Java Fundamentals: The Java Language

Java is a programming language and also a runtime environment.

**Programming language:**

The Syntax, the Data types, Control flow like Looping and Conditional statements and being Object- orientated, right? Doing Inheritance

**Runtime:**

Configuration, Security a Threading Model, Input/output

# JDK VS JRE

# JRE (Java Runtime Environment)

Required to run Java apps

End users normally install JRE only.

# JDK (Java Development Kit)

Provides tools required to create Java apps.

Developers normally require the JDK.

But to run we need JRE, so JDK installation normally includes JRE.

# Creating and Running Java Apps

xyz.java --> jdk tools --> java app (platform-independent byte codes) --> JRE --> Host environment

Using the Java Development Kit we can feed our Source Code into it. And it will produce our Java application. Now Java's not like a low level language like C. C, when you compile a C program it produces an application that can run directly on the Host computer. Java uses an abstraction called Blake Codes that's platform independent. It allows us to not be tied to a particular Host Environment but actually have something that can be run in different Host Environments. And that's where the Java Runtime Environment comes in. The Java Runtime Environment provides what we need in order for our Java app that does Blake Codes to execute in any Host Environment. That's why End-users install the Java Runtime Environment because they just need to run that code. We as developers install the JDK to give us the tools to produce that application.

# To run a program without IDE

1. Go to JRE folder that is required run java.

E.g.: jre1.8.0\_51

1. In folder go to bin where we will have java.exe --> copy this folder address
2. Go and give this as a property in environmental variables with name path.
3. go to folder with main.class and do
4. java main
5. to run it.
6. you can check java version with java -version

# Whitespaces

These are ignored

# Comments

These statements are hided from compiler

Can be used to add human readable notes to the code.

There are 3 types

Single line comments: //

Block comments: /\*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\*/

Javadoc comments: /\*\*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\*/

We can put line comments inside block comments, but not block comments in a block comment.

# Packages

* provides organization
* follow standard naming
* affect source code file structure
* all lower case
* use reverse domain name to ensure uniqueness

E.g.: pluralsight.com 🡪 package com.pluralsight.mypackage

# Variables, Data Types, and Math Operators

## Variables:

Named data storage

Strongly typed

I.e. when we declare a variable we must specify type and it will store only that type

Value can be modified

### Naming Variables:

* Rules allow us to use letters, numbers, $ and \_
* By conventions only letters and numbers are used.
* Rules require that first character is not a number
* By convention it is always a letter
* By convention follow the style often referred to as "Camel Case"
* First letter is lower case
* Start of each word after the first is upper case
* All other letters are lower case
* Example: bankAccountBalance, level2Training

public class Main {

public static void main (String [] args) {

**Int** myVar;

//Gives error, we have to initialize, variables can be initialized in any part of the program

System.out.println (myVar);

}

}

public class Main {

public static void main (String [] args) {

int myVar = 50;

int secondVar = myVar;

myVar = 100;

}

}

Here the value of myVar is 100 and secondVar remains 50, because it is copy by value.

## Primitive Data types

* Built into the language.
* Foundation of all other types.
* We have four categories
* Integer
* Floating point
* Character
* Boolean

### Integer Types

There are 4 types based on storage space they take up which effects the range of values that can be stored in.

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Size | Range of values that can be stored | Literal Format |
| byte | 1 byte | −128 to 127 | 0 |
| short | 2 bytes | −32768 to 32767 | 0 |
| int | 4 bytes | −2,147,483,648 to 2,147,483,647 | 0 |
| long | 8 bytes | 9,223,372,036,854,775,808 to 9,223,372,036,854,755,807 | 0L |

### Floating point types

* Implementation of IEEE 754 floating point standard.
* Stores values containing a fractional portion.
* Supports positive, negative and zero values.

For more details

<http://bit.ly/psjavafp>

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Size(bits) | Smallest Positive Value | Largest Positive Value | Literal Format |
| Float | 32 | 1.4 x 10-45 | 3.4 x 1038 | 0.0f |
| Double | 64 | 4.9 x 10-324 | 1.7 x 10308 | 0.0 or 0.0d |

float milesInAMarathon = 26.2f;

double atomWidthInMeters = 0.000000000000002d;

### Character Types

The char type stores a single Unicode value.

Literal values are placed between single quotes

char smallU = ‘u’;

char accentedU = ‘\u00DA’;

### Boolean Types

It stores true and false

boolean iLoveJava = true;

### Primitive Types are Stored By-value

Java primitive types are stored by value. And that's important to understand because it affects the behavior of applications when we assign values from one variable to another. Let's take a look here. If I declare an integral variable called firstValue and I assign the value 100 to it, what's actually happening under the covers is that an area of memory is being allocated. It's named firstValue. And the value 100 is stored inside that memory. So now if I declare another variable, in this case otherValue and I assign firstValue to it, again I'm allocating an area of storage. I'm giving it a name, otherValue. And when I make the assignment, the value 100 is being copied from firstValue into other value. And what that means is that there's a distinctly separate copy of that value 100 being moved over into other value. What that allows me to do, then, is that if I make modifications to firstValue, for example, assign the value of 50 to it, that 50 replaces what's in firstValue, but it leaves the variable called otherValue completely untouched. And that's what I mean by being stored by value. So each primitive type variable has its own separate copy of the data.

## Arithmetic operators

#### Basic operators

Common arithmetic operators like +, /,\*, - and %.

#### Prefix/postfix operators

++ Increments value by 1

-- Decrements value by 1

int myVar = 1;

System.out.println (++myVar); //output:2

System.out.println (myVar); //output:2

System.out.println (myVar++); //output:2 but the value of myVar changes to 3

#### Compound assignment operators

Combines an operation and assignment.

Applies result of right side to the left side .

Stores that result in variable on left side.

int myVal= 50;

myVal -= 5;

System.out.println (myVal); //output:45

Available for 5 basic math operators

+=, -=, \*=, %= and /=

## Operator precedence

Operators are evaluated in a well-defined order

|  |  |  |
| --- | --- | --- |
| 1 | Postfix | x++, x-- |
| 2 | Prefix | ++x, --x |
| 3 | Multiplicative | \* / % |
| 4 | Additive | + - |

Operators of equal precedence are evaluated left-to-right.

You can override precedence with parenthesis.

Nested parenthesis evaluated from the inside out.

Example:

int valA = 21;

int valB = 6;

int valC = 3;

int valD = 1;

int result1 = valA – valB / valC; //output:19

int result2 = (valA – valB) / valC; //output:5

int result3 = valA / valC \* valD + valB; //output:13

//order for result 3 is (((valA / valC) 🡪7 \* valD) 🡪7 + valB) 🡪13

int result4 = valA / (valC \* (valD + valB)); //output:1

## Type Conversions

Because of multiple data types we need to convert types in some scenarios.

### Implicit type conversions

Conversions performed automatically by the compiler

#### E.g.:

int ival = 50;

long lval = ival;

Widening conversions are automatic (i.e. 32 bit to 64 bit)

The data is safe

|  |  |
| --- | --- |
| Mixed integer sizes | Uses largest integer in equation |
| Mixed floating point sizes | Uses double |
| Mixed integer and floating point | Uses largest floating point in the equation |

### Explicit type conversions

Conversions performed explicitly in code with cast operator.

#### E.g.:

**long** lval = 50;

**int** ival = (long) lval;

Can perform widening and narrowing

Floating to integer drops fraction

Use caution with narrow conversions

Integer to floating point can lose precision.

For more details go to 🡪 <http://bit.ly/pstypeconversion> . Check out **widening primitive conversions** and **narrowing primitive conversions**.

### Demo code

**float** floatVal = 1.0f;

**double** doubleVal = 4.0d;

**byte** byteVal = 7;

**short** shortVal = 7;

**long** longVal = 5;

**short** result1 = byteVal; 🡪 valid , widening conversion

**short** result1 = longVal; 🡪 invalid , narrowing conversion

**short** result1 = (**short**) longVal; 🡪 valid, explicit conversion

**short** result2 = byteVal – longVal; 🡪 invalid; cannot convert long to short

**short** result2 = (**short**) **(**byteVal – longVal); 🡪 invalid

**long** result3 = longVal – floatVal;🡪 invalid, lossy conversion

# Conditional Logic, Looping, and Arrays

## Adding Conditional Logic

1. Relational operators
2. Conditional assignment
3. The if statement
4. Logical operators

### Relational operators

>, >=, <, <=, ==, !=

### Conditional assignment

Assigns a value based on the result of a condition

Result = condition? True-value: false-value;

### If-else statement

An if-else statement conditionally executes a statement rather than just returning a value.

If (condition)   
 True-statement;

Else

False-statement;

The optional else executes if the condition is false.

#### Chaining if-else statements

If-else statements chained together are evaluated in order until one is true.

If (condition-1)   
 True-statement-1;

Else If (condition-2)   
 True-statement-2;

.

.

.

.

.

Else If (condition-N)   
 True-statement-N;

Else

False-statement;

#### Block statements

A block statement groups statements into a compound statement.

{

Statement-1;

Statement-2;

Statement-3;

.

.

.

Statement-4;

}

##### Demo Code

**int** v1=10, v2=4, diff; //we are declaring 3 variables on same line and initializing two of them

**if** (v1>v2)

{

diff = v1 – v2;

System.out.println(“v1 is bigger”);

System.out.println(diff);

}

**else if** (v2>v1)

{

diff = v2 – v1;

System.out.println (“v2 is bigger”);

System.out.println (diff);

}

**else**

System.out.println (“v1 and v2 are equal”);

##### Block Statements and Variable scope

A variable declared within a block is not visible outside of the block.

A variables range of visibility is known as the variable’s scope.

### Logical operators

|  |  |  |
| --- | --- | --- |
|  | Operator | What resolves to True |
| And | & | All must be true |
| Or | | | Any true |
| Exclusive or (XOR) | ^ | One false and one true |
| Negation | ! | false |

#### Conditional Logical operators

|  |  |  |
| --- | --- | --- |
|  | Operator | What resolves to True |
| Conditional and | && | All must be true |
| Conditional or | || | Any true |

Resolves following conceptually similar rules as non-conditional and/or.

Only executes right side if needed to determine the result

* + && only executes right side if left side is true.
  + || only executes right side if left side is false.

###### Demo Code

**int** students = 150;

**int** rooms = 0;

//this throws an exception when rooms=0, so we need to add a check for it.

//**if** (students/rooms > 30)

//it doesn’t even execute & students/rooms > 30 when rooms = 0, since rooms > 0 goes false

**if** (rooms > 0 & students/rooms > 30)

System.out.println (“Crowded!”);

System.out.println ();

System.out.println (“\*\*\* end program \*\*\*”);