**Java Language Basics**

* **Variables**
  + The term fields and variables both often refer to the same thing.
  + Object stores its states in fields.
  + The java programming language defines the following kinds of variables
    - **Instance Variables(Non-Static variables)**
      * Fields declared without static keyword
      * Their values are unique to each instance of the class
    - **Class Variables (Static Variables)**
      * Fields declared with static keyword
      * Tells the compiler that there is one copy of this variable in existence, regardless of how many times a class has been instantiated.
    - **Local Variables**
      * Methods will often stores its temporary states in local variable
      * Which is between opening and closing braces of the method
      * Local variables are only visible to the methods, in which they are declared
      * They are not accessible from outside of the class.
    - **Parameters**
      * Parameters are always classified as variables not fields.
      * A variable declared within the opening and closing parenthesis of a method is called parameter.
  + **Variable Naming Conventions**
    - Variable names are case sensitive
    - Subsequent characters may be letters, digits, dollar or underscore
    - The names you choose consist of only one word, spell that word in all lower case letters.
* **Primitive Data Types**
  + Primitive type is predefined by language and is named by reserved keyword
  + The eight primitive data types supported by java programming languages are
    - Byte
      * 8 bit signed two’s complement integer
    - Short
      * 16 bit signed two’s complement integer
    - Int
      * 32 bit signed two’s complement integer
    - Long
      * 64 bit signed two’s complement integer
    - Folat
      * Single precision 32 bit
    - Double
      * Double precision 64 bit
    - Boolean
      * Two possible values, true and false
    - Char
      * Single 16 bit Unicode character
  + **Default Values**
    - Fields that are declared but not initialized will be set to reasonable default value by the compiler
    - The default value will be zero or null depending on the data type
    - Compiler never assign the default value to an uninitialized local variables

|  |  |
| --- | --- |
| **Data Type** | **Default Value (for fields)** |
| Byte | 0 |
| Short | 0 |
| Int | 0 |
| Long | 0L |
| Float | 0.0f |
| Double | 0.0d |
| Char | '\u0000' |
| String (or any object) | null |
| Boolean | false |

* **Arrays**
  + Array is a container object that holds the fixed number of values of a single type.
  + Length of the array is established when the array is created.
  + After creation its length is fixed.
  + Each item in an array is called an element.
  + Each element is accessed by its numerical index.
* **Declaring a Variable to Refer to an Array**
  + Int[] anArray – declares an array of integers
  + Array declaration has two components, the array’s type and array’s name.
  + Declaration does not actually creates an array
  + It tells the compiler that this variable will hold an array of the specified type.
* **Creating, Initializing and Accessing the Array**
  + One way to create an array is with the new operator.
  + Int[] anArray = new int[10];
  + If this statement were missing, compiler would print an error like the following and compilation will fail.
  + ArrayDemo.java.4:Variable anArray may not have been initialized
  + Alternatively you can use the shortcut syntax to create and initialize the array

int [] anArray = {

100, 200, 300,

400, 500, 600

}

* + The length of array is determined by the number of values provided between {and}
  + Multidimensional array is simply an array, whose components are themselves arrays.

String[][] names = {

{“Mr.”, “Mrs.”, “Ms.”} ,

{“Regu”, “Sanjay”}

}

* **Copying Arrays**
  + System class has an arraycopy method that you can use to efficiently copy data from one array into another.

Public static void arraycopy (Object src, int srcPos,

Object dest, int destPos, int length )

* + The two object arguments specify the array to copy from and the array to copy to.
  + Three int arguments specify the starting position in the source array, starting position in the destination array and the number of elements to copy.

* **Operators**
  + Operators are special symbols that perform the operation on one, two or three operands and then return the result.
  + Operators with higher precedence are evaluated before operators with relatively lower precedence.
  + All the binary operators except from assignment operators are evaluated from left to right.

|  |  |
| --- | --- |
| **Operator Precedence** | |
| **Operators** | **Precedence** |
| postfix | *expr*++ *expr*-- |
| unary | ++*expr* --*expr* +*expr* -*expr* ~ ! |
| multiplicative | \* / % |
| additive | + - |
| shift | << >> >>> |
| relational | < > <= >= instanceof |
| equality | == != |
| bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | | |
| logical AND | && |
| logical OR | || |
| ternary | ? : |
| assignment | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |

* + Assignment operators are evaluated from right to left.
* **Assignment, Arithmetic and Unary Operators**
  + - **Simple Assignment Operator**
      * It will assign the value on its right to the operand on its left.
      * Example: int speed = 0;
      * This operator can also be used on objects to assign object reference.
    - **Arithmetic Operator** 
      * Provides operators that perform addition, subtraction, multiplication and division.
      * % Operator divides one operand by another and returns the reminder as its result.

+ additive operator (also used for String concatenation)

- subtraction operator

\* multiplication operator

/ division operator

% remainder operator

* + - * You can also combine the arithmetic operators with simple assignment operators to create compound assignment. Eg x += 1;
      * + Operator can also be used for concatenating two strings together.
    - **Unary Operator**
      * Unary operators require only one operand.
      * It performs various operations such as incrementing/Decrementing a value by one, negating the expression or inverting the value of Boolean.

+ Unary plus operator; indicates positive value (numbers are positive without this, however)

- Unary minus operator; negates an expression

++ Increment operator; increments a value by 1

-- Decrement operator; decrements a value by 1

! Logical complement operator inverts the value of a boolean

* + - * Increment/Decrement operator can be applied before or after the operand.
      * Prefix (++result) version evaluates to the incremented value.
      * Postfix (result++) version evaluates to the original value.
* **Equality, Relational and Conditional Operators**
  + **Equality and Relational Operators**
    - Equality and relational operators determine, if one operand is greater than, less than, equal to or not equal to another operand.
      * + == Equal to
        + != Not Equal to
        + > Greater than
        + >= Greater than or equal to
        + < Less than
        + <= Less than or equal to
  + **Conditional Operators**
    - The && and || operators perform Conditional-AND and Conditional-OR operations on two Boolean expressions.
      * + && - Conditional-AND
        + || - Conditional-OR
    - Another conditional operator is :?, This is shorthand of if-than-else statement.
    - This operator is known as ternary operator because it uses three operands.
  + **Type Comparison Operator** 
    - The instanceof operator compares the object to a specified type.
    - You can use it to test if the object is instance of a class, an instance of a subclass, or an instance of a class that implements the particular interface.
* **Bitwise and Bit Shift Operators** 
  + The unary bitwise complement operator ~ inverts the bit pattern.
  + It can be applied to any of the integral type, making every 0 as 1 and every 1 as 0.
  + The signed left shift operator << shift the bit pattern to left.
  + The signed right shift operator >> shift the bit pattern to right.
  + The unsigned right shift operator >>> shift the zero into left most position.
  + The bitwise & operator performs bitwise-AND operation
  + The bitwise ^ operator performs bitwise exclusive-OR operation
  + The bitwise | operators performs bitwise inclusive-OR operation
* **Expressions, Statements and Blocks**
* **Expressions**
  + An Expression is a construct made up of variables, operators and method invocations.
  + Which are constructed according to the syntax of the language that evaluates to a single value.
  + Data type of the value returned by the expression depends on the elements used in an expression.
* **Statements**
  + Statements form the complete unit of execution.
  + Statements end with semicolon.
  + The following types of expressions can be made into statement by terminating the expression with semicolon.
    - Assignment expressions
    - Any use of ++ and –-
    - Method invocations
    - Object creation expressions
  + Such statements are called expression statements.
  + Declaration statement declares the variable.
  + Control flow statements regulate the order in which statements get executed.
* **Blocks**
  + Blocks is a group of zero or more statements with balanced braces and can be used anywhere a single statement is allowed.
* **Control Flow Statements**
  + Control flow statements however breakup the flow of execution by employing decision making, looping and branching statements.
  + Enabling your program to conditionally execute the particular block of code.
  + Decision making statements – if –then, if-then-else, switch
  + Looping statements – for , while, do-while
  + Branching statements – break, continue and return.
* **The if-then Statement** 
  + Execute a certain section of code, only if a particular test evaluates to true

void applyBrakes() {

// the "if" clause: bicycle must be moving

if (isMoving){

// the "then" clause: decrease current speed

currentSpeed--;

}

* **The if-then-else statement**
  + - The if-then-else statement provides the secondary path of execution, when an if clause evaluates to false.
* **Switch Statement**
  + The switch statement can have number of possible execution paths.
  + A switch works with byte, short, char and int primitive data types.
  + It also works with enumerated types and string class.
  + Body of the switch statement is known as switch block.
  + A statement in the switch block can be labeled with one or more case or default labels.
  + Each break statement terminates the enclosing switch statement.
  + The default section handles all the values that are not explicitly handled by any one of the section.
* **The While and Do-While Statements**
  + The while statement continuously executes a block of statements while a particular condition is true.
  + Its syntax can be expressed as

*While (expression) {*

*Statement(s)*

*}*

* + We can implement infinite loop using while statements.
  + The difference between do-while and while statement is that do-while evaluates its expression at the bottom of the loop instead of top.
  + The statements within the do block are always executed at least once.

*do {*

*Statement(s)*

*} while (expression)*

* **The for Statements**
  + The for statement provide the compact way to interact over a range of values.

*for (initialization, termination, increment) {*

*Statement(s)*

*}*

* + - The initialization expression initializes the loop; it’s executed once as the loop begins.
    - When the termination expression evaluates to false, the loop terminates.
    - The increment expression is invoked after each iteration through the loop.
    - Three expressions for the for-loop are optional.
    - The for statement has also another form designed to iteration over Collections and arrays.

for (int item: numbers) {

}

* + - The variable item holds the current value from numbers array.
* **Branching Statements**
  + **The Break Statement**
    - The break statement has two forms
      * Labeled
      * Unlabeled
        + Labeled break to terminate a for, while or do-while loop.
        + Unlabelled break to terminate an outer statement.
  + **The Continue Statement**
    - The continue statement skips the current iteration of a for, while or do-while loop.
  + **The Return Statement**
    - The return statement exits from current method and control flow, returns to where the method was invoked.
    - The return statement has two forms
      * One that returns a value
      * Another one that does not.
    - Data type of the returned value type of the methods declared value.
    - When the method is declared void, use the form of return, that does not return a value.