# Java Basics
# (Assuming Alice Background)

by

Dr. Mark C. Lewis

for

Introduction to Programming Logic

## Introduction

This document is intended to provide a quick overview to the Java programming language as well as some examples of how to use it. The Java language is actually a simple programming language, but like all programming languages, it is very picky about the rules being followed. This is in contrast to natural languages which are inevitably extremely complex, but don't have such problems if rules are occasionally broken. Of course, what matters most in this comparison is the audience the language is used to communicate with. Natural languages are used to communicate with other humans and generally other humans are forgiving of your mistakes and can get the needed information out of what is said, even if it isn't said quite right. Programming languages are used to give instructions to a computer to tell it what to do. This process needs to be formal and rigorous because there can't be ambiguity. Also, the computer is a simpler device than the human so you have to really follow the rules closely for it to understand what you are telling it to do.

## Basic Rules/Tips

There are some basic rules that can be laid out for the construction of a Java program. Some of them just make sense, like the fact that you need to match up parentheses. Others are more specific to Java, like the fact that all of your code needs to be placed inside of classes. We'll start by just giving a list of some of these rules that you will want to follow and then give explanations for some of them.

- Java code is constructed by writing classes. Each file needs to have one public class with the same name as the file.
- Classes can contain the properties of objects of that class and methods that give the behaviors of those objects.
- Every open parentheses, brace, or bracket needs to have a matching close.
- You can't put methods inside of methods so make sure you close off the curly braces for one method before starting another.
- Variable declarations have the format of a type followed by the name of the variable. Java allows you to declare variables anywhere inside of a method.

- Import statements are the only things you will write outside of classes and if you use Eclipse you can use Ctrl-Shift-O and Ctrl-Shift-M to do imports and it won't misplace them.
- We call methods on objects using the dot notation. The syntax is object.method(args). Even if there are no arguments you need the parentheses, but you can leave them blank.

When you start writing a statement in Java there generally aren't many options that you have. There are only a few things that are valid for writing a statement or an expression in Java. In most of your statements you will be either asking an object to do something or you will be asking an object for some information that you can evaluate or store in a variable. That means that you will start with the name of a variable that is in scope at that line. These are either parameters to the method, local variables, or properties. Generally you don't have many of these so there really aren't many options for what you can do in any given expression. Use that to help focus and simplify your thinking.

**Non-Object Oriented Programming**

Java is a primarily object oriented programming language, just like Alice was. What does that mean? Well, an object is a construct in a programming language that combines data and the functionality to manipulate that data. That is to say that an object is something that has properties and methods that operate on the properties. Despite the fact that Java is intended to be object oriented, not all Java programs have to be object oriented. The static keyword tells us that a method or property is associated with a class instead of with individual objects. Programs that only use static methods aren't really object oriented because they don't actually have to create new objects. While such programs don't use all of the benefits of the Java language, they can be simple ways of illustrating the logical structure we use to solve simple problems. For this reason, most of the early programs in the course only use static methods and we'll start with that in this document as well.

Let's run through a simple example of a program that we can write using several Java constructs in a non-object oriented way. This program reads numbers from the user and keeps reading until they enter zero. It will print out two averages. One is the average of the positive numbers and the other is the average of the negative numbers. Notice that all of this code can go

inside of a main. Basically, the description of the problem didn't really involve any “objects” so we don't need to be writing multiple classes. The only objects we will use are from the Java libraries for doing input and output. I'm going to leave out all the import statements in all code examples to save space. Eclipse can add them for you easily enough.

```
public class Averages {
    public static void main(String[] args) {
        double positiveSum=0,numPositive=0;
        double negativeSum=0,numNegative=0;
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter numbers. Type 0 to quit.");
        double num=sc.nextDouble();
        while(num!=0) {
            if(num>0) {
                positiveSum+=num;
                numPositive++;
            } else {
                negativeSum+=num;
                numNegative++;
            }
            num=sc.nextDouble();
        }
        if(numPositive>0) {
            System.out.println("The positives had an average of "+
                positiveSum/numPositive);
        } else {
            System.out.println("No positive numbers were entered.");
        }
        if(numNegative>0) {
            System.out.println("The negatives had an average of "+
                negativeSum/numNegative);
        } else {
            System.out.println("No negative numbers were entered.");
        }
    }
}
```

Let's walk through the logic involved in making this code. The program always starts in main and this is a fairly simple program so we just put everything inside of main. We know that we need to read in numbers so we make a Scanner object. We also know that to find an average of a bunch of numbers we have to have the sum of the numbers and the how many numbers there were. For this reason there are four doubles declared to keep the sums and counts for positive and negative numbers respectively. The only other variable we declare is num which we use to store the most recently read value.

Since we want to keep reading numbers until the user enters zero, we put in a while loop.

Reading from the user right before the loop and at the very end of the loop makes it so this works the way we want. Inside of the loop the code checks to see if the value that was read has a positive or negative values and increments the proper variables.

When you go through this code note that there are only a few things we can do on each line or in each expression. We can use constructs like while and if. We can print things with System.out. Alternately, everything else begins with one of the variables we have declared. Either we are calling a method on our Scanner object, sc, or we are working with one of the numeric variables.

Let's look at another example that is a bit larger so we can break the code up across multiple methods. Remember that the real key to programming is decomposition. Never try to approach a big problem straight up. Break it into pieces. The goal is to never do anything complex by taking all pieces of significant complexity and breaking them into pieces that are simple. For this problem we want to read a set of numbers from the user and print some statistics for those numbers. The statistics will require us to run through the numbers multiple times so we need to store them in an array. There are methods that calculate the mean, median, and standard deviations of the numbers. Note that the median method uses the sort method in the Arrays class to help us out. There are faster ways of finding the median, but this requires a lot less programming and is easily fast enough for any realistic number of values that a user is actually going to input by hand.

```
public class ArrayBasedStats {
    /**
     * This program reads a set of numbers from the user then calculates some
     * statistics related to those numbers.
     * @param args The command line arguments.
     */
    public static void main(String[] args) {
        Scanner sc=new Scanner(System.in);
        System.out.println("How many numbers do you want to use?");
        int howMany=sc.nextInt();
        double[] nums=new double[howMany];
        System.out.println("Enter the numbers.");
        for(int i=0; i<howMany; ++i) {
            nums[i]=sc.nextDouble();
        }
        System.out.println("The mean of those numbers is: "+mean(nums));
        System.out.println("The median of those numbers is: "+median(nums));
        System.out.println("The standard deviation of those numbers is: "+
                standardDeviation(nums));
    }
```

```
    /**
     * This method calculates the mean of an array of doubles.
     * @param nums The numbers to use in the calculation.
     * @return The mean of the numbers passed in.
     */
    public static double mean(double[] nums) {
        double sum=0.0;
        for(int i=0; i<nums.length; ++i) {
            sum+=nums[i];
        }
        return sum/nums.length;
    }

    /**
     * This method calculates the median of an array of doubles.
     * @param nums The numbers to use in the calculation.
     * @return The median of the numbers passed in.
     */
    public static double median(double[] nums) {
        Arrays.sort(nums);
        return nums[nums.length/2];
    }

    /**
     * This method calculates the standard deviation of an array of doubles.
     * @param nums The numbers to use in the calculation.
     * @return The standard deviation of the numbers passed in.
     */
    public static double standardDeviation(double[] nums) {
        double av=mean(nums);
        double sum=0.0;
        for(int i=0; i<nums.length; ++i) {
            sum+=(nums[i]-av)*(nums[i]-av);
        }
        return Math.sqrt(sum/nums.length);
    }
}
```

For illustrative purposes we will repeat this same example using lists instead of arrays. The primary objective here is to illustrate why we use arrays for some tasks. The syntax of array is simpler than for lists. Also, we can make arrays of primitive types. Notice that the list is actually a list of Double, not a list of double. This is because lists can only hold object types. Thanks to an addition in Java 5 called autoboxing, this doesn't impact out programming too much, but it does add significant overhead to the program when it is running because primitives are much more efficient that the wrapper classes. The only other significant changes are moving from [i] to .get[i], and from .length to .size().

```java
public class ListBasedStats {
    /**
     * This program reads a set of numbers from the user then calculates some
     * statistics related to those numbers.
     * @param args The command line arguments.
     */
    public static void main(String[] args) {
        Scanner sc=new Scanner(System.in);
        System.out.println("How many numbers do you want to use?");
        int howMany=sc.nextInt();
        List<Double> nums=new ArrayList<Double>();
        System.out.println("Enter the numbers.");
        for(int i=0; i<howMany; ++i) {
            nums.add(sc.nextDouble());
        }
        System.out.println("The mean of those numbers is: "+mean(nums));
        System.out.println("The median of those numbers is: "+median(nums));
        System.out.println("The standard deviation of those numbers is: "+
                standardDeviation(nums));
    }

    /**
     * This method calculates the mean of an array of doubles.
     * @param nums The numbers to use in the calculation.
     * @return The mean of the numbers passed in.
     */
    public static double mean(List<Double> nums) {
        double sum=0.0;
        for(int i=0; i<nums.size(); ++i) {
            sum+=nums.get(i);
        }
        return sum/nums.size();
    }

    /**
     * This method calculates the median of an array of doubles.
     * @param nums The numbers to use in the calculation.
     * @return The median of the numbers passed in.
     */
    public static double median(List<Double> nums) {
        Collections.sort(nums);
        return nums.get(nums.size()/2);
    }

    /**
     * This method calculates the standard deviation of an array of doubles.
     * @param nums The numbers to use in the calculation.
     * @return The standard deviation of the numbers passed in.
     */
    public static double standardDeviation(List<Double> nums) {
        double av=mean(nums);
        double sum=0.0;
        for(int i=0; i<nums.size(); ++i) {
            sum+=(nums.get(i)-av)*(nums.get(i)-av);
        }
        return Math.sqrt(sum/nums.size());
```

```
    }
}
```

**Object Oriented Programming**

The real power of Java comes about when we create multiple classes and begin to instantiate objects. The classes give us another level in which to break up our problems. So we aren't just doing a functional decomposition by making different methods, we can group logical pieces into larger constructs that allow us to represent the objects that take part in our programs.

To begin this process and see how it works we first need to decide what problem we are going to solve. For this example we will look at a simple inventory program for a retail outlet. The program will pull from two files. One file is a product list that keeps track of all of the products that the store sells. The other is an inventory list that keeps track of everything that the store currently has on inventory. We will create three classes for this program. We need one class that represents a product, one that represents the current inventory of a product, and another that will have the main to represent the full program.

We'll begin with the Product class. This class has properties for the name, id, and price of the product. You can find those at the bottom of the class. At the top we start with two different constructor. One will build a product by reading from a Scanner and the other allows us to pass values directly in. We'll find a need for both of them in the main application. Between these two we have a number of different methods that could be useful to us when interacting with Product objects. This includes a method to write products out to file as well as methods to get each of the different fields. The comments on each of the methods describe what they are to be used for.

```
public class Product {
    /**
     * A constructor to read the product from a scanner.  The format it
     * expects is to have the id code followed be an int price in cents
     * and the name.  The id and the price have to bbe single tokens,
     * but the name can have multiple words and includes everything else
     * on the line.
     * @param sc The Scanner to read from.
     */
    public Product(Scanner sc) {
        idCode=sc.next();
        price=sc.nextInt();
```

```
    name=sc.nextLine().trim();
}

/**
 * A constructor that sets the values directly.
 * @param n The name of the product.
 * @param id The id code of the product.
 * @param p The price of the product in cents.
 */
public Product(String n,String id,int p) {
    name=n;
    idCode=id;
    price=p;
}

/**
 * This method writes a line to the specified writer recording the id,
 * price, and name.
 * @param writer
 */
public void writeProduct(PrintStream writer) {
    writer.println(idCode+" "+price+" "+name);
}

/**
 * This method returns whether the specified ID matches the ID of this
 * product.
 * @param id The ID to check.
 * @return True if the passes string matches the products ID.
 */
public boolean checkID(String id) {
    return id.equals(idCode);
}

/**
 * This method returns a well formated string for the product.
 */
public String toString() {
    return "Product: "+idCode+" "+getPriceString()+" - "+name;
}

/**
 * Returns the name of the product.
 * @return The name of the product.
 */
public String getName() {
    return name;
}

/**
 * Returns the ID code of the product.
 * @return The ID code of the product.
 */
public String getID() {
    return idCode;
}
```

```
    /**
     * Returns the price of the product in cents.
     * @return The price of the product in cents.
     */
    public int getPrice() {
        return price;
    }

    /**
     * Formats the price of the product as a string with the dollars and cents
     * separated.
     * @return The price in a string format.
     */
    public String getPriceString() {
        String dollars=Integer.toString(price/100);
        String cents=Integer.toString(price%100);
        if(cents.length()>2) cents="0"+cents;
        return "$"+dollars+"."+cents;
    }

    private String name;
    private String idCode;
    private int price;
}
```

Our next class is one that will actually be used to keep inventory. It is a simpler class than the Product class because it uses the Product class. This class stores the number of items we have in inventory of a particular product. The properties of the class include a Product object and an int to store the number of them. Note the use of an object type that we created as a property. This is a common thing to see in Java and it lets us keep the code simple. This way we don't duplicate for from the Product class here. Instead, we have this store a reference to the Product it is related to. There are a number of different methods in addition to the constructors. For this class they are mostly straight foward.

```
public class InventoryItem {
    /**
     * This constructor builds an inventory item with the given product
     * and zero items.
     * @param p The product for this item.
     */
    public InventoryItem(Product p) {
        product=p;
    }

    /**
     * This constructor builds an inventory item with the given product
     * and the given number of items.
     * @param p The product for this item.
```

```
     * @param n The number of the items in inventory.
     */
    public InventoryItem(Product p,int n) {
        product=p;
        number=n;
    }

    /**
     * Method to write this item out in the format needed for the file.
     * @param writer The stream to write to.
     */
    public void writeItem(PrintStream writer) {
        writer.println(product.getID()+" "+number);
    }

    /**
     * Method to convert the inventory record to a string.
     */
    public String toString() {
        return number+" - "+product;
    }

    /**
     * Returns the number of items of htis product in inventory.
     * @return Number of items.
     */
    public int getNumber() {
        return number;
    }

    /**
     * Changes the number of items for this product.
     * @param change The difference to apply to the inventory count.
     */
    public void alterNumber(int change) {
        number+=change;
    }

    /**
     * Determines is this inventory item matches the specified ID code.
     * @param id The ID code to match.
     * @return A boolean for whether the product matches this id.
     */
    public boolean matchID(String id) {
        return product.checkID(id);
    }

    private Product product;
    private int number=0;
}
```

The last class in this example contains the main and is where we really do most of the work for the program. In order to make this as object oriented as possible we've done something that you haven't seen much. We have the main create an object of that type and then most of the

processing happens through that object. In this case we create an object and have a variable in main called im that references the object. The constructor calls two methods that read in files for the products and the inventory. The main method then calls the runMenu method on im and most of the program happens in this method. The runMenu method has a do-while loop in which we display a menu and get the response from the user. There are then a series of if statements that select the right activity based on the value the user inputs. Each of these calls one of the other methods to help it do what it needs to do.

```
public class InventoryMain {
    /**
     * This is the main for running the inventory progam.
     * @param args The command line arguments.
     */
    public static void main(String[] args) {
        InventoryMain im=new InventoryMain();
        im.runMenu();
    }

    /**
     * The constructor for our inventory program.  This just reads in the
     * files so that we have the starting list of products and the original
     * inventory.
     */
    public InventoryMain() {
        readProducts();
        readInventory();
    }

    /**
     * This is where most of the work happens in the program.  This method
     * contains a loop that runs through repeatedly displaying a menu until
     * the user selects to stop.  It calls other methods as needed to
     * complete the tasks that the user requests.
     */
    public void runMenu() {
        int option;
        Scanner sc=new Scanner(System.in);
        do {
            System.out.println("Select an option:");
            System.out.println("1. Add Product");
            System.out.println("2. Save Products");
            System.out.println("3. Print Products");
            System.out.println("4. Add Inventory");
            System.out.println("5. Sell Items");
            System.out.println("6. Save Inventory");
            System.out.println("7. Print Inventory");
            System.out.println("8. Quit");
            option=sc.nextInt();
            if(option==1) {
                addProduct(sc);
```

```
            } if(option==2) {
                try {
                    writeProducts(new PrintStream(new File("products.txt")));
                } catch(IOException e) {
                    e.printStackTrace();
                }
            } if(option==3) {
                writeProductsPretty(System.out);
            } if(option==4) {
                System.out.println("What is the code of the product you are
adding?");
                changeInventory(sc,1,"How many are you adding?");
            } if(option==5) {
                System.out.println("What is the code of the product you are
selling?");
                changeInventory(sc,-1,"How many have been sold?");
            } if(option==6) {
                try {
                    writeInventory(new PrintStream(new
File("inventory.txt")));
                } catch(IOException e) {
                    e.printStackTrace();
                }
            } if(option==7) {
                writeInventoryPretty(System.out);
            }
        } while(option<8);
    }

    /**
     * This method reads in the products from file.
     */
    public void readProducts() {
        products=new ArrayList<Product>();
        File file=new File("products.txt");
        try {
            Scanner input=new Scanner(file);
            while(input.hasNext()) {
                products.add(new Product(input));
            }
            input.close();
        } catch (FileNotFoundException e) {
            // No file yet so we simply don't have any products.
        }
    }

    /**
     * This methods reads in the inventory from file.
     */
    public void readInventory() {
        inventory=new ArrayList<InventoryItem>();
        File file=new File("inventory.txt");
        try {
            Scanner input=new Scanner(file);
            while(input.hasNext()) {
                String id=input.next();
                int number=input.nextInt();
```

```
                Product p=findProductByID(products,id);
                if(p!=null) {
                    inventory.add(new InventoryItem(p,number));
                } else {
                    System.err.println("Bad ID code in inventory.");
                }
            }
            input.close();
        } catch (FileNotFoundException e) {
            // No file yet.  We don't yet have an inventory.
        }
    }

    /**
     * This is a helper method that runs through the product list to
     * return the product that matches a specific ID.
     * @param products The list of products.
     * @param id The ID code that we are looking for.
     * @return The matching product or null if there is no match.
     */
    public Product findProductByID(List<Product> products,String id) {
        for(Product p:products) {
            if(p.checkID(id)) return p;
        }
        return null;
    }

    /**
     * This method takes user input and adds a new product to the product
     * list.
     * @param sc The Scanner to read user input from.
     */
    public void addProduct(Scanner sc) {
        System.out.println("Enter the ID code, price, and name of the item.");
        products.add(new Product(sc));
    }

    /**
     * A method to write out all the products to a writer.  The format
     * matches what should be in the data file.
     * @param writer
     */
    public void writeProducts(PrintStream writer) {
        for(Product p:products) {
            p.writeProduct(writer);
        }
    }

    /**
     * A method to write out all the products to a writer.  The format is
     * good for users to see.
     * @param writer
     */
    public void writeProductsPretty(PrintStream writer) {
        for(Product p:products) {
            writer.println(p);
        }
```

```
    }

    /**
     * This method adds items to or removes items from the inventory.
     * @param sc A scanner for reading user input.
     * @param factor This should be 1 if adding items and -1 is removing.
     * @param numMessage The message to print when asking number of items.
     */
    public void changeInventory(Scanner sc,int factor,String numMessage) {
        String id=sc.next();
        InventoryItem invMatch=null;
        for(int i=0; i<inventory.size() && invMatch==null; ++i) {
            if(inventory.get(i).matchID(id)) {
                invMatch=inventory.get(i);
            }
        }
        if(invMatch==null) {
            if(factor>0) {
                Product prodMatch=findProductByID(products,id);
                if(prodMatch==null) {
                    System.out.println("That product ID is not valid.");
                } else {
                    System.out.println(numMessage);
                    int num=sc.nextInt();
                    if(num<1) {
                        System.out.println("You aren't allowed to have fewer
than one item in this operation.");
                    } else {
                        inventory.add(new InventoryItem(prodMatch,num));
                    }
                }
            } else {
                System.out.println("You don't have of those in inventory to
sell.");
            }
        } else {
            System.out.println(numMessage);
            int num=sc.nextInt();
            if(num<1) {
                System.out.println("You aren't allowed to have fewer than one
item in this operation.");
            } else {
                invMatch.alterNumber(factor*num);
            }
        }
    }

    /**
     * A method to print the inventory in a format that matches the data
     * files.
     * @param writer The PrintStream to print to.
     */
    public void writeInventory(PrintStream writer) {
        for(InventoryItem item:inventory) {
            item.writeItem(writer);
        }
    }
```

```
    /**
     * A method to print the inventory in a format that us good for users
     * to see.
     * @param writer The PrintStream to print to.
     */
    public void writeInventoryPretty(PrintStream writer) {
        for(InventoryItem item:inventory) {
            writer.println(item);
        }
    }

    private List<Product> products;
    private List<InventoryItem> inventory;
}
```

Most of the methods that runMenu calls on have a loop that either goes through the products or the inventory. Sometimes a for-each style loop is used and in other situations a normal for loop is used. The most complex of the methods is the changeInventory method. This method is called when we add items to the inventory as well as when items are sold off. Let's step through the way that it works. It begins by asking the user for the ID code of the item we want to change the inventory on. Once we have this code we need to see if our inventory already has a record for this product ID so there is a loop that runs through all the inventory items and checks if any of them matches. If one does then we can simply alter the number in that record appropriately. If not then we have to check if there is a product in our product list with that code. If there is, we create a new inventory item for that product with the proper number of items. There are multiple paths through this code where we don't do anything.

As it happens, one check has been left out of this method. It is possible to do something that results in a negative number of items. It is left to the reader to figure out what this is and to determine what code should be altered to fix this.

**Simple Syntax Rules**

Your code goes in files and you get one public class per file. Import statements, if needed, go before the class body. Below is a template for a basic class. Notice that everything goes inside the curly braces and they have to match.

```
public class ClassName {
      // Constructors, Methods, and Properties
```

```
}
```

The properties are nothing more than variable declarations at the class level. They tell us what the objects of that class store. We will always make them private.

```
private Type propertyName;
```

The type can be replaced with any primitive type such as `int`, `double`, `boolean`, or `char`, or it can be replaced with a reference type. All classes are reference types in Java, whether we write them or they are part of the standard libraries (API). `String` is a reference type that we commonly use in our programs.

Methods can be public or private. We typically make them public. They can also be static or non-static depending on whether they are associated with the class as a whole or the objects make from that class. A method in a class has this general form:

```
public [static] ReturnType methodName(ArgType1 arg1,ArgType2 arg2,...) {
      // Code for the method goes in here.
}
```

If there are no arguments, you leave the parentheses empty, but the parentheses still need to be there. In the case of a method, the return type can be void if the method doesn't return anything. If the method does return something, all paths through the method must end in a return statement. Note that things in square brackets are optional. I will use that formalism in all of the example syntaxes.

There are some special methods that we can include in classes called constructors. Constructors are called any time we call new to make a new object. We make new objects using the new operator. The following expression allocates and builds a new object.

```
new ReferenceType(arguments)
```

The arguments that are passed in should match something that is accepted by one of the constructors for that particular class. Because this is an expression that returns an object it is

generally used as part of a statement, either one that includes an assignment to a variable so that we can later refer to the object, or one that calls a method and this new object is an argument for that method. There can be other uses as this can be placed anywhere a reference to `ReferenceType` is needed, but the above are the most common.

Methods and constructors are made from a bunch of different statements. In this section we run through the different statements that you can put inside of methods. Let's run through the types of statements that you can write in Java to look at their syntax. The first type of statement that we have is the variable declaration, which has the following form.

```
Type variableName;
```

This statement declares a variable of the specified type with the specified name. In this context, the variable will have a local scope. That is to say that we can use it anywhere from the statement where it is declared until the end of the block of code it is declared in. Blocks of code are made with curly braces. You can only use a given variable name once inside of a given scope. Variable names can be the same as property names, but this gets really confusing and can lead to hard to find bugs so I recommend against doing that.

Many times you will want to give your new variable a value at the point where you declare it. This uses the following syntax.

```
Type variableName=expression;
```

Java doesn't let you use an uninitialized variable. This is a good thing, but there might be times when the compiler can't tell that a certain assignment line will definitely happen before the variable is used and it will give you an error. This happens occasionally when a variable is set in a pretest loop. In this situation you should just give the variable a value when you declare it to make the compiler happy. Note that this is in contrast to properties which are given default values if you don't initialize them. Numeric properties will be set to zero and reference properties will be set to null by default.

While on the topic of assignments and giving values to variables, you can also make a statement in Java from any assignment expression. There are many different syntaxes for this.

The most common form is shown here.

```
variable=expression;
```

The other forms of assignment expressions involve the shortcut operators that do assignment along with some type of numeric, boolean, or bitwise operation. For example, you can use ++ or += to increment numbers in an assignment. The expression can be arbitrarily complex as long as it has a type that can be stored in the specified variable.

You can also make a statement out of any method invocation by putting a semicolon after it. The syntax for this usually looks like the following.

```
object.method(arg1,arg2,arg3,...);
```

Static methods are generally called with the name of the class instead of the name of an object. In fact, you will get a warning if you call a static method using an object. Of course, methods can return objects and you can invoke methods on those objects that are returned. This leads to having an object followed by a dot and a method name, followed by another dot and a method name. This pattern can be repeated arbitrarily many times and it isn't uncommon to see several dots in an expression holding multiple object returning methods.

Java has a number of different statements for flow control as well. These include both conditional statements and loops. The following is a list of the syntax for the different flow control statements. Note that the curly braces are always optional in Java and can be replaced by a single statement. However, in most situations it will make your code more readable and keep things better organized if you leave the curly braces in. Normally the curly braces will be on different lines and all the statements between them will be indented.

if statements:

```
if(condition) { ... }
if(condition) { ... } else { ... }
```

switch statement:

```
switch(intExpression) {
     case intLiteral1:
          ...
          break;
     case intLiteral2:
          ...
          break;
     ...
     default:
          ...
}
```

while loop statement:

```
while(condition) { ... }
```

for loop statements:

```
for(initializer; condition; iterator) { ... }
for(Type varName:collection) { ... }
```

Example for loops:

```
for(int i=0; i<max; i++) { ... }
for(String s:arrayOfString) { ... }
for(Product p:listOfProducts) { ... }
```

do-while loop statement:

```
do { ... } while(condition);
```

In these statements, when you see `condition`, it needs to be a boolean expression. The switch statement only works with integer values (that includes the char type). In the case of the normal for loop, you can actually leave out parts of it, but the semicolons have to be there even if you leave one of the elements out.

One last topic that is worth describing in this section is that of type casts. There are certain situations where you will have an expression of one type and you need to it be another. Not all type casts are valid, but sometimes they are required. An example of this is if you have a double and you need to use it in a place that requires an int. To cast an expression you simply put the type you want the expression to be in parentheses in front of the expression.

**Real Syntax**

Obviously, Java has a lot more to the syntax than what was listed in the last section. Below is a link to a page that gives the full syntax for the Java language. This type of specification is what is used to create a compiler for a language. The the syntax provides the formal rules of the language. Those are rules that all valid programs in the language must follow.

http://java.sun.com/docs/books/jls/second_edition/html/syntax.doc.html

This is part of the more complete Java Language Specification.

http://java.sun.com/docs/books/jls/second_edition/html/jTOC.doc.html

**Other Topics**

There are quite a few topics that this document has not covered that are worth knowing about. Most of them deal with the libraries and not the languages. One of the main language features not discussed here are inner classes. Those are covered is the "From C to Java" document and your text book. On the surface these are just classes that you write inside of other classes, but they have a lot more capabilities and possibilities than is communicated by that simple description. Generics are another language topic not dealt with here. The angle braces used with lists are an example of generics. For the purposes of this class, we will not be concerned with generics beyond their role in specifying the types of collections in the java.util library. The interested student can learn more about generics by reading "From C to Java".

The feature that really makes Java stand out is not the language. (Though the relative simplicity of the language is nice for educational purposes.) What really stands out about Java are the libraries. This document does not go into them at all and for that information you are directed toward other sources and ultimately, the API, which documents all of the libraries.