Java programming fundamentals, data types, operators

Module 1

JAVA PROGRAMMING FUNDmentals

Evolution Of Java

- Created at Sun MicroSystems
- By James Gosling, Patrick Naughton, Chris Warth, Ed Frank, Mike Sheridan
- Needed platform independent language

OOPS and Java

- Java is a purely an object-oriented language.
- The central idea behind object-oriented Styles programming is to divide a program into isolated parts called Objects.
- Each object contain two parts.
  - Data
  - Functions
- The functions in an object operate on the data involved in that object.

Buzz words

- Simple
- Secure
- Portable
- Object-Oriented
- Robust
- Multi-threaded
- Architecture Neutral
- Interpreted
- High Performance
- Distributed
- Dynamic

Object Oriented Programming

- Computer programs – code and data
- First approach – what is happening (process oriented model)
  - Series of linear steps (code acting on data)
- Second approach – who is being affected (object oriented programming)
  - Organize program around data and have well defined interfaces to that data (data controlling access to code)

Three main OOP principles

- **Encapsulation** – mechanism of binding the data and code that acts on data, together to keep both safe from outside misuse.
- **Inheritance** – process by which one object acquires properties of another object.
- **Polymorphism** – feature that allows one interface to be used for general class of action.
- All these 3 put together produce environment that is robust and scalable.

Encapsulation

Encapsulation is the mechanism to bind the data and code working on that data into a single entity. It provides the security for the data by avoiding outside manipulations.

In Java, encapsulation is achieved using classes. A class is a collection of data and code. An object is an instance of a class That is, several objects share a common structure (data) and behavior (code) defined by that class.

A class is a logical entity (or prototype) and an object is a physical entity. The elements inside the class are known as members. Specifically, the data or variables inside the class are called as member variables or instance variables or data members.

The code that operates on these data is referred to as member methods or methods (In C++, we term this as member function). The method operating on data will define the behavior and interface of a class.

Polymorphism

Polymorphism can be thought of as one interface, multiple methods it is a feature that allows one interface to be used for a general class of actions.

The specific action is determined by the exact nature of the situation. Consider an example of performing stack operation on three different types of data viz. integer, floating-point and characters.

In a non-object oriented programming, we write functions with different names for push and pop operations though the logic is same for all the data types. But in Java, the same function names can be used with data types of the parameters being different.

OOPS and Java - Inheritance

Inheritance is one the most powerful capabilities of object-oriented programming.

Using this concept, a new class of objects can be derived from an old one. This process is called inheritance because the new class inherits the characteristics of the original class. The new class is called a derived class of the original class and the original class is called the base class of the new class.

Key attributes of OOP

- Over-loading
- Over-riding
- Inheritance
- Polymorphism
- Interfaces
- Encapsulation

Java development kit

- Download jdk 6/7
- Install it
- In control panel, go to System->env variable
- Create path var to point to bin dir of jdk and jre
- Create classpath to point to lib of both dir and also add “.” at the start separated by semi-colon
- Close the control panel and start DOS prompt and type javac
- You have to see a list of options appearing.. If java is installed properly, if not command not recognized message is displayed

Java Development Kit (JDK)- Byte code

Bytecodes are a set of instructions that look a lot like machine code, but are not specific to any one processor

Platform-independence doesn't stop at the source level, however. Java binary files are also platform-independent and can run on multiple platforms without the need to recompile the source. Java binary files are actually in a form called bytecodes.

Introduction to Java - JDK Tools

- The important development tools included in JDK are Java compiler, Java interpreter, Java disassembler, Java debugger, tool for C header files, tool for creating HTML documents and tool for viewing Java applets.

- The purpose of each one of these development tools is briefly described in the following table.

First sample program – Example1.java

```
/* this is how comment is written */
// or like this
class Example1
{
   public static void main(String args[])
   {
      System.out.println(“Hello class\n”);
   }
}
```

- Source file – compiling unit
- Should have .java extension
- In java, all code must reside inside a class
- Name of the class should match name of the file, capitalization matters.

Compiling Program

- C:\> javac Example1.java
- .class is the byte code that contains instructions for JVM
- C:\>java Example1
- Comments, class, identifier, class definition, case sensitive, end of statement(;)

Parameters used in First Java Program

Let's see what is the meaning of class, public, static.

void, main, String[], System.out.println().

class keyword is used to declare a class in java.

public keyword is an access modifier which represents visibility. It means it is visible to all.

static is a keyword.

If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create an object to invoke the main method. So it saves memory.

void is the return type of the method. It means it doesn't return any value. main represents the starting point of the program. String] args is used for command line argument. We will learn it later System.out.println() is used to print statement.

Here, System is a class, out is the object of PrintStream class, println() is the method of PrintStream class. We will learn about the internal working of System.out.printin statement later

Basic Syntax

About Java programs, it is very important to keep in mind the following points.

Case Sensitivity – Java is case sensitive, which means identifier Hello and hello would have different meaning in Java.

Class Names - For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case

Example: class MyFirstJavaClass

Method Names - All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

Program File Name - Name of the program file should exactly match the class name.

Handling syntax errors

- Common errors that can occur are –
  - Capitalisation of key words
  - Writing a string over a new line
  - Missing brackets in a no-argument message
  - Forgetting to import a package
  - Treating a static method as if it were an instance method
  - Case-sensitive errors with classes
  - Case-sensitive errors with variables
  - Missing class brackets
  - Specifying method arguments wrongly
  - Omitting void in methods
  - Omitting the return in a method
  - Using a variable before it is given a value
  - Confusing prefix operators with postfix operators

Java Class library

- java.io.*;
- java.lang.*;
- java.util.*;
- java.util.Math.*;
- java.lang.String.*;
- java.io.File.*;

- Variable – named memory location that may be assigned a value.
- Must be declared first before used.
- Data converted to string before display

Blocks of code

- Can group 2 or more statements a block of code called *code blocks.*
- Done by using curly braces.

```
if( x< y)
{
   x= x+y;
   y = x-10;
}  //end of a block
```

Lexical issues

- Java contains
  - White spaces – space, newline, tab
  - Identifiers
  - Literals – constant value, 100, 98.6
  - Comments
  - Operators
  - Separators – (), { }, [ ], ;, comma , .
  - Keywords – 50 currently defined keywords

Identifier

- Used for class name, method name, variable name
- Descriptive sequence of upper case, lower case letters, numbers , underscore, and dollar sign
- Must not begin with number.
- Case-sensitive

Comments

- 3 types
  - Single line
  - Multi line
  - Documentation comments - /** comment */

Java is strongly typed

- Every variable, expression has a type
- All assignments (explicit or implicit) , checked for type compatibility.
- Compiler checks to see type compatibility.
- Java’s safety and robustness from these points.

Introducing data types and operators

Chapter : 2

Primitive data types

- 8 primitive data types
- Integer
  - Byte(-128 to 127) – 8 bits
  - Short(-32,768 to 32, 767) – 16bits
  - Int – 32 bits
  - Long – 64 bits
- Floating point
  - Float - 32 bits
  - Double – 64 bits
- Character – char(0 to 65,536), ASCII ranges from 0 to 127, but here includes all special langs, Uses Unicode representation. Uses for Latin, German, Spanish, French..
- Boolean – boolean (true or false)

Integer literals

- Octal(base 8)
  - numbers with leading 0
  - Range 0 to 7
- Hexa-decimal(base 16)
  - Numbers start with 0x or 0X
  - Range 0 to 15 (0-9 , A-F or a-f)

Different literals

- To specify long type literals, need to do it explicitly by adding L or l
  - 9892374893743743L
- Floating point literal needs decimal and a fraction
  - Std notation – whole number, decimal, fraction
  - Scientific notation – 6.08897E10 or e, 3.06e-06
  - Always defaults to double precision.
  - To specify float literal, use F or f for constant
  - To explicitly specify double literal, use D or d

- Char literal specified inside of ‘’
- Need a back slash for special character(\n).
- String literals specified in “ ”

Variables

- Basic unit of storage
- Defined with combination of identifier, type, optimal initializer.

  type id1 = value1, id2 = value2..

  or

  type id1, id2,…..;
- Dynamic initialization
  - Double c= Math.sqrt(a * b – c);
- Scope and lifetime of variable
  - java_u1_ex2.txt

Type conversion and casting

- Automatic type conversion (widening conversion)
  - Two types are compatible.
  - Destination type is larger than source type.

  Byte->short->int->long

Casting

- Also called as narrowing conversion
- Need to use cast
  - (target_type)value/variable;
  - int a; byte b; b = (byte) a;
- Different conversion occurs when casting float to int – truncation.

Automatic promotion in expression

- Another place for conversion – expressions
- java_u1_ex3.txt
- Java automatically promotes byte, short, char to int.
- Also cause of confusion.

Type promotion rules

- Byte, short, char all promoted to int
- If 1 operand is long, whole expression is long
- If 1 operand is float, whole expression is float
- If 1 operand is double, whole expression is double.

Operators

- Divided into 4 groups
  - Arithmetic
  - Bitwise
  - Relational
  - Logical
  - Instanceof (special operator, comes later)

Arithmetic

- +, - *, /, %
- Division on int truncates the value after decimal
- Modulus returns remainder of division.
- Compound assignment operator

  a = a+4 can be a += 4;

  a = a %2 can be a %= 2;

  var = var op expression is var op=expression
- Increment & decrement

  i = i+1 or i++; similarly i--; (prefix and postfix)

Very big difference in large expressions.

Bitwise

- Applied on integer types – byte, short, int, long, char
- Int stored in bit form represented by binary
- Ints are signed numbers
- Negative numbers are stored in 2’s complement.

- Bitwise logical op = ~, |, &, ^
- Left shift : value << num
- Has the effect of doubling the value
- The highest bit value is lost and 0 is filled in lowest bit. for signed, 1 is filled
- Right shift: value >> num
- Divides the value by 2, leftmost bit, fills the empty space. Imp for signed ints

- Unsigned right shift
  - Value >>> num
- When low order bit is not needed to fill high order bit, use unsigned
- Adds 0’s to the high end.
- Compound assignment operator also available

Relational operator

- Determine relationships
- ==, !=, >, <, >=, <=
- Outcome is Boolean
- Boolean logical operators - &, |, ^,

  ||, && ! – short circuit operators

  ?: - ternary if-then-else (exp ? Exp1:Exp2)

Precedence

- Separators – (), [], .
- Parentheses raises precedence of operation inside of them.
  - Does not degrade performance.
  - Removes ambiguity
  - Page 75 for complete precedence

Program control statements

Chapter : 3

Input chars from keyboard

- System.in.read() read data from keyboard, but need to add java.io.IOException
- Need to read the whole line, then
- Scanner scanStr = new Scanner(System.in);
- Now use scanStr to read data till hasNext()
- Refer to page 544 for further example
- More to come later

Control Statements

- Control statements – causes flow of execution to advance and branch based on changes
- Selection – allows program to choose different path based on outcome of expression
- Iteration – allows program execution to repeat one or more number of times.
- Jump – allows program to execute in non-linear fashion.

Selection statements

- if (condition) st1;
  else st2;
- Can have nested ifs
- If-else-if ladder

```
if(cond)
        st;
else if (cond)
     st;
else if (cond)
…..
…..
else
   st;
```

- Switch – multi way branch statement

```
switch(exp)
{
  case val1 :
                    statement seq
                    break;
  case valn:
                    can have more switch()
                    break;
  default    :
                    default sts
}
```

- Exp has to be of type byte, short, int or char.
- valn has to be unique in each case.
- Default is optional.
- If no case or default case… no action is taken.
- Break used to terminate sequence.
- If no break, then execution passes to next case.
- java_u2_ex1.txt
- Nested switch also possible.

Iteration

- for, while, do-while

```
While(cond){
    body of loop
}
```

If cond is true, body of loop executed, else statement immediately after closing braces executed.

- Do-while

```
do{
    //body of loop
}while(condition);
```

- Body of loop is executed at least once
- Condition must be boolean
- Works well for menu selection

- for(initialization;condition;iteration)
- {
  //body
  }
- First initialization takes place, loop control variable is set here which acts as a counter.
- Condition is evaluated next, if true loop continues
- Iteration part either increments or decrements the loop control variable.
- Can declare and initialize loop control variable inside for loop, scope matters
- java_u2_ex2.txt

Variations of For loop

- Very powerful as it has 3 parts mixed together into 1
- The condition need not always test the loop control var, all it has to be is a boolean expression
- We can omit out some parts of the loop as well or create an infinite loop
- We can include multiple initializations and multiple increment, decrement in the for loop.
- java_u2_ex3.txt

Another flavor of For loop

- Also called as enhanced for loop

  for(type itr-var : collection)

  statement-block
- Suitable for collections, like arrays
- Type must be same as element stored in array
- java_u2_ex4.txt
- Possible to break out of loop by using condition and break statement.

Jump statements

- Break, continue, return
- Break has 3 uses
  - Terminates statement sequence
  - Exit out of loop
    - In nested loops, will break out of inner most loop
  - Can be used as “civilized” form of goto.
    - No goto in java
    - Break label;
    - java_u2_ex5.txt

Continue/Return

- Continue - Helps running thro’ the loop but skipping some part of the code.
- Return – used explicitly to return from a method and transfer the control back to caller of method.
- java_u2_ex6.txt
- Return shown once classes are handled.

Introducing classes, objects and methods

Chapter : 4

Class

- Defines shape and nature of object
- Basis of object oriented programming
- Class is blue-print for creating an object
- Class doesn't occupy memory space while an object does
- Fundamental to java
- Class – template for an object
- Object – instance of a class
  - Object and instance used inter-changeably.

General form of class

- Declared by use of “class” keyword.
- java_u2_ex7.txt
- Variables – instance variables, each obejct has its own variables
- Code – methods
- Data & code together – members of class
- General rule – Methods determine how the class’s data can be used.
- Every class doesn’t need main().
- Applets doesn’t require main() at all.

- Class defines new data type.
- Class – only template, no actual object created.
- To actually create object, use “new”
- Every object created will have its own copy of variables and methods. To call use the dot(.) operator.
- java_u2_ex8.txt
- java_u2_ex9.txt

Declaring Objects

- Even for objects, 2 step process – first declare, then initialize using new operator
- New provides reference to memory location or address of memory location
- Constructor – defines what occurs when an object is created. If no constructor is present, then default is provided.
- Memory allocated during run-time.
- When we assign 1 object reference variable to another, we only create copy of reference , not object.

Methods

- General form

```
Access d_type name(param_list)
{
    //body of method
}
```

  - Type – data returned by method
  - Name – name of method
  - java_u2_ex10.txt

- Methods alter data and provide a cleaner implementation for user.
- Apart from method that alter data, we can have methods that are used by class itself.
- Returning a value – data returned must be compatible.

Methods that take parameter

- Parameter – variable defined by a method that takes values
- Argument – value that is passed to a method, when it is invoked.

Constructors

- Objects initialized through constructors
- Has same name as class and syntactically similar to method.
- No return type, not even void.
- Constructors initializes all instance variables to 0
- Can have parameterized constructors as well
- java_u2_ex12.txt

this keyword

- this can be used inside any method to refer to current object.
- java_u2_ex11.txt
- Instance variable hiding –
  - illegal for java to have 2 variables of same name in same block of code.
  - can have formal parameters to method with same name as local variable.
  - java_u2_ex11.txt

Instance Variable Hiding

As we know, in Java, we can not have two local variables with the same name inside the same or enclosing scopes.

But we can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables.

However, when a local variable has the same name as an instance variable, the local variable hides the instance variable.

Garbage collection

- In C++, manually release the memory using delete operator.
- Here, de-allocation is automatic, by process of garbage collection.
- Occurs sporadically.
- Different manufacturers use different methods, programmers not to worry.

Garbage Collection

- In C and C++, dynamically allocated variables/objects must be manually released using delete operator.
- But, in Java, this task is done automatically and is called as garbage collection.
- When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- Garbage collection occurs once in a while during the execution of the program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection

Finalize() method

- Sometimes, java might need to complete some tasks before object is destroyed.
- Java provides finalization by method finalize().
- JRT calls this method, before object is about to be recycled.
- But no way of knowing if the code is going to be executed, need other means of releasing resources.

The finalize() Method

Sometimes an object will need to perform some action when it is destroyed.

For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed.

To handle such situations, Java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object in just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the finalize() method. The Java run time calls that method whenever it is about to recycle an object of that class.

- General form is

```
protected void finalize()
{
    //method body
}
```

New operator re-visited

- New used to allocate memory for an object from heap
- Not used for primitive data types, which make java process these kind of data much faster
- New allocates memory at run time
- Good programming practice, as if no sufficient memory during runtime, program stops
- A class creates a new data type or construct
- Format is new className();

- Very similar stack example for use of constructor, methods and use of objects in main().
- To read data from keyboard, use System.in.read() which usually returns integers, need to cast it to char, if using characters for comparison.
- If using read(), then need to include java.io.Exception clause.

A Stack Class

To summarize the concepts of encapsulation, class, constructor, member initialization etc, we will now consider a program to implement stack operations.

Concept of Stack:

A stack is a Last in First Out (LIFO) data structure. Following figure depicts the basic operations on stack:

Inserting an element into a stack is known as push operation,

Whereas deleting an element from the stack is

pop operation.

An attempt made to push an element into a full stack is stack overflow and an attempt to delete from empty stack is stack underflow.

More data types and operators

Chapter : 5

Arrays

- Group of like –typed variables, referred to by common name.
- Specific element accessed using index.

  Type var_name[]; // no memory allocated

  Var_name = new type[size];
- Index starts from 0;
  - java_u3_ex1.txt

```
public static void main(String args[]) {
  int month_days[];
  month_days = new int[12];
  month_days[0] = 31;
  month_days[1] = 28;
  month_days[2] = 31;
  month_days[3] = 30;
  month_days[4] = 31;
  month_days[5] = 30;
  month_days[6] = 31;
  month_days[7] = 31;
  month_days[8] = 30;
  month_days[9] = 31;
  month_days[10] = 30;
  month_days[11] = 31;
  System.out.println("April has " + month_days[3] + " days.");
}
```

```
int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
                     30, 31 };
System.out.println("April has " + month_days[3] + " days.");
```

- Can initialize array when declared
- Array initializer is list of comma separated values/expressions surrounded by curly braces.
- int dates[] = { 1,2,3,4,5,6,7};
- Int dates[] = {0*0, 6+5, 3-1};
- If array accessed out of range, run time error occurs.
- java_u3_ex1.txt

For-each for multi-dimensional array

- Array of arrays
- When iterating, each iteration obtains next array, not individual element.
- java_u3_ex2.txt

Multi dimensional array

- Array of arrays

int multiD[][] = new int[10][10];

int multiD[][] = new int[10][];

multiD[0] = new int[10];

multiD[1] = new int[10];

- java_u3_ex2.txt
- Int multi[][][] = new int[5][5][5];
- Alternative array declaration

  type[] var_name;

- Alternative array declaration syntax
  - type[] var_name;
  - java_u3_ex1.txt
  - java_u3_ex2.txt
- Assigning array references and sending back
  - java_u3_ex3.txt
- Using the length member – array special var in java, which has length attr as public and final which contains number of components of array
  - java_u3_ex3.txt
- For-Each style for arrays - java_u3_ex1.txt

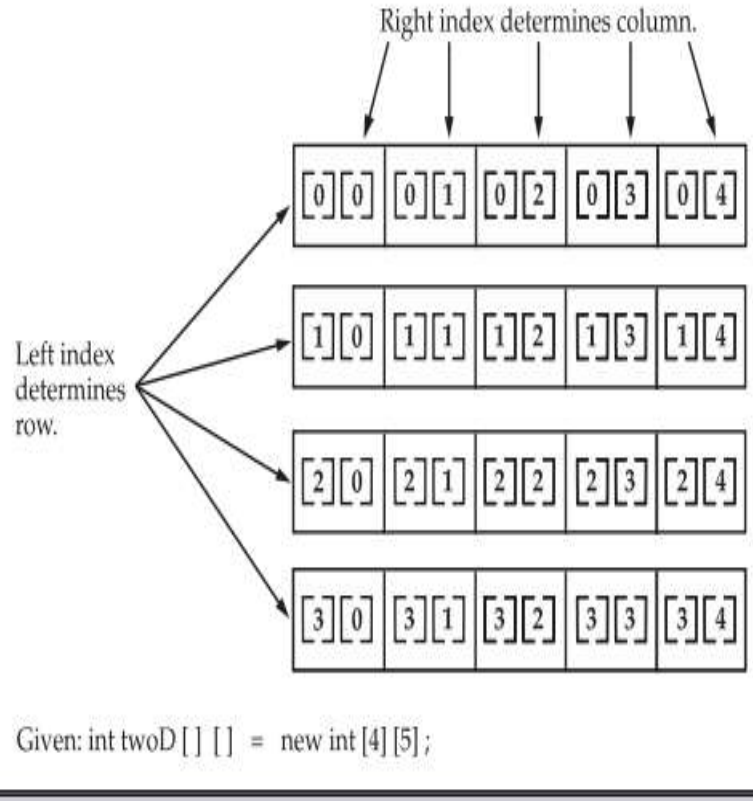

Figure 3-1. *A conceptual view of a 4 by 5, two-dimensional array*

```
// Demonstrate a two-dimensional array.
class TwoDArray {
  public static void main(String args[]) {
    int twoD[][]= new int[4][5];
    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<5; j++) {
        twoD[i][j] = k;
        k++;

      }

    for(i=0; i<4; i++) {
      for(j=0; j<5; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

```
// Manually allocate differing size second dimensions.
class TwoDAgain {
  public static void main(String args[]) {
    int twoD[][] = new int[4][];
    twoD[0] = new int[1];
    twoD[1] = new int[2];
    twoD[2] = new int[3];
    twoD[3] = new int[4];

    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<i+1; j++) {
        twoD[i][j] = k;
        k++;
      }

    for(i=0; i<4; i++) {
      for(j=0; j<i+1; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

This program generates the following output:

```
0
1 2
3 4 5
6 7 8 9
```

The array created by this program looks like this:

Assigning Array References

As with other objects, when you assign one array reference variable to another, you are simply changing what object that variable refers to. You are not causing a copy of the array to be made, nor are you causing the contents of one array to be copied to the other.
For example, consider this program

The drawback of the enhanced for loop is that it cannot traverse the elements in reverse order. Here, you do not have the option to skip any element because it does not work on an index basis. Moreover, you cannot traverse the odd or even elements only. But, it is recommended to use the Java for-each loop for traversing the elements of array and collection because it makes the code readable.

Advantages

- It makes the code more readable.
- It eliminates the possibility of
- programming errors.

Syntax

The syntax of Java
for-each loop consists of data_type with the variable follwed by a colon (;), then
aray or collection.

```
for(data type variable : aray | collection){
//body of for-each lop
}
```

How it works?

The Java for-each loop traverses the array or collection until the last element.
For each element, it stores the element in the variable and executes the body of the for-each loop.

Let us see another of Java for-each loop where we are going to total the elements.

```
Class ForEachExample {
public static void main(strino args[])
{
int arrt]={12,13,14,44}:
```

The for-each style for loop

It provides an alternative approach to traverse the array or collection in Java. It is mainly used to traverse the array or collection elements. The advantage of the for-each loop is that it eliminates the possibility of bugs and makes the code more readable. It is known as the for-
cach loop because it traverses each glement one by one

Strings Handling

Chapter : 6

Strings

- String not a data type, nor array of chars
- Strings – objects, variables

  String str = “This is a string”;

  String s1 =“This is a string”;

- System.out.println(str);

Java Strings

Strings are used for storing text. A String variable contains a collection of characters surrounded by double quotes:

Example

Create a variable of type String and assign it a value:

String greeting = "Hello";

In Java, String is a class but not array of characters. So, the features of String class can be better understood after learning about the concepts of classes in further chapters. For the time-being, we will glance at String type.

Exploring String fundamentals

- One of the most commonly used class
- You actually create a string object or
- Every string const created using literals
- Strings are immutable
- If you want to alter, create a new object
- To make alterations, java provides StringBuffer class
- To concatenate, use “+”

i

String Buffer and String Builder

- StringBuffer – peer class of string
  - Provides most of functionality of string class
  - Provides growable and writeable char seq
  - Can automatically grow to accommodate extra chars
  - Is multi thread safe, mutable(changeable)
  - https://www.javatpoint.com/StringBuffer-class
- String Builder – added in jdk 5
  - Identical to StringBuffer
  - Not synchronized, meaning not thread safe
  - Faster performance

Using command line args

- When we want to pass info to the prg while executing it, we use command line
- Args follow immediately after the program name
- Info is stored as strings in an array
- First arg is stored in arg[0], arg[1]…
- Even numbers are passed as strings, need to convert back to numbers
- java_u3_ex8.txt

We can have array of strings.
A set of characters enclosed within double quotes can be assigned to a String variable.

One variable of type String can be assigned another String variable.
Object of type String can be used as an argument to println as-

```
String str="Hello";
System.out.printin(str);
```

String Handling : String Fundamentals:

A string is a sequence of character and Java implements strings as objects of type String. Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient.

String Handling: String Fundamentals:

For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. Also, String objects can be constructed a number of ways, making it easy to obtain a string when needed

Once a String object has been created, you cannot change the characters of that string. Whenever we need any modifications, a new string object containing modifications has to be created.

However, a variable declared as String reference can point to some other String object, and hence can be changed

In case, we need a modifiable string, we should use StringBuffer or StringBuilder classes.

String, StringBuffer and StringBuilder classes are in java.lang and are final classes. Thus, no class can inherit these classes. All these classes implement CharSequence interface.

The String Constructors

There are several constructors for Sting class.

1. To create an empty string, use defaut constructor

```
String s= new String () ;
```

2 To create a sting and initialize:

```
String s= new String ("Hello") ;
```

3. To create a sting object that contains same characters as another string objedt

```
String (String strobj);
```

For example,
String s= new String (“Hello");
String sl= new String (s);
4. To create a string having initial values:
String (char chars [])
For example,
char ch[]=1 "h', 'e', '1', '1', *o'l
String s new String (ch) /s contains hello
5. To speclty a sub-range of a character array as an initializer use the following constructor:
String (char chars[1, int startIndex, int numChars)
For example,
char ch[l-('a", b', 'e', *d', 'e', '*', 'g'}
String s new String (ch, 2, 3)

Even though Java's char type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set. Because 8-bit ASCII strings are common, the String class provides constructors that initialize a string when given a byte
array.

String Length

A String in Java is actually an object, which contain methods that can perform certain operations on strings. For example, the length of a string can be found with the length) method:

Example

```
String txt= "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
System.out.println("The length of the txt string
is: " + txt.length(O);
```

toUpperCase() and toLowerCase) methods

There are many string methods available, for example toUpperCase(o and toLowerCase():
Example
String txt= "Hello World";
System.out.println(txt.toUpperCase(o): 1

Outputs "HELLO WORLD"

System.out.println(txt.toLowerCaseo); /

Outputs "hello world"

Finding a Character in a String

The indexOf method returns the index (the position) of the first occurrence of a specified text in a string (including whitespace):

Example

```
String txt = "Please locate where 'locate‘ occurs!";
System.out.println(txt.indexOf"locate");

//
Outputs
7
```

```
public class Main{

public static void main(String[] angs) {

String txt= "Please locate where "locate' occurs!";

System.out.println(txt.indexOf("locate");

}
```

Output

7

String Concatenation

The +operator can be used between strings to combine them. This is called concatenation:

Example

```
String firstName = "John";
String lastName = "Doce";
System.out.println(firstName+" " + lastName);
```

Concat method

You can also use the concat() method to concatenate two strings:

Example

```
String first Name = "John ";
String lastName = "Doe";
System.out.println(firstName.concat(last Name);
```

Special Characters

Because strings must be written within quotes, Java will misunderstand this string, and generate an error:

String txt= "We are the so-called "Vikings" from the north.";

The solution to avoid this problem, is to use the backslash escape character.

The backslash () escape character turns special characters into string characters:

For objects, valueOf( ) calls the toString() method on the object. Every class implements toString() because it is defined by Object.

However, the default implementation of toString() is seldom sufficient. For our own classes, we may need to override toString) to give our own string representation for user-defined class objects. The toString) method has this general form:

String toString()

String Conversion and toString():

Java uses valueOf() method for converting data into its string representation during concatenation.
valueOf() is a string conversion method defined by String.
valueOf( ) is overloaded for all the primitive types and for type Object.
For the primitive types, valueOf( ) returns a string that contains the human-readable equivalent of the value with which it is called

For objects, valueOf( ) calls the toString() method on the object. Every class implements toString() because it is defined by Object.

However, the default implementation of toString() is seldom sufficient. For our own classes, we may need to override toString() to give our own string representation for user-defined class objects. The toString) method has this general form:

String toString()

charAt( )

To extract a single character from a **String**, you can refer directly to an individual character via the **charAt( )** method. It has this general form:

char charAt(int *where*)

Here, *where* is the index of the character that you want to obtain. The value of *where* must be nonnegative and specify a location within the string. **charAt( )** returns the character at the specified location. For example,

```
char ch;
ch = "abc".charAt(1);
```

assigns the value "**b**" to **ch**.

getChars( )

If you need to extract more than one character at a time, you can use the **getChars( )** method. It has this general form:

void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)

Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from *sourceStart* through *sourceEnd*–1. The array that will receive the characters is specified by *target*. The index within *target* at which the substring will be copied is passed in *targetStart*. Care must be taken to assure that the *target* array is large enough to hold the number of characters in the specified substring.

The following program demonstrates **getChars( )**:

```
String s = "This is a demo of the getChars method.";
int start = 10;
int end = 14;
char buf[] = new char[end - start];

s.getChars(start, end, buf, 0);
System.out.println(buf);
```

startsWith( ) and endsWith( )

String defines two routines that are, more or less, specialized forms of **regionMatches( )**. The **startsWith( )** method determines whether a given **String** begins with a specified string. Conversely, **endsWith( )** determines whether the **String** in question ends with a specified string. They have the following general forms:

boolean startsWith(String *str*)
boolean endsWith(String *str*)

Here, *str* is the **String** being tested. If the string matches, **true** is returned. Otherwise, **false** is returned. For example,

```
"Foobar".endsWith("bar")
```

and

```
"Foobar".startsWith("Foo")
```

are both **true**.

A second form of **startsWith( )**, shown here, lets you specify a starting point:

boolean startsWith(String *str*, int *startIndex*)

Here, *startIndex* specifies the index into the invoking string at which point the search

- **getBytes()** : It is an alternative to **getChars()** that stores the characters in an array of bytes. It uses the default character-to-byte conversions provided by the platform. Here is its simplest form:

```
byte[ ] getBytes( )
```

Other forms of **getBytes( )** are also available. **getBytes( )** is most useful when you are exporting a **String** value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

- **toCharArray() :** If you want to convert all the characters in a **String** object into a character array, the easiest way is to call **toCharArray( )**. It returns an array of characters for the entire string. It has this general form:

  char[ ] toCharArray( )

  This function is provided as a convenience, since it is possible to use **getChars( )** to achieve the same result.

```
String s1="hello";
char[] ch=s1.toCharArray();

for(int i=0;i<ch.length;i++)
    System.out.print(ch[i]);
```

String Comparison

equals( ) and equalsIgnoreCase( )

To compare two strings for equality, use **equals( )**. It has this general form:

boolean equals(Object *str*)

Here, *str* is the **String** object being compared with the invoking **String** object. It returns **true** if the strings contain the same characters in the same order, and **false** otherwise. The comparison is case-sensitive.

To perform a comparison that ignores case differences, call **equalsIgnoreCase( )**. When it compares two strings, it considers **A-Z** to be the same as **a-z**. It has this general form:

boolean equalsIgnoreCase(String *str*)

Here, *str* is the **String** object being compared with the invoking **String** object. It, too, returns **true** if the strings contain the same characters in the same order, and **false** otherwise.

Here is an example that demonstrates **equals( )** and **equalsIgnoreCase( )**:

```
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {
  public static void main(String args[]) {
    String s1 = "Hello";
    String s2 = "Hello";
    String s3 = "Good-bye";
    String s4 = "HELLO";
    System.out.println(s1 + " equals " + s2 + " -> " +
                       s1.equals(s2));
    System.out.println(s1 + " equals " + s3 + " -> " +
                       s1.equals(s3));
    System.out.println(s1 + " equals " + s4 + " -> " +
                       s1.equals(s4));
    System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
                       s1.equalsIgnoreCase(s4));
  }
}
```

regionMatches( )

The **regionMatches( )** method compares a specific region inside a string with another specific region in another string. There is an overloaded form that allows you to ignore case in such comparisons. Here are the general forms for these two methods:

```
boolean regionMatches(int startIndex, String str2,
                      int str2StartIndex, int numChars)

boolean regionMatches(boolean ignoreCase,
                      int startIndex, String str2,
                      int str2StartIndex, int numChars)
```

equals( ) Versus ==

It is important to understand that the **equals( )** method and the == operator perform two different operations. As just explained, the **equals( )** method compares the characters inside a **String** object. The == operator compares two object references to see whether they refer to the same instance. The following program shows how two different **String** objects can contain the same characters, but references to these objects will not compare as equal:

```
// equals() vs ==
class EqualsNotEqualTo {
  public static void main(String args[]) {
    String s1 = "Hello";
    String s2 = new String(s1);

    System.out.println(s1 + " equals " + s2 + " -> " +
                       s1.equals(s2));
    System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
  }
}
```

compareTo( )

Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is *less than*, *equal to*, or *greater than* the next. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The **String** method **compareTo( )** serves this purpose. It has this general form:

int compareTo(String *str*)

Value	Meaning
Less than zero	The invoking string is less than *str*.
Greater than zero	The invoking string is greater than *str*.
Zero	The two strings are equal.

Searching Strings

The String class provides two methods indexOf) and lastIndexOf) that allow you to search a string for a specified character or substring. Both these methods are overloaded to take different types of arguments for doing specific tasks as listed in the table given below-

Method	Purpose
int indexOf(int *ch*)	To search for the first occurrence of a character
int lastIndexOf(int *ch*)	To search for the last occurrence of a character,
int indexOf(String *str*)	To search for the first or last occurrence of a substring
int lastIndexOf(String *str*)	
int indexOf(int *ch*, int *startIndex*)	Used to specify a starting point for the search. Here, *startIndex* specifies the index at which point the search begins. For **indexOf()** method, the search runs from *startIndex* to the end of the string. For **lastIndexOf( )** method, the search runs from *startIndex* to zero.
int lastIndexOf(int *ch*, int *startIndex*)	
int indexOf(String *str*, int *startIndex*)	
int lastIndexOf(String *str*, int *startIndex*)	

Modifying a String

Since String objects cannot be changed, whenever we want to modify a String, we must either copy it into a StringBuffer or StringBuilder, or use one of the following
String methods, which will Construct a new copy of the string with our modifications complete

substring( )

You can extract a substring using **substring( )**. It has two forms. The first is

String substring(int *startIndex*)

Here, *startIndex* specifies the index at which the substring will begin. This form returns a copy of the substring that begins at *startIndex* and runs to the end of the invoking string.

The second form of **substring( )** allows you to specify both the beginning and ending index of the substring:

String substring(int *startIndex*, int *endIndex*)

concat( )

You can concatenate two strings using **concat( )**, shown here:

String concat(String *str*)

This method creates a new object that contains the invoking string with the contents of *str* appended to the end. **concat( )** performs the same function as **+**. For example,

```
String s1 = "one";
String s2 = s1.concat("two");
```

replace( )

The **replace( )** method replaces all occurrences of one character in the invoking string with another character. It has the following general form:

String replace(char *original*, char *replacement*)

Here, *original* specifies the character to be replaced by the character specified by *replacement*. The resulting string is returned. For example,

```
String s = "Hello".replace('l', 'w');
```

puts the string "Hewwo" into **s**.

trim( )

The **trim( )** method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form:

String trim( )

Here is an example:

```
String s = "   Hello World    ".trim();
```

The method **toLowerCase()** converts all the characters in a string from uppercase to lowercase. The **toUpperCase()** method converts all the characters in a string from lowercase to uppercase. Non-alphabetical characters, such as digits, are unaffected. Here are the general forms of these methods:

```
String toLowerCase()
String toUpperCase()
```

For example,

```
String str = "Welcome!";
String s1 = str.toUpperCase();
System.out.println(s1);          //prints WELCOME!
String s2= str.toLowerCase();
System.out.println(s2);          //prints welcome!
```

StringBuffer Class

We know that, String represents fixed-length, immutable character sequences. In contrast, StringBuffer represents growable and writeable character sequences. We can insert characters in the middle or append at the end using this class. StringBuffer will automatically grow to make room for such additions and often has more characters pre-allocated than are actually needed, to allow room for growth.

Method	Description
CharSequence subSequence(int *startIndex*, int *stopIndex*)	Returns a substring of the invoking string, beginning at *startIndex* and stopping at *stopIndex*. This method is required by the **CharSequence** interface, which is now implemented by **StringBuffer**.
int indexOf(String *str*)	Searches the invoking **StringBuffer** for the first occurrence of *str*. Returns the index of the match, or –1 if no match is found.
int indexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the first occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*)	Searches the invoking **StringBuffer** for the last occurrence of str. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the last occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.

S.No.	Method	Description
1.	append(String str)	Append the specified string at the end of this string.
2.	insert(int offset, String str)	Insert specified string at the offset indicated position.
3.	replace(int startIndex, int endIndex, String str)	Replace the substring of the string buffer from startIndex to endIndex-1 with specified string.
4.	delete(int startIndex, int endIndex)	delete the substring of the string buffer from startIndex to endIndex-1.
5.	reverse()	replace the string buffer's character sequence by reverse character sequence.
6.	capacity()	returns the current capacity of string buffer. Capacity refers to the amount of available storage.
7.	ensureCapacity(int minCapacity)	ensures that the capacity is at least equal to the specified minimum.

Data Conversion Using valueOf( )

The **valueOf( )** method converts data from its internal format into a human-readable form. It is a static method that is overloaded within **String** for all of Java's built-in types, so that each type can be converted properly into a string. **valueOf( )** is also overloaded for type **Object**, so an object of any class type you create can also be used as an argument. (Recall that **Object** is a superclass for all classes.) Here are a few of its forms:

```
static String valueOf(double num)
static String valueOf(long num)
static String valueOf(Object ob)
static String valueOf(char chars[ ])
```

ASSIGNMENT QUESTIONS

Briefly explain the characteristics features of java programming
Explain the key attributes of object oriented programming?
List and explain the Java's primitive data types.
Write the syntax of for loop, enhanced for loop, while loop, do-while, with suitable example
Explain the following (i) new operator (ii) this keyword (iii) break (iv) continue (v)IndexOf() (vi) lastIndexOf() with an example.
List and explain the visibility and scope of the access specifier?
What is type casting? Illustrate with an example. What is meant by automatic type promotion? Give an example.
What is the use of StringBuffer? Explain StringBuffer class with a suitable example