A QUICK LOOK AT

Java 6 Fundamentals

First Edition

Krishna Mohan Koyya

Glarimy Technology Services
106, Vars Casa Rosa, Pai Layout,
BANGALORE – 560 016
India

www.glarimy.com

A Quick Look At

Java 6 Fundamentals

First Edition

Contents

About the Book

This is book is part of the Glarimy's QuickLook series and by nature it is quick, short and to the point. These titles are meant for Java practitioners, learners and enthusiasts and not for sale.

On-line Version

The whole content of this book is also available on-line in HTML format and PDF downloads at www.glarimy.com/quicklook.

Code

The code used as part of the illustrations in this book is available on-line as a jar file at www.glarimy.com/quicklook. The jar files contain both source code and compiled classes.

Comments and Suggestions

Your comments and suggestions are always appreciated to improve the quality and utility of these books. Please post your comments at quicklook@glarimy.com.

Preface

Welcome to *A Quick Look At Java 6 Fundamentals*. This is book is aimed to help the serious programmers who want to get into Java 6, *quickly*. When I say serious programmers, I refer to the programmers who not only just read the technology but apply the theory and explore the technology it in their own way.

The book designed and organized to understand the concepts *quickly* and to see the illustrations immediately and then to explore further beyond the concepts. Thus, each of the chapters is divided into three parts

- Understanding the Concepts
- Applying the Concepts
- Beyond the Concepts

True to their names, the part of *Understanding the Concepts* takes you through the concepts, quickly. It presents the problem that a feature of Java addresses, the solution and other related facts. The part of *Applying the Concepts* deals with the code. It presents a full working code supplemented with the explanation and output. This cements your understanding of the concepts. Once you are armed with the concepts, then the third part *Beyond the Concepts* takes you through few interesting facets and questions. Some of them are answered and some of them left for you to explore by writing code, going through the Java documentation or just by thinking further.

All you need to use this book is JDK 1.6 and any IDE of your choice. You are expected to be good at basic programming fundamentals. It would be easier for you to grasp the topics if you have good understanding of operating systems.

Hope this book helps in enhancing your interest and confidence in writing basic programs in Java, again, *quickly*.

Krishna Mohan Koyya
Bangalore
3rd June 2011

About the Author

Krishna Mohan Koyya has been associated with the software industry for the last 14 years as a developer, teacher, trainer and author. His primary interests in technology include Java, object oriented architectures and Web 2.0.

As a software developer he worked with OpenView technology at Hewlett-Packard, Lucent's GSM OSS at Wipro Technologies and CiscoWorks NMS platform at Cisco Systems nearly for 10 years.

He taught Software Engineering at Sasi Institute of Engineering and Technology, Tadepalligudem for an year before taking up training and consulting as full time career in 2008.

As a trainer and consultant he handles most of the Java technologies ranging from JSE, JEE, Spring Framework, Struts Framework, JAX-WS, OSGi and etc and other technologies like Dojo framework, Mash-ups, UML and etc., Few of the clients to whom he delivered the services include Hewlett-Packard, Lucent-Alcatel, Intel, Sapient, IBM, Cap Gemini, Samsung, Oracle, Wipro, L&T and MindTree.

Currently he heads Glarimy Technology Services, Bangalore, India.

Academically he holds a bachelors in Electronics and Communications Engineering and masters in Computer Science and Technology, both from Andhra University, Visakhapatnam, India.

He can be reached at krishna@glarimy.com or at www.glarimy.com/krishna

Chapter 1
Introducing Java

- Programming with Java
- Java Virtual Machine and Byte Code
- Java Runtime Environment (JRE)
- Java Development Kit (JDK)
- Setting the Classpath
- Writing and compiling Java program
- Running a Java program

Understanding the Concepts

Contrary to popular opinion, Java is not just a programming language. Its actually an architecture. Its goal is to develop platform independent applications. For example, as a developer, you need not to worry about the size of an integer on the target platform when you are writing Java code. The size of integer in Java is independent of the target platform. Let's see how it is possible?

Java Virtual Machine

It is possible because of JVM or Java Virtual Machine. JVM is a software-only microprocessor, in a way. All the Java code that you write will have to be compiled to run on this processor. The processor has its own architecture and instruction set like all other processors. Only difference is that there is no hardware implementation. Its a *virtual* machine.

Byte code

Well, you write a program and want to compile it against the JVM. Use `javac` compiler just to do that, as long as your program is written in Java. The code that is generated out of the compilation is object code against the JVM and it is known as *bytecode*. It comprises of the instructions to the virtual machine, not to the real machine and the real platform can not understand the bytecode (think, why?). How to resolve this issue?

Java Runtime Environment

- Some one must be there to translate the bytecode instructions to the real platform specific machine instructions.
- Some one must simulate virtual machine on the real machine, at run time.
- Some one must start the JVM to run your application.

That someone is JRE or *Java Runtime Environment*. By using the JDK tool `java`, you can start the JVM and submit your application to run.

Why all this?

By using JVM, JRE and the byte code we are writing Java programs. But, just why we choose Java? Here are quick advantages of Java.

- With this architecture, you are not really bothered about the real platform. You just write the code by keeping JVM in mind. That is platform independence.
- If something goes wrong, your machine will not be hurt. Its only the JVM crashes. That is robustness.
- Your program will not manage or access the memory of the real machine directly. That is lot of security.

What all this means? Using Java, you can write *platform independent* applications that are *robust* and *secure*.

Plus, Java programming language is fully *object oriented*. Thats a bonus.

Java Development Kit

The JDK or Java Development Kit comes with several tools that are required to develop Java applications. They include `javac, java, jar, javadoc` and etc.,

Remember, to run the applications, you just need the JRE. However, to develop applications using Java programming language, you need the JDK and its tools.

Make sure the JRE is always latest compared to the compiler. Having both the JRE and compiler of the same version is the best option, you know.

Finally, the path & classpath

An operating system locates an executable by searching in the folders listed as part of the environmental variable called PATH. Similarly, the JVM finds the bytecode by searching the folders listed as part of the variable called CLASSPATH.

Once you understood JDK, JVM, JRE, `javac`, `java` and environmental variables, you are all set to write your first Java program.

Applying the Concepts

Objective

To install and set-up the Java working environment.

Procedure

Installing and configuring JDK

Download JDK from http://www.oracle.com/technetwork/java/javase/downloads/index.html. Choose the appropriate flavor depending on your machine and operating system. I am using windows XP as the operating system.

Install JDK. By default it gets installed in the folder “Java” under “Program Files” unless you change it while installing. This location is what we call as 'home' of Java on the machine.

Once this is done, update the environment variable PATH. It should now include the bin folder of Java. Make sure, you are just only adding the new folder, but not deleting any of the folders from the path definition. That meas, on my machine, the path variable would be containing c:\\Program Files\Java\jdk1.6.0_14\bin as the first entry;

Once the path is updated, open a new command prompt and type the following command:

```
set path
```

This is to check if the variable is properly updated. The final test is by typing the following command to check the versions of the JDK that you have installed.

```
javac -version
java -version
```

Setting up the working directories

Create a folder where you are going to save all your Java programs. You can use any name for the folder. I chose `workspace` as the name for this demonstration. Under this folder create two folders named `src` and `bin`. The folder `src` is where all our source code goes into. The `bin` is the folder where the bytecode would be saved.

Setting up the CLASSPATH

Add a new environmental variable called CLASSPATH whose value would be the fully qualified path of the `bin` folder you just created. On my machine, it is `D:\workspace\bin`.
Open a new command prompt and type the following:

```
set classpath
```

This is to check if the CLASSPATH is properly defined.

Writing the first program

Open notepad or any other text editor to type in the code of the following listing. Save the file as `PrintDemo.java` in the `src` folder.

Listing: PrintDemo.java

```
1. class PrintDemo {
2.     public static void main(String[] args) {
3.         System.out.print("Welcome to Java! ");
4.         System.out.println("");
5.     }
6. }
```

Explanation

Compiling the program: Go to the `src` folder on the command prompt and type the following command:

```
javac -d ..\bin PrintDemo
```

If you have written the code exactly like what is listed, you would get the command prompt back typically within less than a second. Check the folder `bin` to find the class `PrintDemo.class`.

Results

Running the program: On the command prompt, from anywhere, type the following command shown in **bold** to get the output shown in regular font.

```
java PrintDemo.
Welcome to Java!
```

Beyond the Concepts

Interesting Questions

Is it only Java programming language using which we generate the bytecode?

No. As long as you could generate the bytecode, it doesn't matter which programming language you are using. However, then, it is your responsibility to find the appropriate compiler that can generate the bytecode from that language. Looks great, isn't it?

Is it important to have the CLASSPATH variable defined?

No. the current directory of the JVM is taken as the CLASSPATH, by default. That means, as long as you are running the `java` command from the folder where you have the bytecode, (class files) you need not to have CLASSPATH variable set.

However, in case you have several class files scattered around many directories, its a good idea to have the CLASSPATH defined. It allows you to run the applications from anywhere on the machines.

What happens if the versions of `javac` and `java` are different?

`javac` generates the bytecode against a particular version of JVM.

For example, assume that your `javac` compiles against JVM of version 1.5. As long as the JRE understands this bytecode, it can translate it for the real platform. That means, if the JRE is of version 1.5 or later, it can understand it.

Instead, if JRE is of version 1.3, it may not understand the bytecode of 1.5 version. One can backward compatible, but not possible to become forward compatible, isn't it?

Is updating the path variable a must?

Not at all, as long as you have the patience of typing the `javac` command with its full path. However, even if you have patience, it still make sense to save time by updating the variable, once for all.

The placement of the opening and closing braces does matter in Java? What happens If we keep the opening brace in a new line like we do in C?

Technically there is no problem. It is just a matter of style. We use the Java style like all most all of the professional Java developers.

Coding Exercises

1. Change the `String` to `string` (case changed) in the above code and compile. See the impact.
2. Change the `args` to `params` or any other string of your choice. Any impact? Why?
3. Change the name of the file from `PrintDemo.java` to `FirstDemo.java`. Any impact? Why?

Quiz

1. If the `java` command of JRE 1.3 is used to run the class files compiled by the `javac` of JDK 1.4, will it work?
2. How you can add current folder also as one of the CLASSAPTH entry?
3. How is the `-d` option of `javac` command helpful?

Chapter 2
Working with Primitive Data Types

- Java support for data types
- Declaring and initializing variables
- Handling different types
- Printing values of variables
- Type casting

Understanding the Concepts

Java is a strongly typed language. It implies that, you will have to declare and intialize the variable before you actually use it. The declaration must include the name of the variable and data type that it holds. Java supports various data types that fall in to different groups.

Data Types

The following table presents the supported data types, groups to which they belong to, size of memory they occupy and range of values that they can hold.

Group	Type	Memory	Min Value	Max Value
Integer	byte	1 byte	-128	127
	short	2 bytes		
	int	4 bytes		
	long	8 bytes		
Float	float			
	double			
Char	char	2 bytes		
Logical	boolean	1 byte		

Recollect from the previous discussion that the memory size of a data type does not depend on the underlying platform, as all these sizes are with respect to the virtual machine.

Declaring and initializing the variables

There are several ways in which you can declare and initialize the variables.

You can declare and initialize each variable on separate lines. A semicolon must terminate the lines.

```
int a;  //declaring a variable named 'a' as an 'int'
a = 12;  //initializing a declared variable 'a' with value 12.
```

You can declare and initialize a variable in the same line.

```
int a = 23; //declaring and initializing the variable 'a'.
```

You can declare several variables of same type in the same line. Remember to separate them using comma.

```
int a, b, c;  //declaring variables a, b, c.
```

You can declare several variables of the same type and initialize some or all of them in the same line.

```
int a , b = 20, c = 45, d = 109; //initializing only b, c and d.
```

Working with `long` and `float` types

When assigning a long or float value to an appropriate variable, the prefix `L` and `F` must be used respectively.

```
long d = 3456712345673L;
float e = 3.5F;
```

Working with `boolean` types

A variable of type `boolean` accepts only the values `true` or `false`. These values are keywords and must not be enclosed in quotes. Remember that no other values are accepted including `0`, `NULL`, `-1` and etc.,

Working with `character` types and strings

The value of a `character` variable must be enclosed in a pair of single quotes. Remember that this type takes only one character as the value except in the case of escape sequences.

```
char g = 'a';
char newline = '\n';
```

In case, the value should consist of more than one characters, then the variable must be declared as `String`. `String` is not a data type, in true sense. We would discus about `String` in detail later. For now, we just use them.

A string can be declared and initialized separately or on the same line like any other type. String values must be enclosed in a pair of double quotes. You can assign a value as a literal or an object (for now, do not worry about what an object is).

```
// declaring I as a string and assigning a literal value.
String i = "Glarimy Technology Services";

// declaring j as a string and assigning a string object as the value.
String j = new String("Glarimy Technology Services");
```

From the functionality point of view, both of the above lines deliver the same.

Printing the values of variables

Values of variables can be printed using the method `System.out.println()` or `System.out.print()`. The first one terminates the line after printing whereas the later do not.

For example, the following line prints the value of a declared and initialized variable `a`. The variable can be of any type including a `String`.

```
System.out.println(a);
```

In case, you want to print a variable along with some supporting text, you can use `+` operator.

```
// printing the value of 'a' along with some text.
System.out.println("Value of variable a : " + a);
```

The operator `+` concatenates a string with anything else. This operator can also be used for arithmetic operations as we will see later.

Type Casting

Can you assign the value of a variable to another variable? Why not? You can assign/initialize a variable with any value as long as it satisfies the following rule.

The value and the variable must be of the same type.

For example, an `int` value can be assigned to a variable of type `int`. Nothing strange! However, some times, you may want to assign a `short` value or a `long` value to an `int` variable.

Think about it. An `int` variable have enough memory to store a value of `short` type? The answer is obvious yes. Hence, the following code just works.

```
short a = 25;
int b = a; //works because b is an int, large enough to hold short
```

Now, think about an another scenario. A `byte` variable have enough memory to store a value of `int` type? The answer is equally obvious no (look at the table). Hence the following code doesn't work.

```
int a = 128;
byte b = a; // doesn't work. Byte is not large enough to hold an int.
```

The above code doesn't work even if the value of `a` happens to be just 10. The compiler just checks the types, not the actual values. As long as you are trying to store/copy the value of bigger type into a variable of smaller type, compilation fails with an error. The compiler thinks that you are *narrowing* the value unknowingly.

Why any one wants to write such code? Wait, there is scenarios where you may want to do it. In such cases, you have to force an explicit narrow down what is known as *typecasting*. Look at the following:

```
int a = 128;
byte b = (byte) a; // typecasting a to narrow down it to a byte
```

This time the compiler would not complain. It obeys you.

However, guess what is the value really gets stored into the variable `b`? It can not be 128, for sure as it does not fit in to a `byte`. It actually stores -128. (This about bit shifting!). That means, by typecasting, there is a chance of loss of data precision. If its OK with you, no issues.

Summary is this: You can store smaller values in variables of bigger types (*widening*). You can store bigger values in variables of smaller types (*narrowing*) by explicit typecasting (this kind of typecasting is also known as down-casting) at the risk of loosing the precision.

Applying the Concepts

Objective

To declare and initialize variables of various types and print their values.

Listing: DataTypeDemo.java

```
1. public class DataTypeDemo {
2.
3.     public static void main(String[] args) {
4.         byte a = 1;
5.         short b = 200;
6.         int c = 250000;
7.         long d = 3456712345673L;
8.         float e = 3.5F;
9.         double f = 40.56;
10.        char g = 'a';
11.        boolean h = false;
12.
13.        String i = "Glarimy";
14.        String j = new String("Glarimy Technology Services");
15.
16.        System.out.println("Value of variable a : " + a);
```

```
17.        System.out.println("Value of variable b : " + b);
18.        System.out.println("Value of variable c : " + c);
19.        System.out.println("Value of variable d : " + d);
20.        System.out.println("Value of variable e : " + e);
21.        System.out.println("Value of variable f : " + f);
22.        System.out.println("Value of variable g : " + g);
23.        System.out.println("Value of variable h : " + h);
24.        System.out.println("Value of variable i : " + i);
25.        System.out.println("Value of variable j : " + j);
26.
27.        a = (byte)b;
28.        c = b;
29.        System.out.println("a after assigning b: " + a);
30.        System.out.println("c after assigning b: " + c);
31.    }
32.}
```

Explanation

Every program of Java must be written as part of a *class*. Here we are writing a class called `DataTypeDemo`. The code starts from the `main()` method.

We are declaring and initializing various variables from line number 4 to 14 and printing them from line 16 to 25.

At line 27, we are down casting `b` to assign it to `a`.
At line 28, we are simply assigning `b` to `c`. No explicit casting is required.

Last two lines are printing the changed values of `a` and `c`. Guess what? The down casting of `short` to `byte` results in precision loss.

Results

Running the above program results in the following output.

```
Value of variable a : 1
Value of variable b : 200
Value of variable c : 250000
Value of variable d : 3456712345673
Value of variable e : 3.5
Value of variable f : 40.56
Value of variable g : a
Value of variable h : false
Value of variable i : Glarimy
Value of variable j : Glarimy
a after assigning b: -56
c after assigning b: 200
```

You may try changing the values assigned to the variables and the formatting instructions to see the impact.

Going Beyond the Concepts

Interesting Questions

Can we cast a `String` to an `int` explicitly?

No. By casting explicitly, we can satisfy the compiler, but not the JVM. Only the following castings work.

Any integer type to any other integer/float/char types.
Any float to any other integer/float types.
Any char to integer types.

`Strings` and `boolean` values can not be casted to other types, at run time.

Can we declare a variable twice?

No. A variable once declared, can not be redeclared in the same scope. We look at scope in detail later. For now, just assume that, once a variable is declared in the main method, it can not be redeclared in the same method. This is true, even if you are trying to redeclare it with the same type.

Quiz

1. What would you do if you want to store a value that is larger than that fits in a `long`?
2. An integer enclosed by double quotes should be declared as a `String` or an `int`?
3. What is the difference between treatment of `boolean` types in Java and in C?
4. Are the variable names are case sensitive?

Exercises

1. Write a Java program to declare, store and print your age, height and weight.
2. Write a Java program to declare, store and print your name, phone number, city name and pin code.

Chapter 3
Arithmetic Operations

- Operations and operators
- Expressions and Statements
- Data types of expressions
- Typecasting the expressions
- Unary operations

Understanding the Concepts

Operations and Operators

You know it already. To do arithmetic operations, you need numbers. The most basic arithmetic operations are addition, subtraction, multiplication and division. Other operations include finding the reminder after dividing two numbers (finding modulus), incrementing a number or decrementing a number. The following table presents the operators for each of these operations.

Operation	Operator
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulus	%
Increment	++
Decrementing	--

Expressions and Statements

An expression is a group of one or more variables or literals along with operators applied on them. An arithmetic expression consists of only numbers and operators. A number can be of any type belonging to integer or float group. The numbers can be supplied literally or by using variables.

The following is an arithmetic expression with two variables (a and b), two operators (+ and -) and one literal (2).

```
2 + a – b;
```

Once the expression is resolved during execution, it's result is obtained and optionally stored in other variable. A statement is where the result of the expression is stored in another variable, as shown below:

```
c = 2 + a – b;
```

Data type of an expression

In Java, every expression belongs to the type of the value to which it is resolved to. So, how do you know the type of an expression when you have more than one operand in it? It's simple. Look at the type of the each of the operands of the expression. Each of them belonging to one or other type. Pick the type belonging to the highest order. That's the type of the expression.

For example, the type of the following expressions is `int`.

```
int a = 5;
int b = 2;
int p = a / b;   // p holds the quotient of a and b.
```

Both `a` and `b` are declared as `int` types. So is the variable `p` that is receiving the expression value. Guess the value of `p`! Its going to be 2 (not 2.5).

This is because the largest data type of the operands in the expression is `int` and thus the expression itself is of type `int`. Decimal points are not allowed in the integers and there by the result is just 2.

Type casting the expressions

What if the `p` is declared as `double` in the above example?

```
int a = 5;
int b = 2;
double p = a / b; //value of integer expression is assigned to double
```

Guess the value of `p`, now. It's still not going to be 2.5. It is 2.0. This is because the expression is still an integer expression that results in 2. The variable `p` holds it as 2.0 as it is declared as `double`. The result is still not accurate. Now look at the following code:

```
int a = 5;
int b = 2;
double p = (double) a / b; // expression is promoted to double
```

We are casting the type of `a` to `double` in the expression. As `double` is the larger type of the operands in the expression, the type of the expression turns out to be `double`. Hence, it resolves to 2.5. As `p` is also `double`, it would have the correct value of 2.5 after executing this line.

Unary Operations

Using the unary operator `++` a number can be incremented by 1 and using `--` a number can be decremented by 1. However, positioning the operator is important. Let's how it is!

The following line is incrementing the value of `a` *after* using the current value of `a`.

```
int a = 10;
int b = a++; // post incrementing a
```

After executing these two lines, the value of `b` would be 10 and value of a would be 11. This is because `a` is first used to assign itself to `b` and only then it is incremented. This is called *post-incrementing*.

However, the following line is incrementing the value of `a` before using the current value of `a`.

```
int a = 10;
int b = ++a; // pre incrementing a
```

After executing these two lines, both `a` and `b` are going to be 11. Think why? This is called *pre-incrementing*.

Same is true with the decrementing as well.

Now, consider this. You want to add the value of `b` to `a` and store the sum back in `a`, as follows.

```
int a = 10;
int b = 20;
a = a + b; // a is assigned with sum of a and b
```

The same can be done using self-adding operator like the following:

```
int a = 10;
int b = 20;
a  +=  b;  // a is assigned with the sum of a and b
```

Applying the Concepts

Objective

To demonstrate the arithmetic operations on two integers.

Listing: ArithmeticDemo.java

```
1. public class ArithmeticDemo {
2.
3.     public static void main(String[] args) {
4.         int number = 10;
5.         int operand = 4;
```

```
6.
7.          int sum = number + operand;
8.          int diff = number - operand;
9.          int product = number * operand;
10.         double quotient = (double) number / operand;
11.         int reminder = number % operand;
12.
13.         System.out.println(number + "+" + operand + "=" + sum);
14.         System.out.println(number + "-" + operand + "=" + diff);
15.         System.out.println(number + "*" + operand + "=" + product);
16.         System.out.println(number + "/" + operand + "=" + quotient);
17.         System.out.println(number + "%" + operand + "=" + reminder);
18.         System.out.println("Number before incrementing: " + number);
19.         System.out.println("Pre-incrementing: " + ++number);
20.         System.out.println("Post-incrementing: " + number++ );
21.         System.out.println("After post incrementing: " + number);
22.         number--;
23.         System.out.println("Decrementing the number: " + number);
24.     }
25. }
```

Explanation

This code is straight forward. It declares two integers and applies all possible arithmetic operations on them. I believe that it is self-explanatory when you run and see the results.

Results

```
10 + 4 = 14
10 - 4 = 6
10 * 4 = 40
10 / 4 = 2.5
10 % 4 = 2
Value of the number before incrementing: 10
Pre incrementing the number: 11
Post Incrementing the number: 11
After post incrementing the number: 12
Decrementing the number: 11
```

Beyond the Concepts

Interesting Questions

Does Java support Operator Overloading?

The answer is Yes and No. Java has few operators doing multiple things. For example, + can be used as an arithmetic operator or as concatenating operator for strings. However, as a developer, you can not change/overload the behavior of any of the operators.

Adding two integers would always result in an integer?

Adding two integers may result in a sum which may be larger than an integer. However, if the sum is stored in an integer, the sum would be down casted to an integer thus loosing precision.

How about the priorities of operators?

The same BDMAS rule is applicable here as well. First the brackets are resolved. Then divisions and multiplications are resolved. And finally additions and subtractions are resolved.

What is the result of the following expression?

```
System.out.println("sum is " + 24 + 25);
```

```
sum is 2425.
```

If an expression is having at least one string, the `+` would behave as a concatenation operator only. However,

```
System.out.println("sum is" + (24+25));
```

would print

```
sum is 49.
```

This is because the summation happened in the brackets before concatenating the string.

Quiz

1. Can you increment a `boolean` value?
2. Can you add `char` values to `int` values?

Coding Exercises

1. Write a program to compute the area and perimeter of a rectangle where length is 250 meters and width is 100 meters.
2. Write a program to convert temperature from centigrade to Fahrenheit.
3. Write a program to print the average of 10 various numbers.

Chapter 4
Auto-boxing: Working with wrappers

- Objects
- Wrappers
- Auto-boxing
- Retrieving primitives from wrappers

Understanding the Concepts

Java is an object oriented language and hence we except the application that we develop contains just a groups of interacting objects. But, after all, what are objects?

Objects

We have seen `int` values which are primitive data and `String` values which are objects. A primitive data just holds value whereas an object holds one or more values along with methods to access and modify the values. Do you remember the methods of `String` that we have used?

You may think arrays also as objects because they also hold more than one piece of primitive data. However, there is a basic difference between an object and an array. An array holds collection of values of single data type (*homogeneous* values) whereas object holds collection of values of various data types (*heterogeneous* values). That way, arrays are objects of special case.

If you have experience in programming using C, you may also think Java objects are lot like C structures. Again, there is a difference. C structures only hold values whereas Java objects hold values as well as methods to access and modify those values. Not only in Java, but in every object oriented programming language, an object contains data values (*state*) and methods (*behavior*).

We dwell more into objects later. At this point, it is sufficient to understand that the primitive data types that we have seen so far are not objects!

Wrappers

At some point of time, for various valid reasons that we would see in future, it may become necessary to represent even a primitive integer value also as an object.

To make our life simpler, Java provides objects corresponding to each of the primitive data types. For example, Java provides `Integer` which is an object version of primitive type `int`. These object versions are called as *wrappers*. We can also get the actual primitive value from its wrapper.

The following table presents the primitive types and their wrappers.

Primitive Type	Wrapper
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Auto-boxing

The operation of converting data from the primitive form to its wrapper form is called *boxing* and reverse operation is called *unboxing*.

The current version of Java does these operations of boxing and unboxing automatically and hence they call the that feature as *auto-boxing*. By this name we refer to the fact that Java internally converts the primitive data and wrappers seamlessly. It means, you can add a primitive `int` with wrapper `Integer` without explicit boxing or unboxing. Java takes care of it on behalf of you.

Have a look at the following code:

```
Integer a = new Integer(56);
int b = 20;
int sum = a + b;
Integer difference = a - b;
```

It just works just fine though we are using `Integer` wrappers and `int` values in the same expression without explicit boxing.

Retrieving the primitive values from Wrappers

Every wrapper is having methods to retrieve the primitive value. More over, you can actually get the primitive values in various data types.

The following code snippet is retrieving the `int` value out of the wrapper. It also tries to get the primitive value as a `double` as well.

```
Integer a = new Integer(56);
```

```
int b = a.intValue();
double c = a.doubleValue();
```

Applying the Concepts

Objective

To repeat the previous illustration using Wrappers.

Listing: AutoBoxDemo.java

```
1. public class AutoBoxDemo {
2.
3.     public static void main(String[] args) {
4.         Integer number = 10;
5.         int operand = 4;
6.
7.         System.out.println("Value as int: " + number.intValue());
8.         System.out.println("Value as Integer: " + number);
9.
10.        int sum = number + operand;
11.        int diff = number - operand;
12.        int product = number * operand;
13.        Double quotient = (double) number / operand;
14.        int reminder = number % operand;
15.
16.        System.out.println(number + "+" + operand + "=" + sum);
17.        System.out.println(number + "-" + operand + "=" + diff);
18.        System.out.println(number + "*" + operand + "=" + product);
19.        System.out.println(number + "/" + operand + "=" + quotient);
20.        System.out.println(number + "%" + operand + "=" + reminder);
21.        System.out.println("Number before incrementing: " + number);
22.        System.out.println("Pre incrementing: " + ++number);
23.        System.out.println("Post Incrementing: " + number++ );
24.        System.out.println("After post incrementing: " + number);
25.        number--;
26.        System.out.println("Decrementing the number: " + number);
27.    }
28.}
```

Explanation

Observe the line 4. Here the number is declared as an `Integer` (not `int`). At line 7, we are retrieving the value of `int` out of this `Integer`. At line 8, JDK is doing that for us using auto-boxing.

At lines from 10 to 14, we are operating on wrapper (number) and primitive value (operand). Still, we need not do any additional coding, it just happens because of auto-boxing.

Results

The results are self-explanatory.

```
Value as int: 10
Value as Integer: 10
10+4=14
10-4=6
10*4=40
10/4=2.5
10%4=2
Number before incrementing: 10
Pre incrementing: 11
Post Incrementing: 11
After post incrementing: 12
Decrementing the number: 11
```

Beyond the Concepts

Interesting Questions

What is the wrapper version for `String`?

The auto-boxing is meant for interchanging the format of a data between primitive version and wrapper versions. Hence, it does not apply to `String`. `String` is already an object, not a primitive data type.

Can a primitive value be casted to its wrapper?

Yes. Vice-versa is also true. But, auto-boxing obviates the need for casting them explicitly.

Is it a good idea to write the program using just wrappers to make it really object oriented?

No. Primitive data types are built into core of the Java. Their wrappers consume little more processing and memory footprint. Though cost is not prohibitive, some times it may be significant depending on the frequency of usage of a wrapper. Continue to use primitive types unless there is a real need for them to be wrappers.

Quiz

1. Which version of Java introduces the feature of Auto-boxing?

Coding Exercises

1. Write a program to compute the area and perimeter of a rectangle where length is 250 meters and width is 100 meters. Use only wrappers.
2. Write a program to convert temperature from centigrade to Fahrenheit. Use only wrappers.
3. Write a program to print the average of 10 various numbers. Us only wrappers.

Chapter 5
Making decisions

- Checking a condition
- Choosing alternative paths
- Nesting the conditions

Understanding the Concepts

This topic should be known to you already, if you have little exposure to any of the programming languages. If you are coming from language like C, then you may not find anything different (except on one note).

Making a decision and proceeding to a path of logic flow is common in many programs. This is done by using a statement like `if`. In case, the requirement is to choose between two different paths based on a decision, you may use `if … else` statement. Its the same thing in Java as well.

Checking a condition

The statement `if` always checks a condition. The condition must be resolved to a `boolean` value. (In C, this is not the case. The condition can be resolved to an integer, there).

The following code checks if the value of a variable `n` is equal to `25` or not. If its `true`, it prints *Success*.

```
if (n == 25){    // the operator == checks equality of operands
   System.out.println("Success"); //true block
}
```

In case the condition is satisfied, the block immediately following the condition gets executed. Such a block is called *true block*. A *block* starts with `{` and ends with `}`. In case the block contains only one statement (like in the above code), these block delimiters are just optional.

Choosing alternatives paths

In case, you want to choose between two blocks based on a condition, you use the `else` clause of the `if` statement, as follows:

```
if (n==25) {
    System.out.println("Success"); // true block
else
    System.out.println("Failed"); // false block
```

In case `n` is `25`, the *true block* gets executed to print *Success*. Otherwise, the *false block* prints *Failed*.

Going further

Look at the following. The snippet is first checking a condition. In the case of failure, it is checking another condition.

```
if (n==25) {
    System.out.println("Success");
else if (n == 35)
    System.out.println("Failed");
else
    System.out.println("No result");
```

Though it severs the purpose, there is a better way for implementing the above logic, which we see in the chapter 9.

Nesting the conditions

Here is another possibility in making decisions. An inner condition is chosen based on the result of the outer condition. Means, we are nesting the condtions.

```
if (n==25) {     // outer if condition
    if (m == 50)      // true block of outer if with nested condition
        System.out.println("M is 50"); // true block of nested if
    else
        System.out.println("M is not 50"); // false block of nested if
}else{
    if (p == 10)  // false block of outer if with nested condition
    System.out.println("P is 10");
    else
        System.out.println("P is not 10");
}
```

In the above code, the `if` statements that are checking the values of variables `m` and `p` are nested within the `if` statement that is checking the value of `n`.

Applying the Concepts

Objective

To develop a program to find if a number is even or odd.

Listing: IfDemo.java

```
1. class IfDemo {
2.     public static void main(String[] args) {
3.         int number = 25;
4.         if (number % 2 == 0)
5.             System.out.println(number + " is an even number");
6.         else
7.             System.out.println(number + " is an odd number");
8.     }
9. }
```

Explanation

Code is fairly straight forward. Line 3 is declaring a variable `number` and initializing it to `25`. Line 4 is checking if the reminder after dividing the `number` with `2` is zero. If it is zero, it is declared as even; odd otherwise.

Observe that both the *true* and *false blocks* are not having any delimiters as they are having only one statement per block.

Results

The result is equally straight forward.

```
25 is an odd number.
```

Try running this program by changing the value of number to something else and check if the results are correct.

Beyond the Concepts

Interesting Questions

What is the outcome of the following code? (`a = b` is written instead of `a == b`)

```
If (a = b)
    System.out.println("a and b are same");
else
    System.out.println("a and be are not same");
```

It raises a compile time error unless `a` and `b` are declared as `boolean`. In Java, the condition

expression must always be resolved into `boolean`.

Can the conditional statement check if a values less than some other value? Or it is always only == symbol?

Any expression that is resolved into a `boolean` is just fine.
So, the statements like `if (a < 20)` are perfectly fine.

Quiz

1. What is wrong with the following code?

```
If (a == b) {
    int c = 30;
}else{
    c++;
}
```

2. What is the outcome of the following program?

```
If (a == b)
    System.out.println("A and B are same");
    System.out.println("Got the positive result");
else
    System.out.println("A and B are not same");
```

Coding Exercises

1. Write a program to check if a given number is a multiple of another given number.
2. Write a program to print if the given year is a leap year or not.

Chapter 6
Working with Strings

- Handling Strings
- Analyzing Strings
- Index

Understanding the concepts

Strings are not just arrays of characters in Java. What I mean is that just by creating an array of characters does not make a string. To work with strings, we will have to create a `String` object or a `String` literal. First of the following snippets creates a `String` object while the later creates a `String` literal.

```
String s1 = new String("some text");
String s2 = "some other text";
```

We find out the exact difference between these two representations once we really crash into object oriented programming with Java. For now, we just focus on how to handle a string irrespective of the ways in which it is created.

Handling Strings

The following are the some of the important methods that are available for handling strings. Remember, none of these methods change the content of the string.

`trim()`	To get a trimmed copy of string by removing leading and trailing spaces
`toLowerCase()`	To get a copy of string with all characters in lower case
`toUpperCase()`	To get a copy of string with all characters in upper case
`replace()`	To get a copy of string by replacing certain characters

Analyzing a String

Following are some of the methods to inspect and analyze a string.

`charAt()`	To find the character at a specific position in the string
`indexOf(char c)`	To find the position of first occurrence of a character
`lastIndexOf(char c)`	To find the position of last occurrence of a character
`index of (String s)`	To find the position of first occurrence of a substring
`lastIndexOf(String s)`	To find the position of last occurrence of a substring
`length()`	To find the number of characters in a string
`startsWith(String s)`	To check if a string starts with a substring
`endsWith(String s)`	To check if a string ends with a substring
`equals(String s)`	To check if a string equals to another string in content
`equalsIgnoreCase(String s)`	To check if a string equals to another string in content irrespective of case

Index

Remember while working with the strings that the index of a string starts from 0. Means, the index of last character in string of 10 characters is 9 (not 10).

Applying the Concepts

Objective

To operate on strings.

Listing: StringDemo.java

```
1. public class StringDemo {
2.
3.     public static void main(String[] args) {
4.         String name = new String(" Glarimy Technology Services ");
5.
6.         String city = " Bangalore ";
7.
8.         System.out.println("City before trimming: " + city);
9.         city = city.trim();
10.        System.out.println("City after trimming: " + city);
11.        city = city.toLowerCase();
12.        System.out.println("City in lower case: " + city);
13.        city = city.toUpperCase();
14.        System.out.println("City in upper case: " + city);
15.        city = city.replace('B', 'M');
16.        System.out.println("City after replacing: " + city);
17.        System.out.println("Character at position 3 is " +
   city.charAt(2));
18.        System.out.println("Last index of A in city: " +
   city.lastIndexOf('A'));
```

```
19.       System.out.println("First index of A in city: " +
   city.indexOf('A'));
20.       System.out.println("Last index of AN in city: " +
   city.lastIndexOf("AN"));
21.       System.out.println("First index of AN in city: " +
   city.indexOf("AN"));
22.       System.out.println("");
23.
24.       System.out.println("Length of name: " + name.length());
25.       if (name.startsWith("Glarimy"))
26.          System.out.println("Name is starting with 'Glarimy'");
27.       else
28.          System.out.println("Name is not starting with 'Glarimy'");
29.
30.       if (name.endsWith("Services"))
31.          System.out.println("Name is ending with 'Services'");
32.       else
33.          System.out.println("Name is ending with 'Services'");
34.
35.       if (city.equals("BANGALORE"))
36.          System.out.println("City is BANGALORE, exactly");
37.       else
38.          System.out.println("City is not BANGALORE, exactly");
39.
40.       if (city.equalsIgnoreCase("BANGALORE"))
41.          System.out.println("City is BANGALORE, ignoring the case.");
42.       else
43.          System.out.println("City is not BANGALORE, ignoring the
   case.");
44.    }
45.
46.}
```

Explanation

At lines 4 and 6, two strings are created. Between lines 8 to 16, several operations are done, which are self-explanatory. The strings are analyzed in the remaining code.

Results

Running the above code results in the following output.

```
City before trimming:  Bangalore
City after trimming: Bangalore
City in lower case: bangalore
City in upper case: BANGALORE
City after replacing: MANGALORE
Character at position 3 is N
Last index of A in city: 4
First index of A in city: 1
```

```
Last index of AN in city: 1
First index of AN in city: 1

Length of name: 29
Name is not starting with 'Glarimy'
Name is ending with 'Services'
```

Beyond the Concepts

Interesting questions

In C, a string is represented as array of characters followed by the special terminating character. How is it done in Java internally?

Frankly, as Java programmers we better not to worry and depend on the internal representation of a string. Going by object oriented principles, the structure of the string is hidden from the user of the string. We just rely on the methods that are exposed to work with the strings.

If I declare an array of characters, how its different from a String?

An array of characters is just like any other array only. None of the methods that we have seen in the case of `String` are applicable for such an array (except the method `length()`).

Cant we equate two strings using simple == operator?

We can use the `==` operator but its meaning is different. If you are checking if the content of two strings is same, using the `equals()` method is the only way. The `==` operator checks if two variables point to the same reference irrespective of their contents. Look at the following snippet.

```
String s1 = "one";
String s2 = "one";
```

Here `s1` and `s2` are two different strings but their contents are same. First of the following returns `true` while the other returns `false`.

```
s1.equals(s2);
s1 == s2;
```

Quiz

1. When equating two strings, are the leading or trailing spaces considered?
2. Can we equate strings those having just spaces.
3. Can we equate 'A' with “A”? (note the first is a character and second is a string.)

Coding Exercises

1. Write a program to print a string by changing the case of each of its characters.
2. Write a program to print a string by reversing its content.

Chapter 7
Interacting with the user

- Reading normal text using Console
- Reading secret text using Console

Understanding the Concepts

So far, we have been declaring the variables and initializing them in the code itself. Running these programs several times results in the same output as neither the input nor the program is changing. Everything is *hard coded* in the program itself.

Now, we change that. We keep the logic same but we change the way the input is provided. Lets go interactive. We see how to prompt a user to input the data. We see how to read and interpret that input so we use it for the actual computation.

There are several ways to collect input from the user, interactively. We see one of them by using a Java object called `Console`.

Reading normal text using Console

Using `Console`, normal text with printable characters and numbers can be read. Using the the method `readLine()` of `Console`, we can prompt the user as well as collect the input from the user. Look at the following snippet.

```
// Get hold of Console from System
Console c = System.console();

//Prompt the user
String in =  c.readLine("Enter input: ");
```

This snippet is prompting the user with the hint *`Enter input:`* and waits. Only when the user

enters the input and presses the *Enter* key, it collects all the input in to a `String` called `in`.

Remember, the text is always a `String`. Even if the user enters *125* which is a number, it is still treated as a `String`. So, if the program is expecting an integer from the user, then it has to parse the input string for the integer using the wrapper class, as follows:

```
Console c = System.console();  // Get hold of Console from System
String in =  c.readLine("Enter input: "); //Prompt & collect the input
int inputNumber = Integer.parseInt(in); // Get the integer
```

Reading secret text using Console

In case, the input entered by the user must not be seen on the screen (such as passwords), the method `readPassword()` of the `Console` comes handy. Using this also, we can provide a prompt to the user and collect the secret input.

However, unlike in the case of normal text, this time we collect the input into an array of `char` (not into a `String`). Think, why?.

Applying the Concepts

Objective

To collect the user name and password followed by two numbers from the user and print the sum of the numbers.

Listing: ConsoleDemo.java

```
1. import java.io.Console;
2.
3. public class ConsoleDemo {
4.
5.     public static void main(String[] args) {
6.
7.         Console console = System.console();
8.         String uname = console.readLine("Enter user name: ");
9.         char[] pwd = console.readPassword("Enter password: ");
10.        int first = Integer.parseInt(console
       .readLine("Enter the first integer: "));
11.        int second = Integer.parseInt(console
       .readLine("Enter the second integer: "));
12.        int sum = first + second;
13.        System.out.println(first + " + " + second + " = " + sum);
14.        System.out.println("Thanks!");
15.    }
16.}
```

Explanation

At line number 1, we are *importing* `Console`. For now, assume that its somewhat like *including* a header file in C program using `#include` (but they are not same). We need to import `Console` as it is not automatically available for the Java programs.

Lines 7 and 8 are prompting the user for *name* and *password* and collecting them. We are not really using this input, its just for demonstration.

Lines 10 and 11 are prompting for numbers, collecting them after due parsing.

The remaining program is doing some arithmetic and printing the results. Nothing new here, we have seen it already.

Results

Running the above program gives the following output. The input I have provided while running this program is shown in **bold**. You may want to give different input to get different output in each run of the program.

```
Enter user name: krishna
Enter password:
Enter the first integer: 12
Enter the second integer: 45
12 + 45 = 57
Thanks!
```

Beyond the Concepts

Interesting Questions

What are the other alternatives for reading input interactively?

Besides using the `Console`, there is another object called `Scanner`. It offers more general purpose methods than `Console`. You can directly read various data types from the console without needing to parse the Strings. However, there is no way to hide the input entered by the user (like what we have done while reading the password).

There is yet another low level way for reading data from the console. You may want see this approach only after understanding *Java I/O* operations in detail.

What happens if the program expects an integer from the user and the user enter a value which is not an integer?

The program just reads the value as it is. However, when you try to parse it for the integer from it fails. At that point of time, the program exits by throwing information regarding what went wrong. These are the problems we call runtime errors.

Providing a prompt is a must?

Not a must but useful hint to the user. Otherwise, user may not know what to enter.

Quiz

1. When using the `Scanner`, how the `import` statement should look like?
2. Which version of JDK offers the `Console` and `Scanner`?
3. What are the methods available in `Scanner`?

Coding Exercises

1. Rewrite the above program by using `Scanner` instead of `Console`.
2. Write a program to read 5 decimal values from console and print the average of them.
3. Write a program to read the password and check if it is “glarimy”?

Chapter 8
Formatting the console UI

- Formatting using Console
- Support for data types
- Specifying the positions & Formatting instructions

Understanding the Concepts

We have seen how to write a simple program, how to do few basic computations, how to read data from the user and how to print the output. This time we climb one more step. Lets see how to print the output in a more presentable manner.

Formatting using Console

The `Console` object that we have used in the previous topic again comes handy while formatting the output. If you still remember to the `printf()` function of C programming, then Java formatting should not look too different. In fact, the name of the method in Java also is `printf()` !

The method `printf()` of `Console` takes a `String` as the first argument followed by any other arguments. The `String` value looks something like the following:

```
Hello %1$10s, welcome to %1$15s
```

As you can see, this string is combination of static text (like *Hello*) along with the placeholders for variable values along with formatting instructions (like *%$10s*). The rest of the arguments provide the actual data to be printed either as literals or variables. The `printf()` function uses this data to fill the placeholder locations in order for preparing the final output text.

Lets look at a more complete snippet:

```
String name = "Krishna";
```

```
String venue = "Bangalore";
printf("Hello %s, welcome to %s", name, venue);
```

The `%s` is formatting instruction to indicate that its the place for a `String` value. The variables `name` and `venue` are providing the content to these placeholders.

Also, look at the order of variables in the above snippet. Value of the first variable (`name`) goes for first place holder, value of the second variable goes to the second place holder and so on. The final printed output would be:

```
Hello Krishna, welcome to Bangalore
```

Support for data types

We can provide the formatting instructions for various other data types, not just for `String` types. Here is the list:

`%d` – integer group
`%f` – float group
`%c` – character group
`%b` – boolean group
`%s` – strings

Specifying the positions

The above `printf()` example picks up the values for substitution based on the position. Thus, first variable value goes to first %s and second variable value goes to second %s. It has a shortcoming.

If you want to print the output like the following:

```
Welcome to Bangalore. Bangalore is cool!
```

Then the printf statement should look something like this:

```
printf("Welcome to %s. %s is cool!", venue, venue);
```

Observe that the variable `venue` is repeated twice. Instead of relying up this scheme, we can actually provide the index of the variable arguments for reuse. See the following:

```
String name = "Krishna";
String venue = "Bangalore";
printf("Hello %1s, welcome to %2s. %2s is cool", name, venue);
```

You guess the output. %1s suggests to pick the first variable argument parameter whereas %2s suggests to pick the second value parameter.

But where is the real formatting

The above example only shows how to prepare an output with place holders for various data types and how to really pass the values for the place holders. Now let us see how to format them using

the following example:

```
String name = “Krishna”;
String venue = “Bangalore”;
printf(“Hello %1$10s, welcome to %1$15s”, name, venue);
```

This snippet prints the output with a different format. It allocates space for 10 characters to print the `name` and 15 characters to print the `venue`. If the values are not having those many characters, it pads the string using spaces. If the values are having more than those many characters, it just prints the value ignoring the formating instruction.

Applying the Concepts

Objective

Print values of various data types. The format is to have a table with two columns. First column of size 11 characters and the second column is of 16 characters.

Listing: FormatDemo.java

```
1. class FormatDemo {
2.     public static void main(String[] args) {
3.         byte a = 1;
4.         short b = 200;
5.         int c = 250000;
6.         long d = 3456712345673L;
7.         float e = 3.5F;
8.         double f = 40.56;
9.         char g = 'a';
10.        boolean h = false;
11.        String i = "Glarimy";
12.
13.        java.io.Console console = System.console();
14.        console.printf("-----------|----------------\n");
15.        console.printf("%1$10s | %2$15s\n", "Variable", "Value");
16.        console.printf("-----------|----------------\n");
17.        console.printf("%1$10c | %2$15d\n", 'a', a);
18.        console.printf("%1$10c | %2$15d\n", 'b', b);
19.        console.printf("%1$10c | %2$15d\n", 'c', c);
20.        console.printf("%1$10c | %2$15d\n", 'd', d);
21.        console.printf("%1$10c | %2$15.2f\n", 'e', e);
22.        console.printf("%1$10c | %2$15.2f\n", 'f', f);
23.        console.printf("%1$10c | %2$15c\n", 'g', g);
24.        console.printf("%1$10c | %2$15b\n", 'h', h);
25.        console.printf("%1$10c | %2$15s\n", 'i', i);
26.        console.printf("-----------|----------------\n");
27.    }
28.}
29.
```

Explanation

Lines from 3 to 11 are declaring and initializing variables of various data types. The remaining code is printing the same data in a tabulated form.

Observe that we are not writing the `import` statement to use the `Console` object at line 13. This is a shortcut in case you are not planning to declare many variables of type `Console` in the same file. Of course, this short cut is nothing to do with the formatting.

Results

Running the above code prints the following output.

```
-----------|----------------
Variable   |            Value
-----------|----------------
       a   |                1
       b   |              200
       c   |           250000
       d   |   3456712345673
       e   |             3.50
       f   |            40.56
       g   |                a
       h   |            false
       i   |          Glarimy
-----------|----------------
```

Beyond the Concepts

Interesting Questions

Is there way to format the prompts to the user?

Why not? In fact, whatever you want to print (be it a prompt or an output), you can format it. The methods `readLine()` and `readPassword()` of the `Console` accept not just one parameter. They can accept any number of parameters, first being `String`. If the string contains any formatting instructions, the actual values can be passed as the rest of the arguments.

What happens if I use `%f` but provide an `int` value? What happens vice-versa?

The program exits after printing that *format conversion fails*. We must be careful in matching the data types in the format and the data types of the real values we provide.

Quiz

1. How to align the data? For example, how to say that the data must be left-aligned?
2. What happens if the number of parameters is not matching the indexes of the format placeholders?

Coding Exercises

1. Print the multiplication table of a number in a nice table. Prompt for the number from the user.
2. Accept a string from the user and print its words in a column with a nice format.

Chapter 9
Choosing from many alternatives

- switch statement
- break statement
- default statement

Understanding the Concepts

Conditional execution of a path using the `if` statement is useful, but may not be elegant in some situations. For example, in case the condition results in more than one possibilities, `if … else … if … else` statements may not look good. In such a situation, having `switch` statement really helps in writing clean code.

`switch` statement

`switch` statement in Java works just like how it works in the C language. Take a look at the template.

```
switch (condition) {
    case option 1:
        {block 1}
    case option 2:
        {block 2}
    case option 3:
        {block 3}
    .
    .
    default:
        {default block}
}
```

Every `switch` statement takes a condition like what an `if` statement takes. However, unlike in the case of `if` statement, `switch` condition must always be resulted in `int` value. Each of the values becomes a `case` for the `switch` statement. And each `case` decides the *entry point* into the `switch`

`break` statement

Once the execution enters into a `case` block based on the condition value, it executes the specific block and all other blocks of the `switch` that follows it, till it reaches a `break` statement or end of the `switch` statement. Having a `break` at the end of each of the cases is not mandatory. You decide where where you want to exit the `switch` statement.

`default` case

If the condition does not resolve in any of the cases of the `switch` statement, a block named `default`, if present, would be executed. If the `default` case is not present, execution simply exits the `switch` statement.

Now, lets look at an example:

```
switch (mod) {
case 0:
    System.out.println("The number is an even as mod is zero");
    break;

case 1:
    System.out.println("The number is odd as mod is 1");
    break;

default:
    System.out.println("error");
}
```

The above code taking the value of an integer variable `mod`.

If the value of `mod` is 0, it enters the `case 0` and would have run till the end of the `switch` statement. However, because there is a `break` at the end of `case 0`, it breaks away from the `switch` immediately after executing the `case 0`.

If the value of `mod` is 1, it enters the `case 1` and would have run till end of the `switch` statement. Again, because of presence `break` statement, it just exits.

If the value of `mod` is neither 0 nor 1, there are no specific cases present for those other values. Hence, it executes `default` block.

By now, you might understand that `switch` is essentially like an `if` statement only but with more than two possible paths of executions. `switch` gives neat organization of code in such circumstances.

Applying the Concepts

Objective

Write a program to either add or multiply two numbers based on user selection.

Listing: SwitchDemo.java

```
1. public class SwitchDemo {
2.
3.     public static void main(String[] args) {
4.         java.io.Console console = System.console();
5.         int option = Integer.parseInt(console
6.                 .readLine("Choose: 1) Add 2) Multiply 3) Exit: "));
7.         int first, second;
8.         switch (option) {
9.         case 1:
10.            first = Integer.parseInt(console.readLine("First: "));
11.            second = Integer.parseInt(console.readLine("Second: "));
12.            int sum = first + second;
13.            console.printf("Sum: %1$5d", sum);
14.            break;
15.        case 2:
16.            first = Integer.parseInt(console.readLine("First: "));
17.            second = Integer.parseInt(console.readLine("Second: "));
18.            difference = first - second;
19.            console.printf("Difference: %1$5d", difference);
20.        case 3:
21.            console.printf("Thanks!");
22.            System.exit(0);
23.        default:
24.            console.printf("Invalid option. ");
25.        }
26.    }
27. }
```

Explanation

This programs starts with prompting the user with a menu through the `Console`. User is free to select any one from the three options. Based on the selected option, a specific block of code gets executed. Each `case` is separated by the `break` statement and hence only one block gets executed under any circumstances.

Line 22nd demands additional line of explanation. The call `System.exit(0)` makes the system to shutdown. We do this when a program can not proceed further with any useful computation (we call this graceful shutdown). You can achieve the same result by simply placing a `break` statement. Its just that we used a different call.

Result

Running the above code may give the following results.

The test data entered is shown in **bold**.

```
Choose: 1) Add 2) Multiply 3) Exit: 1
First Number: 34
Second Number: 78
Sum:  112
```

Beyond the Concepts

Interesting Questions

Is the `break` statement part of `Switch` statement, always?

No. The `break` statement is used to exit from the `switch` statement. In case, you do not want to exit from the switch in the middle, you may not use the `break` statement.

Is the `default` case part of `Switch` statement, always?

No. The `default` case is used to catch all the cases for which there is no specific handling is required.

What happens if I put the `default` as the first case?

Nothing strange happens. Its also just like any other `case` only.

Quiz

1. Can we put a `switch` inside an `if` block? Can we put `if` statement in a `switch` case? Can a `switch` case again have another `switch`?

Coding Exercises

1. Enhance the previous listing to add the cases for multiplication and division as well.
2. Rewrite the listing used in the previous topic to print the capital of a specified country using the `switch` statement.

Chapter 10
Iterations

- `while` loop
- `do...while` loop
- `for` loop
- `break` and `continue` statements

Understanding the Concepts

We know that computer is good at doing repeated jobs in a loop at a faster pace than a human being can do. Yet, we have not seen any program having loops, so far in this book. This is the time to look at writing such programs.

Three Kinds of Loops

Java supports three kinds of loops like most of the languages, namely `while`, `do...while` and `for` loops. Though we learn all three here, most likely you would use `for` loop more often than the other loops.

`while` loop

If you want a set of statements to be executed for zero to any number of times, then the `while` loop is meant for that job. Take a look at the following snippet that is printing the 10 multiples of 2.

```
int a = 1, b;
while(a <= 10) {
    b = a * 2;
    System.out.println(b);
    a++;
}
```

Here, the `while` loop is first checking the value of variable `a`. As long as it is less than or equal to 10, it executes the block that immediately follows the condition. That block is computing and printing the next multiple of 2 and incrementing the the value of `a`. Thus, after 10 iterations, the `while` condition breaks and the program comes out of the loop.

The character of the `while` loop is that it's block gets executed only when the condition is satisfied. Means, there is no guaranty that the loop gets executed at least once.

do...while loop

Slightly different loop is `do...while`. This loop also contains the block of code that is repeatedly executed and a condition to proceed further. However, this loop guaranties that the block gets executed at least once. The above code is written using `do..while` loop that produces the same result. Take a look.

```
int a = 1, b;
do {
   b = a * 2;
   System.out.println(b);
   a++;
}while(a <= 10);
```

`for` loop

In both of the above examples, I would like you to make three observations. We have few variables being read or written. We have some computation in the block. And we have an operation to increment a variable so that at some point of time the loop is broken.

The `for` loop also have all these three, but in a slightly different arrangement, like the following.

```
for (int a = 1, b; a <= 10; a++) {
   b = a*2;
   System.out.println(b);
}
```

Result is the same.

The structure of the for loop is like this: It has an *initializer*, a *condition*, a *counter* and a *procedural block*. The template look as follows:

```
for(initializer; condition; counter) {
   procedural block;
}
```

The variables that are going to be used locally in the loop are declared in the *initializer*. These variables may optionally be initialized as well. In fact, the initializer part itself is optional. This part is executed only once, if present.

Then comes the *condition* part. This is executed immediately after the *initializer* part. If the *condition* is resolved to `true`, then the *procedural block* gets executed. If the *condition* is resolved

to `false`, then the execution comes out of the loop.

The *counter* part is executed at the end of executing the *procedural block*. This is followed by checking the *condition* once again. And it goes on like this.

Point worth noting is that none of these parts are mandatory. You may write a `for` loop like the following:

```
for(; ;);
```

Guess what happens, with this kind of code?

The other loops

Java also supports a different kind of loop popularly called as `for … each` loop. We look into this once we see the array handling in Java.

The `break` and `continue` statement

We have already seen the `break` statement as part of `switch`. It forces the path of execution out of `switch` statement. The fact is that the `break` statement always forces the path of execution out of the current block, no matter to which block it belongs to. If it happens in the `for` loop or while loop, then the program comes out of the loop, what else?

The `continue` statement is little polite. It just skips the rest of the code in the current block of the loop. That means, the `continue` statement in a `for` loop forces the execution to skip the rest of *procedural block* and goes to the *counter* part.

Together with the decision making statements (`if` and `switch`), the loops form vital tools for the programmer in logic development.

Applying the Concepts

Objective

To print the multiplication table using different loops.

Listing: LoopsDemo.java

```
1. import java.io.Console;
2.
3. public class LoopsDemo {
4.     public static void main(String[] args) {
5.         Console console = System.console();
6.         int n = Integer.parseInt(console.readLine("Number: "));
7.         int terms = Integer.parseInt(console
8.                 .readLine("Enter number of terms: "));
9.
```

```
10.       console.printf("Table using FOR loop\n");
11.       for (int c = 1, p = 0; c <= terms; c++) {
12.          p = c * n;
13.          console.printf("%1$5d x %2$5d = %3$5d \n", n, c, p);
14.       }
15.
16.       int c;
17.       int p;
18.
19.       console.printf("Table using WHILE loop\n");
20.       c = 1;
21.       while (c <= terms) {
22.          p = c * n;
23.          console.printf("%1$5d x %2$5d = %3$5d \n", n, c, p);
24.          c++;
25.       }
26.
27.       console.printf("Table using DO...WHILE loop\n");
28.       c = 1;
29.       do {
30.          p = c * n;
31.          console.printf("%1$5d x %2$5d = %3$5d \n", n, c, p);
32.          c++;
33.       } while (c <= terms);
34.    }
35.}
```

Explanation

This program accepts a number and prints its table for up to the specified terms. Lines from 11 to 14 are printing the table using the `for` loop. Same is being done from lines 21 to 25 using the `while` loop. Lines from 29 to 33 are doing the same using `do … while` loop. Code is self explanatory.

Results

I entered 125 as the number and 5 as the number of terms and got the following results.

```
Number: 125
Enter number of terms: 5
Table using FOR loop
  125 x     1 =   125
  125 x     2 =   250
  125 x     3 =   375
  125 x     4 =   500
  125 x     5 =   625
Table using WHILE loop
  125 x     1 =   125
  125 x     2 =   250
  125 x     3 =   375
```

```
  125 x     4 =   500
  125 x     5 =   625
Table using DO...WHILE loop
  125 x     1 =   125
  125 x     2 =   250
  125 x     3 =   375
  125 x     4 =   500
  125 x     5 =   625
```

Beyond the Concepts

Interesting Questions

I cannot wait any more. If loops can be implemented using any of these three (`for`, `while` and `do … while`), which is the best choice?

Surely, its the `for` loop.

The reason is simple. Recollect our discussion about scope of the variables. If a variable is declared in a block, it is visible only in that block or its inner blocks.

In the case of `while` and `do...while` loops, we are declaring and initializing the variables out side of the loops. Because of this, they belong to not only the `while` and `do...while` loops, but any code that follows these loops. Accidentally if we are using these variables out side of the loops where it is not intended for, we may end up in problems.

In the case of `for` loop, we have a way to declare and initialize them in the `for` statement itself. So, once the loop is over, the variables are no more available. So, no accidents.

Should the counter *part of the `for` loop always only increment an integer?*

Nothing like that. Though we call it as *counter* part, you can write any code over there as long as it leads to breaking of the loop at some point of time.

How to implement an infinite loop?

Use `while(true) { …. }` or `do { …. } while (true)` or `for(;;) { … }`.

Can one loop have some other loop?

Why not? The *procedural blocks* (for that matter, any part) of the loops can contain any valid Java code.

Quiz

1. What is the result of the following code?

```
int a = 10;
```

```
boolean b = true;

while (a < 10) {
    if (b)
        break;
    else
        a++;
}
System.out.println(a);
```

2. What is the result of the following code?

```
int a = 10;
boolean b = true;

while (a < 10) {
    if (b)
        continue;
    else
        a++;
}
System.out.println(a);
```

Coding Exercises

1. Develop an application which prompts the following menu to the user:

   ```
   Choose 1) Add 2)Multiply 3)Exit:
   ```

   Once the user selects either 1 or 2, the application should prompt for two numbers and print the sum or product of them appropriately. And, it should prompt the menu again.

   Only if the user selects the option 3, the application should be closed.

Chapter 11
Doing Logical Lperations

- AND operation
- OR operation
- NOT operation
- Short circuiting

Understanding the Concepts

We have developed logic using arithmetic operations like addition and multiplication. Now, we would see few other operations largely fall under the category *logical operations*. We use these operations in combining more than one condition in an `if` statement or in loop statements.

While arithmetic operations results in numbers, logical operations results in boolean values `true` or `false`.

`AND` operation

Look at the following snippet. The `if` statement checks if the value of variable `a` is less than 10 and also is greater than 5.

```
if (a > 5 & a < 10)
    System.out.println("A is between 5 and 10");
```

Observe that there are two conditions in the `if` statement: one to check if `a < 10` and the other is to check if `a > 5`.

These two conditions are connected using the `AND` operator with the symbol `&`. This operator

resolves both the conditions and returns `true` only in case both of them are `true`. It returns `false` in all other cases. This is depicted in the following *truth table*.

Value of a	Is a < 5?	Is a < 10?	Result of AND operation
6	TRUE	TRUE	TRUE
4	FALSE	TRUE	FALSE
13	TRUE	FALSE	FALSE

`OR` operation

Having understood the `AND` operation, dealing with the `OR` operation is easy. The `OR` operator in Java is symbolized using | (pronounced as *pipe*) symbol. The `OR` operator returns `true` when either of its operand conditions proves to be `true`. Look at the truth table below.

Value of a	Is a < 5?	Is a < 10?	Result of AND operation
6	TRUE	TRUE	TRUE
4	FALSE	TRUE	TRUE
13	TRUE	FALSE	TRUE

So, the following statement simply checks if the value of `a` is more than 5 or less than 10 or both.

```
if (a > 5 | a < 10)
   System.out.println("A is either greater than 5 or less than 10 or both");
```

`NOT` operation

The last logical operation is `NOT` operation. It works against just one condition by negating the result of the condition. For example, if the following condition returns `true`

```
   if ( a > 5 )  // assume it returns true
```

then by using the `NOT` operator as shown below, it becomes `false`.

```
   If (!(a > 5) // it becomes false
```

As seen, `!` is the symbol for `NOT` operation.

Short Circuiting

Lets look at the following code. Note that we are using post-increment operations. Now, guess the result!

```
int a = 1, b = 1;

if (a++ > 1 & b++ > 1) {
   ... // true block
}
System.out.println(b);
```

The first condition is checking if the value of `a` is greater than 1 (and finds it is not). After the check, it is incrementing the value of `a` to 2.

The second condition is checking if the value of `b` is greater than 1 (and finds it is not) followed by incrementing the value of b 2.

As both the conditions are returning `false`, the `AND` operator returns `false`. Because there is no *false block* for this `if`, execution just moves to the next line of code where it is printing the value of `b` which is 2.

In general, the `AND` operator returns `false` if at least one of its operands return `false`. In our example, the first condition itself is returning `false`. Now, lets pause and ask ourselves a question. Why not the `AND` operator returns immediately with the `false` as it is going to do it any way after checking the other conditions as well. Interesting, isn't it? It saves some unnecessary computation, if it returns at the earliest. However, the regular `AND` operator still go with resolving all the conditions before returning the final result.

Interestingly, there is another `AND` operator which is called as *short-circuit `AND` operator*. It does return immediately once it is sure about the final result.

The following code is using such as short circuit (symbolized by `&&` symbol).

```
int a = 1, b = 1;

if (a++ > 1 && b++ > 1) {
    ... // true block
}
System.out.println(b);
```

Take Notice

Because it is returning immediately after the first condition returns `false`, the second condition is never going to be executed and hence the post-increment of `b` would never happen. Final result? The value of `b` continues to be 1 only (not 2).

Same case with the `OR` operator. If *short-circuit `AND`* returns once it finds a condition resolved to `false`, the *short-circuit `OR`* returns once it finds a condition resolved to `true`. The symbol for *short-circuit `OR`* is `||`.

You may want to use regular or faster short-circuit operators as long as you keep an eye on these kinds of side-effects.

Applying the Concepts

Objective

To demonstrate the effects of various logical operations.

Listing: LogicalDemo.java

```
1.  public class LogicalDemo {
2.      public static void main(String[] args) {
3.          int limit = 1;
4.          int terms = 5;
5.
6.          int number = 2;
7.          int multiplier = 1;
8.          int product = 0;
9.
10.         System.out.println("Using AND operator");
11.         while (product < limit & multiplier++ < terms) {
12.             System.out.println(product);
13.             product = number * multiplier;
14.         }
15.
16.         System.out.println("Using SHORTCUT AND operator");
17.         while (product < limit && multiplier++ < terms) {
18.             System.out.println(product);
19.             product = number * multiplier;
20.         }
21.
22.         System.out.println("Using OR operator");
23.         while (product < limit | multiplier++ < terms) {
24.             System.out.println(product);
25.             product = number * multiplier;
26.         }
27.
28.         System.out.println("Using SHORTCUT OR operator");
29.         while (product < limit | multiplier++ < terms) {
30.             System.out.println(product);
31.             product = number * multiplier;
32.         }
33.     }
34. }
```

Explanation

The code is self-explanatory.

Results

```
Using AND operator
0
Using SHORTCUT AND operator
Using OR operator
4
8
Using SHORTCUT OR operator
```

Beyond the Concepts

Interesting Questions

Between the arithmetic and logical operations, which gets precedence?

It is always the arithmetic operation that get precedence over the arithmetic operation.

Quiz

1. What is the result of the following code?

```
while( true & false )
   System.out.println("Glarimy");
```

Coding Exercises

1. Read the year and month from the user and print the number of days of the month (check if the year is leap year or not and etc.,).

2. Read a number and print if its a prime number or not.

Chapter 12
Working with Strings, Again

- Mutable and immutable strings
- String Builder
- Tokenizing a String

Understanding the Concepts

Well, we have seen how to create and analyze a string. Having also seen how to make decisions and iterations, we would examine the strings in little more interesting ways.

Mutable and Immutable Strings

We have already seen this. By using the `+` operator we can concatenate strings. Have a look at the following code for a quick recap.

```
String s1 = "Hello";
String s2 = "World!";
String s1 = s1 + " " + s2;
System.out.println(s1);
```

This would print the following:

```
Hello World!
```

Now, lets see how to concatenate string by other means. But why again we look at other means of string concatenation? Here is the reason.

Though we see only two strings (`s1` and `s2`) in the above code, behind the scenes there are two more strings are getting created. This is because, once a string is created, it's content can never be changed. For this reason, we call `String` objects as *immutable objects*. If you want to modify its

contents, create a new `String` and put the new content in it. Means, the operation `s1 + " "` is silently creating a new `String` with `"Hello "` as the content. Then this new string is concatenated with *s2* to produce yet another new string object with the content `"Hello World!"`. Finally this new string is then renamed as `s1` when we assigned it to the variable `s1`. Means, for two `+` operations, we got two more `String` objects created.

Note that creating every new object costs some memory and more CPU operations. Think if you are creating two objects in a loop for 100000 times. It would create 200000 more objects. Phew!

In a server environment which keeps on running for months together, the cost of these invisible creations is significant. It may some time leads to slower performance.

So, how to avoid this performance degradation while concatenating strings?

StringBuilder

This is where `StringBuilder` comes in handy. `StringBuilder` creates a *mutable string* and here is how you work with it.

```
StringBuilder s1 = new StringBuilder();
s1.append("Hello");
s1.append(" ");
s1.append("World!");
String s2 = new String(s1); //Creating the actual string
```

Use this method of string creation when you expect the string contents to be changed during the program execution.

Tokenizing a String

Let's look at a completely different scenario.

How many words the string `'Hello World'` is made up of? Two!

How do you say that?

Because we know how to identify words because we know that the words in a sentence are separated by spaces. To put in another words, the sentence is made up of several *tokens* separated by space as the *delimiter*.

Now, lets see if you can find the number of tokens in the following string, if comma is treated as the delimiter.

```
Ram, Bhim & Shyam went to market to buy books, pens & albums.
```

How many tokens you figured out? Three, isn't it? Here are they:

```
Ram
 Bhim and Shyam went to market to buy books
 pens and albums
```

Observe the leading space in the last two tokens.

Now, if the symbol `&` is treated as the delimiter, how many tokens you can find?

```
Ram, Bhim
 Shyam went to market to buy books, pens
 albums
```

Now, how about taking both comma and `&` as the delimiters? Then, the following are the tokens:

```
Ram
 Bhim
 Shyam went to market to buy books
 pens
 albums
```

Enough, having seen how to *tokenize* a string using a set of delimiters, lets see how to code for it using Java.

StringTokenizer

To tokenize a string in Java, you create a `StringTokenizer` object by providing it with the string and list of characters that you would want to use as delimiters.

For example, the following code creates a `StringTokenizer` that is capable of *tokenizing* a string `str` using comma as the separator.

```
StringTokenizer st = new StringTokenizer(str, ",");
```

Similarly, the following creates a `StringTokenizer` that takes both comma and semicolon as the delimiters.

```
StringTokenizer st = new StringTokenizer(str, ",;");
```

Reading the Tokens

Once the `StringTokenizer` is created, then you can use `nextToken()` method to get the next token as follows:

```
// moves to the first token and returns it
String firstToken = st.nextToken();

//moves to the next token and returns it
String secondToken = st.nextToken();

// moves to the next token and returns it
String thirdToken = st.nextToken();
```

If there are no more tokens, the `nextToken()` method returns `null`.

More tokens out there?

In case, you would want to know if there are more tokens available before really reading them, use the method `hasMoreTokens()`. This method returns `true` if there are more tokens to read, `false` otherwise.

Applying the Concepts

Objective

To read several lines of text from the console.

Listing: StringManipulationDemo.java

```
1. class StringManipulationDemo {
2.     public static void main(String[] args) {
3.         java.io.Console console = System.console();
4.         String line = null;
5.         StringBuilder buffer = new StringBuilder();
6.         console.printf("Enter multi-line text:”);
7.          console.print(“Enter period on a new line to stop\n");
8.         while (!(line = console.readLine()).equals("."))
9.            buffer.append(line);
10.        System.out.println(new String(buffer));
11.    }
12. }
```

Explanation

This program prompts the user to input multiple lines of text. User presses *Enter* key to complete a line. Also, inputs *full-stop* (period) symbol in a new line to denote the end of input.

This program keep on reading and adding the lines to the buffer which is a `StringBuilder`. Once the input is over, it creates a single string out of all the lines and prints it.

Result

Running this program with a sample data gives the following output.

```
Enter text. Enter period on a new line to stop
Jana gana mana
Adhinaayaka Jayahe
Bharatha Bhagya Vidhatha
.


Jana gana manaAdhinaayaka JayaheBharatha Bhagya Vidhatha
```

Objective

To print tokens of a string given the delimiters.

Listing: Tokenizer.java

```
1. import java.util.StringTokenizer;
2.
3. public class Tokenizer {
4.     public static void main(String[] args) {
5.         String line = "I am proud of my country, Mother India.";
6.         StringTokenizer st =
7.           new StringTokenizer(line, " ,.", false);
8.         while (st.hasMoreTokens())
9.            System.out.println(st.nextToken());
10.    }
11. }
```

Explanation

This is a straightforward program. It creates a `StringTokenizer` by passing a string and list of delimiters. Then in a `while` loop reads and prints all the tokens.

Results

Running the above program prints the following.

```
I
am
proud
of
my
country
Mother
India
```

Beyond the Concepts

Interesting Questions

Isn't it good to use `StringBuilder` always instead of `String`?

No. First, the overhead of using `String` is evident only when you are trying to modify its content frequently. If the content of the string is not going to change (or changes very infrequently), then why you want to create a `StringBuilder` and then create a `String` out of it?

Second point, like `StringBuilder` there is another class `StringBuffer` which is useful in *multi-threaded* environments. Using `StringBuilder` in such environments is not a good idea.

I want to treat the delimiters also as as tokens. Can I?

Yes. In such a case, create the `StringTokenizer` as follows:

```
StringTokenizer st = new StringTokenizer(str, ",", true);
```

Note the third parameter, `true`. This is what signals the tokenizer to count the *delimiters* also as *tokens*.

Quiz

1. Like appending, can I truncate the contents of a `StringBuilder`?
2. What happens when you are trying to tokenize a string using comma as delimiter and that string is having three subsequent commas?
3. Look at all the methods of `StringTokenizer`. What is the difference between the methods `nextToken` and `nextElement`?

Coding Exercises

1. Look at all the methods of `StringBuilder`.
2. Find the advantage of using that `split` method of `String`.
3. Write a program to print number of lines and number of words present in a paragraph of text.
4. Write a program to read the name of user. The name must be a combination of first name, middle name followed by last name. Your program must validate it if the name entered by the user is in the expected format or not.

Chapter 13
Working with arrays

- Declaring arrays
- Constructing arrays
- Initializing arrays
- Knowing and changing the size
- Accessing Elements using `for` loop

Understanding the concepts

We have been working with the values which are inherently scalar. If we declare a variable `x` of type `int`, it holds just one integer. Some times, we may want to treat several integers as a single group. Look at the following case.

You want to write a program to find the sum of 10 integers. One of the ways is to declare 10 integer variables to hold these values and then have another variable where you would store the computed sum of these values.

Now, think if you want to write the above logic for 1000 integers. Does it really make sense to create those many integer variables? Absolutely no, if we have an alternative. *Arrays* are such an alternative.

An *array* is a data structure that holds a collection of values that belong to the *same* type.

Means, an array can hold ten integers. Other array can hold thousand booleans. But not that an array can hold five integers and five booleans.

In order for using an array, first it needs to be declared. Then it needs to be constructed and finally it needs to be initialized. Only then its ready for actual computations.

Lets see how we do each of them.

Declaring arrays

```
int data[]; // declaration
```

The above line of code is declaring a variable `data` as an *array of integers*. The pair of square brackets immediately followed by the variable name is what makes it an array. You can also declare an array in the following way.

```
int[] data; // declaration
```

Both these lines result in the same thing: *an integer array called* `data`.

Constructing Arrays

An array needs to be *constructed* by specifying the *maximum capacity*. The following snippet constructs the `data` array with a capacity of 25;

```
int[] data; // declaration
data = new int[25]; //construction

// declaration and construction in just one line
int[] other = new int[25];
```

Initializing Arrays

Once we have specified the maximum capacity, then we are ready to actually keep the real values at specified positions in the array. These positions are called *indexes*.

The first index is numbered as 0. The next index is numbered as 1. And so on. That means, if the capacity of the array is N, then the last index position of the array is going to be N-1.

Look at the following code which is filling up the three positions of the array.

```
int[] data = new int[25];
data[0] = 134; // first position in the array holds the integer 134
data[1] = 786; // second position in the array holds the integer 786.
data[24] = 1024; // last position in the array holds the integer 1024.
```

Note that there is no need of keeping the values sequentially. You can keep the values at positions 1, 2, 13, 20, 21, 4, 19 … in any order, as long as the index is valid.

All in one

If you have a smaller array and have the values ready, then you can do all three steps in just one line. Here is an example.

```
String[] days = {"SUN", "MON", "TUE", "WED", "THU", "FRI", "SAT"};
```

Here, the construction is automatically done with the capacity to store seven strings before they get initialized with the comma separated values in the pair of braces.

Changing the size

One limitation of the arrays is that, once you specify the capacity of an array, you can not change it. Means, if you create an array with capacity of 1000, you can not increase it any more. You can not even reduce it also. If you have values for only 10 elements, you simply waste the other indexes of the array but you can not reduce the capacity, once declared.

Knowing the size

At any point of time, you can identify the size of an array by calling the *property* `length` (its a property, not a method) as follows:

```
int[] data = new int[25];
...
int l = data.length; // it must return 25, irrespective of number real elements
```

Accessing the elements using `for` loop

To access any element in the array, simply use the index position as follows.

```
System.out.println(data[3]); // prints the fourth element in the array
int sum = data[0] + data[1]; // adds the first two values in the array
```

In case, you would want to access all the elements in the array, then the improved `for` loop really helps you. Here is how you use this for loop.

```
for(int element : data) {
   // use the integer a for the computations
}
```

The above `for` loop takes data array and copies each of the elements in to the local variable `element` (one per iteration). The body of the `for` loop can use current value of the `element`.

Applying the Concepts

Objective

1. To print the elements of an array in their actual order, sorted order and inverted order.
2. To find the maxium value in the array.
3. And to check if two arrays are same.

Listing: ArrayDemo.java

```
1. public class ArrayDemo {
2.
3.     public static void main(String[] args) {
4.         // Declaration
```

```
5.        int data[];
6.
7.        // construction
8.        data = new int[6];
9.
10.       // initialization
11.       data[0] = 12;
12.       data[1] = 32;
13.       data[2] = 2;
14.       data[3] = 10;
15.       data[4] = 11;
16.
17.       // traversal
18.       System.out.print("Array Data: ");
19.       for (int number : data)
20.          System.out.print(number + " ");
21.
22.       // finding maximum
23.       int max = 0;
24.       for (int number : data)
25.          if (number > max)
26.             max = number;
27.       System.out.print("\nMaximum: " + max);
28.
29.       // reversing
30.       int temp;
31.       for (int index = 0; index < data.length / 2; index++) {
32.          temp = data[data.length - index - 1];
33.          data[data.length - index - 1] = data[index];
34.          data[index] = temp;
35.       }
36.       System.out.print("\nInverted Array: ");
37.       for (int number : data)
38.          System.out.print(number + " ");
39.
40.       // sorting
41.       for (int count = 0; count < data.length; count++)
42.          for (int i = 0; i < data.length - count - 1; i++)
43.             if (data[index] > data[i + 1]) {
44.                temp = data[i + 1];
45.                data[i + 1] = data[i];
46.                data[i] = temp;
47.             }
48.       System.out.print("\nSorted Array: ");
49.       for (int number : data)
50.          System.out.print(number + " ");
51.
52.       // Array constants
53.       int other[] = { 10, 12, 45, 10, 23 };
54.
55.       System.out.print("\nOther Array: ");
```

```
56.        for(int number: other)
57.           System.out.print(number + " ");
58.
59.        // Checking if two arrays are same
60.        if (data.length == 0 || data.length != other.length) {
61.           System.out.println("\nArrays are not the same");
62.        } else {
63.           boolean same = true;
64.           for (int i = 0; i < data.length; i++)
65.              if (data[i] != other[i]) {
66.                 same = false;
67.                 break;
68.              }
69.           if (same)
70.              System.out.println("\nArrays are same");
71.           else
72.              System.out.println("\nArrays are not same");
73.        }
74.     }
75. }
```

Explanation

The above program is working with a sample array which is declared, constructed and initialized by line 15.

Lines 18-20 are accessing all the values of the array using the improved `for` loop.

Lines 23-27 are also using the improved `for` loop to find the maximum value found in the array.

Lines 30-35 are reversing the array. Means, swapping the values between ith and N-1-ith positions where i starts from 0 and progresses till the middle of the array. This code is using the normal `for` loop as it will have to use the index position for computation.

Lines 41 to 50 are sorting the array. This sorting technique is known as *bubble sort*.

If `data` and `other` are two array variables, to check if they are same or not, we use == symbol. However, if we want to check if their contents are same we need to traverse both the arrays and compare each of the values. That is what happening in the rest of the program.

Results

Running this program yields the following results:

```
Array Data: 12 32 2 10 11 0
Maximum: 32
Inverted Array: 0 11 10 2 32 12
Sorted Array: 0 2 10 11 12 32
Other Array: 10 12 45 10 23
Arrays are not the same
```

Beyond the Concepts

Interesting Questions

If the capacity of an array is 10, what happens when we try to insert a value at index 10?

With the capacity of 10, the last index available in the array is only 9. Hence, trying to use any index beyond 10 would result in program error.

What happens if we are trying the access elements in an array which is not initialized?

Elements in an array are automatically initialized to their default values once the array is constructor. There is no question of garbage values in the array.

The improved for loop is meant to be used only with the arrays?

Not exactly. This form of loop is meant to be used with any collection that can be *iterated*. So far we have seen only arrays that fall under this category. There are other collections available in Java and this loop is useful there as well.

What is good choice between normal for loop and improved `for` loop?

In case you would want to iterate through all the element without worrying about the index position of the current element, use the *improved* `for` loop because of its simplicity. Otherwise, if you want to know the index position for computation, using normal `for` loop is better.

How about creating 2-dimensional arrays? They are available in other languages.

You can do the same here as well. An array that contains arrays as its elements is a 2-dimensional array (like a matrix). If you want to create a 4 x 6 matrix, you would declare it as follows:

```
int[][] matrix = new int[4][6];
```

And `matrix[4][6]` accesses the 6th index of the array that is at the 4th index of the `matrix`.

The same can be generalized for any other dimensions not just 2.

Can I declare an array of `integers` and hold `short` values in it?

Yes. The same typecasting concepts are applicable for the values of an array as well.

Quiz

1. If an array is declared to hold 12 integers, what is the size of the array?
2. Can an array be created with zero size? What's the use of it?
3. A multi-dimension array contains arrays as the elements. Should all these arrays must have the same size?
4. What are the properties and methods available for an array (like length property)?

Coding Exercises

1. Write a program to check if the given string is a *palindrome* or not.
2. Write a program to create an array of words found a sentence.
3. Write a program to print sum of two matrices.
4. Write a program to print the product of two matrices.
5. Write a program to print the average of all the odd numbers found in an array.
6. Write a program to print the Fibonacci series.

Chapter 14
Dealing with Command Line Arguments

- Command Line Arguments
- Parsing the arguments

Understanding the Concepts

We have seen two ways of providing input to the program. One is by hard coding the input in the program itself. The other is by inputting the data interactively to a running program. This is the time to look at the third way . We neither hard code the input data nor prompt the user to input the data, in this third approach. Instead, we specify the input data as part of the *command line*. The input data that goes as part of the command line are called *command line arguments*.

Command Line Arguments

To run a Java program called `Multiplier.class`, we use the following command:

```
java Multiplier
```

If you want to pass some input to the program as part of the command line, we change it slightly like this:

```
java Multiplier 23 45
```

The above command passes the values 23 and 45 to the `Multiplier` as *command line arguments*.

String array

Though we see 23 and 45 on the command line as numbers in the above example, Java always considers them as just `String` objects only. No matter what you pass as command line argument, Java reads them as `String` objects.

Not only that, all the arguments are collected into a `String` array and given to the main program. Look here:

```
class Multiplier {
    public static void main(String[] args) {
        … // program
    }
}
```

The parameter named `args` to the `main()` method is what holds all the command line arguments that we pass to this program.

Means, in our example, `args[0]` would contain a `String` whose content is `23` and `args[1]` would contain another `String` whose value is `45`.

Parsing the String

To do the real math using the numbers on the command line, we need to parse the integers from those strings. So, the program looks some thing like this:

```
class Multiplier {
    public static void main(String[] args) {
        int first = Integer.parseInt(args[0]);
        int second = Integer.parseInt(args[1]);
        int product = first * second;
        System.out.println(product);
    }
}
```

If you are expecting `double` values from the command line, then you will have to use `Double.parseDouble()` method. Now you can guess what need to be done for other types.

Applying the Concepts

Objective

Write a program to find the maximum of all the integers passed as command line arguments.

Listing: CommandLineDemo.java

```
1. // Reading command line args
2. class CommandLineDemo {
3.     public static void main(String[] args) {
4.         int max = 0;
5.         System.out.print("Data supplied: ");
6.         for (String arg : args) {
7.             System.out.print(arg + " ");
8.             int number = Integer.parseInt(arg);
9.             if (Integer.parseInt(arg) > max)
```

```
10.                 max = number;
11.         }
12.         System.out.print("\nMaximum: " + max);
13.     }
14. }
```

Explanation

This program reads each of the command line arguments from the `args` array and parses it to an `int`. Then it uses the standard logic to check if it is the largest value read so far and resets if necessary. By the end of the program, whichever number remains as the largest value thats the one is declared as the maximum.

Results

Running the above program using the following test data prints the following results.

```
Data supplied: 1 23 41 17 109 11
Maximum: 109
```

Beyond the Concepts

Interesting Questions

Can I change the data type of the `args` parameter in the `main` method?

You may. Changing the data type would not give compile time error. However, Java always look for the `main` method with `String[]` as the data type of the parameter to start the program. So, when you try to run the program, it results in run time error (stating that the `main` method is not found), if change the data type of `args`.

How do I pass different data types together as command line arguments?

Nothing different. The following command passes an integer and a string as parameters.

```
java Profile Gandhi 78
```

It is your program that should take care of different data types. Only the `args[1]` needs to be parsed as an integer in the program.

Is it a must to parse the Strings before we use them?

You parse them to different data types only when you need them as those data types. Otherwise, its not a rule, if you are just Ok with the strings.

Can I pass comma separated values as command line arguments?

No. Java uses only space as the delimiter, not comma.

Then, how do I pass `space` as the value itself?

Use double quotes. Look at the following snippets.

```
java Profile Mohan Das // two arguments Mohan and Das
java Profile "Mohan Das" // only one argument Mohan Das
```

Among hard coding, interactive inputting and command line arguments, which is the better way to provide the input data to program?

If the input data never changes, go for *hard coding.*
If the user wants to frequently change the input for each run, go for console.
If a given user wants to use the same input for all the runs, but each user wants to use a different input, go for command line arguments.

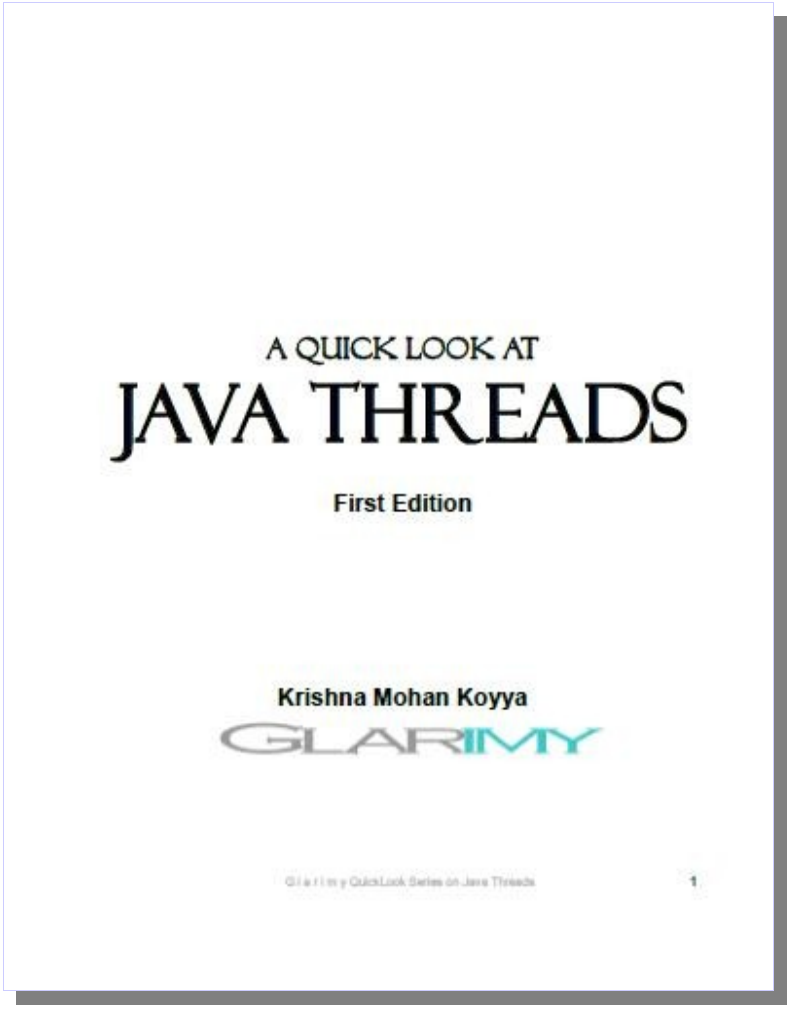
Quiz

1. What is the maximum number of arguments that can be passed to a program?
2. If there are no command line arguments, what is the contents of `args` array?
3. Can you change the name of the array from `args` to something else?

Coding Exercises

1. Write a program to check if the string provided on the command line is a palindrome or not.
2. Write a program to find the average of all the `double` values passed as the command line arguments.
3. Write a program to print the number of command line arguments passed.

Other QuickLook Titles from Glarimy



A Quick Look At

Java 6 Fundamentals

First Edition

Covers basic programming constructs of JDK 1.6

- Compiling and running Java programs
- Data Types, Variables, Operators
- Autoboxing
- Decision making
- Iterations
- Interacting with user
- Formatting Input and output
- Arrays
- Command Line Arguments
- String Handling
-

With complete code illustrations.