per-object. Each object has its own instance of each non-static member variable. Changes to one object's member variables are independent of all other objects.

Methods that operate only on static member variables may be declared `static`. Static methods are related to a class but do not access object instance specific member variables. The `main()` methods in `SimpleBot` and `SimpleBotPlus` are `static` methods. These methods can be executed without being associated with an object.

References and "this"

Whenever an object's methods or member variables are accessed, a reference to the object tells the Java virtual machine which object to use. In the `SimpleBot` example, the `forward()` method executes the `setMotorPower()` method for the left motor by using the statement:

```
    mLeftMotor.setPower(Motor.MAX_FORWARD);
```

The member variable mLeftMotor is a reference to a Motor object.

When executing a non-static method, the reference is implicitly passed to the method and becomes the `this` reference while the method executes. When a method accesses member variables or other methods without specifying a reference, `this` is used implicitly. In `SimpleBot.go()`, the methods `forward()`, `stop()` and `back()` are not preceded by a reference, because `this` is used implicitly. The `SimpleBot.forward()` method implicitly uses `this` when referring to `mLeftMotor` and `mRightMotor`. `this` can be used explicitly, as well. Whether used implicitly or explicitly, the result is the same.

When an object needs to pass a reference to itself to another method, `this` tells the Java compiler to use the reference to the current object.

There is no `this` reference for static methods or static member variables. Static members in other classes are referred to by preceding the method or member with the class name. For example, SimpleBot's go method puts the current execution thread to sleep by calling the `sleep()` method in the `Thread` class using the statement:

```
    Thread.sleep(5000);
```

When calling static methods in the same class, the class name can be omitted because the compiler will implicitly assume the method is in the same class.

Inheritance

Inheritance allows one class to inherit the implementation of another class. The `SimpleBotPlus` class inherits the implementation of the `SimpleBot` class by declaring it extends `SimpleBot` using the statement:

```
    public class SimpleBotPlus extends SimpleBot
```

`SimpleBotPlus` overrides the `go()` method in `SimpleBot` by declaring its own go method to implement a different robot behavior. `SimpleBotPlus` inherits the methods `forward()`, `backward()` and `stop()` from `SimpleBot` because they do not need to change.

The `Object` Class

The root class of all classes is the class `Object`. All classes inherit from this class. If a class is declared without extending another class then it is implicitly a subclass of `Object`.

Interfaces and Abstract Classes

Interfaces specify a set of methods that define functionality that a class may declare it implements. By using interfaces, dependencies between classes can be reduced, making your software easier to maintain and extend. For example, the `SimpleBot` class maintains references to two objects for the left and right motors. These references refer to the `Motor` interface, allowing the `SimpleBot` class to work with any class that implements the `Motor` interface. By using the `Motor` interface the `SimpleBot` class is able to work with conventional motors, servo motors, or any class that implements the `Motor` interface, rather than being limited to only work with one type of motor.

Abstract classes are partially implemented classes that delegate the implementation of certain methods to subclasses. Since its implementation is incomplete, you cannot create an object of an abstract class, you must create objects of a subclass which completes the implementation. For example, the `Motor` interface used by `SimpleBot` could have been made an abstract class instead of an interface. The method `setPower()` would then have been declared `abstract`. A subclass of `Motor` would then implement the `setPower()` method.

Interfaces are less restrictive than abstract classes because any class can implement an interface. However, subclasses can inherit methods from abstract class. The methods declared by an interface are always abstract. That is, interfaces do not implement methods. Interfaces only declare methods that must be implemented by any class that declares it `implements` the interface.

Abstract classes are declared by using the `abstract` keyword in the class declaration. For example:

```
public abstract class A
```

Abstract methods are declared by using the `abstract` keyword in the method declaration and not including statements which would implement the method. For example:

```
public abstract void go();
```

The following is an example of an interface:

```
public interface AnInterface {
     public void go();
}
```

A similar abstract class could be:

```
public abstract class AnAbstractClass {
     public abstract void go();
     public void stop() {
          …
     }
}
```

The `stop()` method contains an implementation that can be inherited. The `go()` method is abstract and must be implemented by a subclass.

Access Keywords

The access to member variables and methods can be restricted using access keywords in the declaration of the member variable or method. The following keywords may be used:

- **`private`** – member variables and methods are only accessible to the declaring class.
- **no keyword** – member variables and methods are only accessible to the declaring class and classes in the same folder (package). Note: Folders are referred to as “packages” in the Java language.
- **`protected`** – member variables and methods are only accessible to the declaring class, subclasses of the declaring class and other classes in the same folder.
- **`public`** – member variables are accessible anywhere.

Variables

The Java language supports the following types of variables:

- **static member variables** – per-class variables – there is only one instance of the variable regardless of the number of objects.
- **member variables** – per-object variables - there is one instance of the variable for every object of the class.
- **local variables** – declared in methods – access is limited by scope, there is one instance of the variable within the scope the variable is defined in. Generally the scope is the set of braces – `{}` – the variable is declared between, or the entire method if the variable is an argument of the method. Local variables are accessible from the point where they are declared to the end of the scope. In `SimpleBot.main()`, `simpleBot` is a local variable and is only accessible in the `try` block it is declared in.

Data Types

The Java language supports the following data types:

- **`boolean`** – a variable with a `true` or `false` value. The values `true` and `false` are the only values a `boolean` can take on, unlike other languages where booleans are simply integers.
- **`char`** – a literal character such as ‘A’. `char` variables are unsigned 16 bit quantities that can be cast to an `int`.
- **`byte`** – a signed 8 bit integer variable.
- **`short`** – a signed 16 bit integer variable.
- **`int`** – a signed 32 bit integer variable.
- **`long`** – a signed 64 bit integer variable.
- **`float`** – a 32 bit floating point variable.
- **`double`** – a 64 bit floating point variable. Note: RoboJDE supports the double type but treats it as if it were a `float`. Therefore, `double` variables have the same range and precision as `float` variables when executing a program using the RoboJDE virtual machine.
- **`void`** – used to indicate a method does not return a value.
- **A reference to an object.** References are declared by preceding the variable name with a class name. For example, the statement

  ```
  SimpleBot simpleBot;
  ```

  declares the variable `simpleBot` is a reference to an object which is a `SimpleBot` class or a subclass of `SimpleBot`. References may also have the value `null`, which means the reference currently refers to no object. An exception is thrown if a program attempts to access a member variable or method using a reference that is null.
- **Arrays of the above data types, except void.** Arrays are declared by following the data type by square brackets, for example,

  ```
  int[] intArray;
  ```

  Multi-dimensional arrays are declared by using multiple sets of brackets. For example,

  ```
  int[][] twoDArray;
  ```

Program Structure

Java programs are structured in blocks of statements between braces – `{}` – similar to C and C++. The statements in a method are enclosed by braces, as are the statements in a branch of a conditional statement. If a conditional statement is followed by only one other statement, the braces can be omitted.

The `main()` Method

The program execution starts at the method named `main` in the “main class.” In RoboJDE you specify which class is the main class in the Project Properties dialog.

Comments

The Java language supports two methods of denoting comments:

- `//` - end-of-line comment. Everything after the `//` to the end-of-line is ignored by the compiler.
- `/* */` - block comment. Everything between `/*` and `*/` is ignored by the compiler.

Operators

Java supports the operators in Table 1.

Table 1 - Java Operators

Operators	**Description**
`+ -`	addition and subtraction
`* / %`	multiplication, division and modulo division
`++ --`	increment or decrement the operand – placed left of the operand for pre-increment/decrement (`++x`) and right of the operand for post-increment/decrement (`x++`).
`== !=`	equality and inequality – when applied to references equality is `true` if the compared references refer to the same object instance. Use the `equals()` method to check for equality of object values.

`>= <= > <`
`instanceof`
	greater than or equal to, less than or equal to, greater than, less than and test if an object is an instance of a particular class

`&& || !`	Boolean AND, OR and NOT
`(`*type*`)`	cast operand – convert the type of the operand to the *type* specified between the parentheses.

`<< >>`
`>>>`
	left shift, arithmetic right shift and logical right shift

`& | ^ ~`	bitwise AND, OR, exclusive OR and NOT
`? :`	conditional operator – if the value to the left of the `?` is `true` evaluate the expression to the left of the `:` otherwise evaluate the expression to the right of the colon. For example, the following statement can be used to find the maximum of x and y: `max = (x >= y) ? x : y`

`= *= /=`
`%= += -=`
`<<= >>=`
`>>>= &=`
`^= |=`
	assignment operators – operators other than = perform the operation denoted to the left of the equals sign then assign the result.

Conditional Statements

The Java language provides the following conditional statements:

- **`if`** (*expression*) - if the *expression* is `true`, the block of statements that follows is executed.
- **`else if`** (*expression*) - if the *expression* is `true` and the preceding `if` and `else if` expressions are `false`, the block of statements that follows is executed.
- **`else`** – if the preceding `if` and `else if` expressions are `false`, the block of statements that follows is executed.
- **`switch`** (*integer*) - branch to the case statement corresponding to the value of the *integer* variable or if no case with the corresponding value exists branch to the `default` keyword. If the `default` keyword is not present, branch to the end of the block.
- **`case`** *value*: - identifies the point to branch to if the *value* matches that of the integer used in the `switch` statement.
- **`default` –** identifies the point to branch to if the value in the `switch` statement does not correspond to any of the cases.

Loops

The Java language supports the following types of loops:

- **`for`** – iterate the loop a specified number of times.
  For example:

  ```
  for (int i = 0; i < 10; i++) { … }
  ```

  repeats the statements in the braces 10 times.
- **`while`** (*expression*) - iterate the loop while the *expression* is `true`.
  For example:

  ```
  while (i < 10) { … }
  ```

- **do** – similar to while loop but always executes the block of code at least once.
  For example:

  ```
  do { … } while (i < 10);
  ```

Exception Handling

The Java language supports exception handling, which allows for abruptly transferring control when an abnormal condition occurs. When a condition occurs that makes it necessary to discontinue the current program flow, such as a `null` reference or an out-of-bounds array index, an exception results and an object that is a subclass of `Throwable` is thrown. Control is then transferred to the first enclosing `catch` statement for which the thrown class is an instance of the class specified in the catch statement. If the current method does not catch the exception, then the calling method is checked, and so on up through the chain of calling methods until a suitable catch statement is found. If no suitable catch statement is found, the current thread silently exits. The structure of a try-catch-finally block is as follows:

```
try {
     …
}
catch (Throwable t) {
     …
}
finally {
     …
}
```

- **`try`** – denotes the start of a block of code where certain exceptions can be caught by following catch statements.
- **`catch`** `(`*throwable-class*`)` – catches any thrown exceptions of the specified class or any of its subclasses. A single `try` block may be followed by several `catch` blocks. When an exception occurs the catch statements are checked in order. The first catch statement for which the thrown object is an instance of the listed class determines the `catch` block that is executed.
- **`finally`** – denotes a block of code that should always execute regardless of whether an exception occurs or whether an exception is caught.

Transfer of Control

The following statements transfer execution out of the current block of statements.

- **`break`** – branch out of the enclosing `switch` statement or enclosing loop.
- **`continue`** – branch to the beginning of the enclosing loop and start the next iteration.
- **`return`** *value* – return from a method and return the specified value. No value can be returned if the method declaration states it returns void. The return statement can be omitted at the end of a method whose return type is `void`.
- **`throw`** *throwable-object* – transfer control to the first suitable `catch` statement passing the specified object to the catch block.
  For example:

```
throw new Exception(“Test exception”);
```

Casting and “`instanceof`”

The Java language is strongly typed and requires that any reference be cast to the desired class type rather than assuming an object is the desired class. The `cast` operation allows a super class reference to be cast to a subclass of the super class. If the reference refers to a class that is not an instance of the specified class a `ClassCastException` is thrown. The `instanceof` operator allows for testing the class of an object without throwing an exception.

- (*type*) – casting a reference or data type to the *type* specified.
  For example:

  ```
  Motor motor = (Motor)new ContinuousRotationServo(servo);
  ```

- **`instanceof`** – logical operation which is true if the object to the left is an instance of the class to the right. For example:

  ```
  if (motor instanceof HandyBoardMotor) {…}
  ```

Packages and Imports

Libraries of classes are organized as “packages.” A package is all the classes in a single file folder. The base Java classes are in the package `java.lang`. For the programs you develop using RoboJDE all of the classes in one project folder are in the default package.

When a file refers to a class or interface from another package the class must be imported using the import statement. For example,

```
import com.ridgesoft.robotics.Motor
```

could be used to import the `Motor` class. All classes in a package can be imported by using `*` instead of a particular class name in the import statement. In the `SimpleBot` example, the statement:

```
com.ridgesoft.robotics.*
```

imports all classes in the robotics package.