JAVA SYNTAX

Rapid Learning & Just In Time Support

CONTENT

1 INTRODUCTION 5

1.1 About JAVA 6

1.2 Install 7

1.2.1 Automatically 8

1.2.2 Manually 9

1.3 Create Application 10

1.3.1 Console 11

1.3.2 Applet 12

1.4 Support 13

1.4.1 javac.exe 14

1.4.2 java.exe 16

2 BASIC SYNTAX 18

2.1 Comments 19

2.1.1 Single Line 20

2.1.2 Multi Line 21

2.2 Operators 22

2.2.1 Comparison 23

2.2.2 Arithmetic 25

2.2.3 Assignment 26

2.2.4 Bitwise 27

2.2.5 Logical 28

2.3 Statements 29

2.3.1 Conditional 29

2.3.1.1 Branching 30

2.3.1.1.1 if 31

2.3.1.1.2 else 31

2.3.1.1.3 else if 32

2.3.1.1.4 switch 33

2.3.1.1.5 ? : 34

2.3.1.2 Looping 35

2.3.1.2.1 for 36

2.3.1.2.2 while 37

2.3.1.2.3 do ... while 38

2.3.2 Jumping 39

2.3.2.1 break 40

2.3.2.2 continue 41

2.4 Data types 42

2.4.1 Scalar 46

2.4.1.1 null 47

2.4.1.2 bool 48

2.4.1.3 byte 49

2.4.1.4 short 50

2.4.1.5 int 51

2.4.1.6 long 52

2.4.1.7 float 53

2.4.1.8 double 54

2.4.1.9 char 55
2.4.2 Compound 56
2.4.2.1 String 57
2.4.2.2 array 58
2.4.2.3 Class 59
2.5 Literals 60
2.5.1 Null 61
2.5.2 Boolean 62
2.5.3 Integer 63
2.5.3.1 Decimal Notation 64
2.5.3.2 Octal Notation 65
2.5.3.3 Hexadecimal Notation 66
2.5.4 Long 67
2.5.4.1 Decimal Notation 68
2.5.4.2 Octal Notation 69
2.5.4.3 Hexadecimal Notation 70
2.5.5 Float 71
2.5.5.1 Basic Notation 72
2.5.5.2 Scientific Notation 73
2.5.6 Double 74
2.5.6.1 Basic Notation 75
2.5.6.2 Scientific Notation 76
2.5.7 Character 77
2.5.8 String 78
2.5.9 Array 79
3 OBJECT ORIENTED SYNTAX 80
3.1 Classes 81
3.1.1 Inheritance 83
3.1.2 Reflection 84
3.1.3 ClassLoader 86
3.2 Interface 87
3.3 Objects 88
3.3.1 Create 89
3.3.2 Upcasting 90
3.3.3 Downcasting 91
3.4 Exceptions 92
3.5 Methods 93
3.5.1 Create 94
3.5.2 Parameters Scope 95
3.5.3 Overloading 97
3.5.4 Overriding 98
4 ADVANCED SYNTAX 99
4.1 Generic 99
4.1.1 Class 100
4.1.2 Method 102
4.1.3 Type inference 103
4.1.4 Type parameters 103
4.1.5 Wildcards 104

5 BUILT IN FUNCTIONALITIES 107

5.1 Collections 107

5.1.1 HashMap 108
5.1.2 Vectors 109
5.1.3 Iterator 110
5.1.4 Hashtables 111
5.1.5 Enumerator 113
5.1.6 Date/Time 114

5.2 Serialization 116

6 APPENDIX 116

6.1 ASCII Table 117

6.2 Cheat Sheets 119

6.2.1 bool 120
6.2.2 byte 121
6.2.3 short 123
6.2.4 int 125
6.2.5 long 127
6.2.6 float 129
6.2.7 double 130
6.2.8 char 131
6.2.9 String 132
6.2.10 array 134
6.2.10.1 Single dimensional 134
6.2.10.2 Multi Dimensional 136

1 Introduction

Info

- This book contains tutorials about syntax of JAVA Programming Language.
  Syntax is the set of rules that you must obey in order to write proper code in specified language.
- Tutorials are standalone and minimalistic focusing only at one problem at the time.
  Each tutorial includes working example with a complete code needed to test discussed functionality.

How to use this book

- This book can be used as an introduction to JAVA Programming Language covering all of the basic core functionalities.
- Book is also intended as Just In Time Support so that user can learn what it needs when it needs it.
- This is why tutorials are standalone and minimalistic focusing only at one problem at the time.

Meaning of symbols used in tutorials

- In most tutorials you will find symbols like [R], [E] and [C] whose meanings are described in following table.

Symbols

SYMBOL	NAME	DESCRIPTION
[R]	Reference	Link to reference which was used while creating tutorial.
[E]	Error	Link to solution for an error that might occur while following tutorial.
[C]	Create application	Link to a tutorial which shows how to create an application in order to test tutorial's code.

Chapters organization

Chapter 1

- First chapter contains short introduction to JAVA Syntax followed by tutorials that explain how to install JAVA.
- Chapter contains tutorials which show how to create either JAVA Console or Applet Application.

Chapter 2

- Second chapter contains tutorials explaining basic JAVA syntax which doesn't include rules related to classes and objects.

Chapter 3

- Third chapter contains tutorials that explain JAVA syntax related to classes and objects.

Chapter 4

- Forth chapter contains tutorials that explain advanced parts of JAVA syntax some of which were added to later versions.

Chapter 5

- Fifth chapter contains tutorials that explain how to use built in that functions which give you more power of what you can do with the language compared to using just syntax.

Chapter 6

- Sixth chapter describes different tools that might prove helpful while working with that like Eclipse IDE.

Chapter 7

- Sixth chapter contains list of errors that might occur while working with PHP and how to solve them.

Chapter 8

- Last chapter contains additional informations related to JAVA like cheat sheets containing most useful code snippets.

1.1 About JAVA

Info

- JAVA
  - is Object Oriented Programming Language (OOP)
  - is platform independent which means that it can run on any OS-Operating System that supports JVM.
    JVM stands for Java Virtual Machine which is capable of running JAVA intermediate code
  - has no pointers and the job of removing objects that are not needed from memory is taken care by Garbage Collector.
    Since it has no pointers direct access to memory is not impossible.
- JAVA's goal was to bring order in the world of programming languages ruled thus far by C, C++ an PERL.
  All the tasks that could be done by computer were transferred from programmer to computer.
  This is why JAVA has no headers since it is smart enough, unlike C compilers, to find function definition on its own.
- Java Development Kit (JDK) is collection of files and applications needed to create JAVA applications.
- Java Runtime Environment (JRE) is collection of files and applications needed to run JAVA applications.

1.2 Install

Info

- Following tutorials show how to install JAVA.

1.2.1 Automatically

Info

- This tutorial shows how to install JAVA by using installer program.
- Environment variables will be automatically set.

Download

- http://java.sun.com/javase/downloads/index.jsp
- JDK 6 Update 12: Download
- Platform: Windows
- I Agree: ON
- Continue
- jdk-6u12-windows-i586-p.exe
- Save to: D:\Downloads\jdk-6u12-windows-i586-p.exe

Install

- Start jdk-6u12-windows-i586-p.exe

Test

- Create JAVA Console Application

1.2.2 Manually

Info

- This tutorial shows how to install JAVA by copying JAVA installation and setting environment variables manually.

Extract

- Extract zipped JDK to D:\Installed\Programming

Edit Environment Variables

- Start
- Settings
- Control Panel
- System
- Advanced
- Environment Variables
- System Variables

Environment Variables

EXAMPLE		DESCRIPTION
PATH	;D:\Installed\Programming\JDK6\bin	Allows you to execute java.exe from any directory while in MSDOS.
CLASSPATH	;D:\Installed\Programming\JDK6\bin	Allows javac.exe to find .class files listed in this variable.
JAVA_HOME	;D:\Installed\Programming\JDK6	Some programs, like Eclipse, use it to find installed JAVA.

Test

- Create JAVA Console Application

1.3 Create Application

Info

- Following tutorials show different ways of creating JAVA application.

1.3.1 Console

Info

- This tutorial shows how to create "JAVA Console Application".
- Console application uses command line to interact with the user.

Example

- Create D:\Temp\Test.java
- Start MSDOS
- cd D:\Temp
- javac Test.java

Test.java

```
public class Test {

  public static void main(String[] args) {
    System.out.println("Hello World.");
  }

}
```

Test

- Start MSDOS
- cd D:\Temp
- java Test

output

```
Hello World.
```

1.3.2 Applet

Info

- This tutorial shows how to create Applet.
- Applet is JAVA program meant to run under Web Browser.

Example

- Create JAVA Console Application Test.java
- Create Web Page D:\Temp\Test.html

Test.java

```
import java.applet.*;
import java.awt.*;

public class Test extends Applet {
  public void paint(Graphics g) {
    g.drawString("Hello world!", 50, 25);
  }
}
```

Test.html

```
<html>
  <body>
    Here is the output of my program:
    <applet code="Test.class" width="150" height="25" alt=""></applet>
  </body>
</html>
```

Test

- D:\Temp\Test.html

1.4 Support

Info

- Following tutorials show how to use javac.exe and java.exe to compile and run JAVA code.

1.4.1 javac.exe

Info

- javac.exe is application used to compile java code .java into intermediate code .class.

Examples

COMMAND	DESCRIPTION
`javac Test.java`	Compile Test.java from current dir and save Test.class in the same dir.
`javac *.java`	Compile all files with .java suffix from current dir and save their .class files in the same dir.

-sourcepath

- Defines where to find source files needed to compile target source file.
- You can define directories, JAR or ZIP files.
- Needed source files are also compiled and their .class files are saved in the same directory where they were found.

Examples

COMMAND	DESCRIPTION
`javac -sourcepath src Test2.java`	Look for source files in src dir in current directory.
`javac -sourcepath .;..\src;C:\stuff Test2.java`	Look for source files in listed directories.
`javac -sourcepath servlet.jar;C:\util.zip Test2.java`	Look for source files in listed directories.

-classpath, -cp

- Defines where to look for class files needed to compile target source file.
- You can define directories, JAR or ZIP files.

Examples

COMMAND	DESCRIPTION
`javac -classpath classes Test2.java`	Look for class files in class dir in current directory.
`javac -classpath .;..\classes;C:\stuff Test2.java`	Look for class files in listed directories.
`javac -cp servlet.jar;C:\util.zip Test2.java`	Look for class files in listed dirs, JARs or ZIP files.

-d

- Without -d parameter javac creates .class files in current directory ignoring package directive.
- Only if -d parameter is used javac will create directory structure as defined with package directive.
- Directory structure will be created in directory defined with -d parameter.
- .class file will be created in the last directory.

Examples

COMMAND	DESCRIPTION
`javac Test.java`	Create Test.class in current directory ignoring package directive.
`javac -d . Test.java`	Create directory structure in current directory.
`javac -d C:\stuff Test.java`	Create directory structure in directory C:\stuff.
`javac -d classes Test.java`	Create directory structure in directory classes in current directory.
`javac -d temp\classes Test.java`	Create directory structure in directory temp\classes from current directory.
`javac -d ..\ Test.java`	Create directory structure in parent directory from current directory.
`javac -d ..\..\classes Test.java`	Create directory structure in directory classes two dirs above current directory.

-version

- Displays JAVA version.
- **java -version**

output

```
java version "1.5.0"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-b64)
Java HotSpot(TM) Client VM (build 1.5.0-b64, mixed mode)
```

-help

- Displays possible parameters and their usage.
- **javac -help**

output

```
Usage: javac <options> <source files>
where possible options include:
  -g                         Generate all debugging info
  -g:none                    Generate no debugging info
  -g:{lines,vars,source}     Generate only some debugging info
  -nowarn                    Generate no warnings
  -verbose                   Output messages about what the compiler is doing
  -deprecation               Output source locations where deprecated APIs are used
  -classpath <path>          Specify where to find user class files
  -cp <path>                 Specify where to find user class files
  -sourcepath <path>         Specify where to find input source files
  -bootclasspath <path>      Override location of bootstrap class files
  -extdirs <dirs>            Override location of installed extensions
  -endorseddirs <dirs>       Override location of endorsed standards path
  -d <directory>             Specify where to place generated class files
  -encoding <encoding>       Specify character encoding used by source files
  -source <release>          Provide source compatibility with specified release
  -target <release>          Generate class files for specific VM version
  -version                   Version information
  -help                      Print a synopsis of standard options
  -X                         Print a synopsis of nonstandard options
  -J<flag>                   Pass <flag> directly to the runtime system
```

Example

- This example can be used to test different parameters of javac.exe.
- Create JAVA Console Application Test.java

Test.java

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Hello World.");
  }
}
```

Test2.java

```
public class Test2 {
  public static void main(String[] args) {
    Test test = new Test();
         test.main(null);
  }
}
```

1.4.2 java.exe

Info

- java.exe is used to start JVM and run .class files in it.

Run .class which was made using package keyword

- MyJava.java has line: package begin.middle;
- Save MyJava.class in: C:\Stuff\src\begin\middle
- Start MSDOS
- cd C:\Stuff\src>java begin.middle.MyJava

Define VM version to run .class

- C:\MyClasses>C:\Install\JDK14\bin\java MyClasses

Define paths to classes

- Following line defines two class paths which are used to find MyProgram.class and all classes used by it. MyProgram.class is in directory Tests and it has no package defined in it.

```
java -cp .;D:\JavaProjects\Jemmy\Tests;D:\ProgramFiles\JemmyClasses MyProgram
```

- Following line defines that Java will look for needed classes in current directory and in C:\Stuff\Some\JDBCDriver_Oracle9.zip. Class TestSome is called with parameter 3.416

```
java -cp .;C:\Stuff\Sinisa\JDBCDriver_Oracle9.zip TestSome 3.416
```

```
java -cp .;..\lib\httpunit.jar;..\lib\js.jar; TestHTTPUnit videobox 7n433vfs
```

Define output

- Output of program will be written to the file
- java Hello > ivor.log

Get Version

- java -version

Errors

'java' is not recognized as an internal or external command, operable program or batch file.

- You have to check Environment Variable PATH. If it looks like this without quotes "C:\Install\JDK142\bin;D:\Stuff" DOS will try to find java.exe first in C:\Install\JDK142\bin directory and then in D:\Stuff directory. This means that first java.exe found is executed. Be aware that DOS only looks for java.exe in listed directories but not in their subdirectories.
- Be aware that there are two types of PATH variable, one defined as User Variable and the other as System Variable. If User Variable PATH exists java uses ONLY its value. If it doesn't exists, but System Variable PATH exists, then it uses ONLY its value. If both variables are missing DOS will not be able to find java.exe.
- In order for changes in Environment Variables to become visible in DOS you have to restart DOS.

Exception in thread "main" java.lang.NoClassDefFoundError: Ivor

- You have to check Environment Variable CLASSPATH. If it looks like this without quotes ".;D:\MyClasses" it means that java first looks for classes in current directory and then in D:\myClasses directory. If it can not find the class in any of those it reports above error. Be aware that DOS only looks for java.exe in listed directories but not in their subdirectories.
- Be aware that there are two types of CLASSPATH variable, one defined as User Variable and the other as System Variable. If User Variable CLASSPATH exists java uses ONLY its value. If it doesn't exists, but System Variable CLASSPATH exists, then it uses ONLY its value. If both variables are missing java will by default look only in current directory.
- In order for changes in Environment Variables to become visible in DOS you have to restart DOS.

Moj.jar access denied

- Set attribute file-a to executable.

Moj.jar cannot execute binary file

- Class path is not valid. Try to defined it like

```
java -cp ".:../dir1/Moj.jar:../dir2/Moj.jar MojProgram"
```

2 Basic Syntax

Info

- Following tutorials explain basic JAVA syntax.
- Later chapter Object Oriented Syntax explains syntax related to classes and objects.
- Syntax of a programming language is the set of rules that define the combinations of symbols that are considered to be correctly structured programs in that language.

2.1 Comments

Info [C]

- Comments are pieces of text inserted into source code to clarify it.
- They are not instructions for the computer and are therefore ignored by execution engine.
- Specially formatted comments can be used to generate documentation directly from the source code.
- JAVA supports both single line and multi line (block) comments.

Comment types

SINGLE LINE COMMENT TYPE	EXAMPLE
Single line	`//Single line comment`

Multi line	`/* Multi line comment.`
	`  Second line.`
	`*/`

2.1.1 Single Line

Info [C]

- Single Line comments can't span over multiple lines.
- Start of single line comment is indicated by especially reserved combination of characters `//`.
- End of single line comment is indicated by the end of the line.
- Benefit of single line comments is that they do not need to be terminated with terminator keyword.
- Downside of single line comment is that creating longer comments requires placing indicator for start of comment at the beginning of each line.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //Print something.
    //Say Hello World.
    System.out.println("Hello World");  //Single line comment.

  }

}
```

2.1.2 Multi Line

Info [C]

- Multi Line comments can span over multiple lines.
- Start of multi-line comment is indicated by combination of slash and multiply character `/*`.
- End of multi-line comment is indicated by combination of multiply and slash characters `*/`.
- Benefit of multi-line comments is that they allow you to write a lot of text that spans over multiple lines without having to prefix each line with predefined prefixes as is the case with single line comments.
- Downside of multi-line comments is that they have to be terminated with specific indicator which presents typing overhead compared to single line comments for shorter text.

Test.java

```
public class Test {

  public static void main(String[] args) {

    /* Multi line comment.
       Block comment.      */
    System.out.println("Hello World");

  }

}
```

2.2 Operators

Info

- Operators are used for changing values of variables.

2.2.1 Comparison

Info

- Comparison operators are used to compare values of two variables.
- This way they provide a method to direct program flow depending on the outcome.
- JAVA comparison operators should be used only for comparing numerical values.
- Strings should be compared using predefined JAVA functions designed for that purpose.

Operators

TYPE	NAME	DESCRIPTION
`a == b`	Equal	TRUE if 'a' is equal to 'b' after type juggling
`a != b`	Not equal	TRUE if 'a' is not equal to 'b' after type juggling.
`a <> b`	Not equal	TRUE if 'a' is not equal to 'b' after type juggling.
`a === b`	Identical	TRUE if 'a' is of the same value and type as 'b'.
`a !== b`	Not identical	TRUE if 'a' is not of the same value and type as 'b'.
`a < b`	Less than	TRUE if 'a' is strictly less than 'b'.
`a > b`	Greater than	TRUE if 'a' is strictly greater than 'b'.
`a <= b`	Less than or equal	TRUE if 'a' is less than or equal to 'b'.
`a >= b`	Greater than or equal	TRUE if 'a' is greater than or equal to 'b'.

Numerical values [C]

- This example shows how to use comparison operators on numerical values.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //FOR char, byte, short, int, long, float, double.-------------------------------------------------
    char left  = 65;
    long right = 70;
    if (left == right) { System.out.println("Left    equals           right."); }
    if (left != right) { System.out.println("Left is different from    right."); }
    if (left <  right) { System.out.println("Left is smaller then     right."); }
    if (left <= right) { System.out.println("Left is smaller or equals right."); }
    if (left >  right) { System.out.println("Left is greater then     right."); }
    if (left >= right) { System.out.println("Left is greater or equals right."); }

    //FOR boolean.-------------------------------------------------------------------------------------
    boolean l = true;
    boolean r = false;
    if (l == r)        { System.out.println("Left    equals         right."); }
    if (l != r)        { System.out.println("Left is different from right."); }

  }

}
```

Objects and Strings [C]

- This example shows to compare Objects and String for which you can't use comparison operators.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //FOR Objects.-------------------------------------------------------------------------------
    String text = "Hello";
    if (text instanceof String)  { System.out.println("Object is of class String."); }

    //FOR Strings.-------------------------------------------------------------------------------
    String text = "Hello";
    if (text.equals          ("Hello")) { System.out.println("text is equal to \"Hello\"."         ); }
    if (text.equalsIgnoreCase("hello")) { System.out.println("text is equal to \"hello\"."         ); }
    if (text.contains        ("llo"  )) { System.out.println("text contains \"llo\"."              ); }
    if (text.contentEquals   ("Hello")) { System.out.println("text is equal to \"Hello\"."         ); }
    if (text.startsWith      ("He"   )) { System.out.println("text starts with \"He\"."            ); }
    if (text.startsWith      ("llo",2)) { System.out.println("text at index 2 starts with \"llo\".."); }
    if (text.endsWith        ("llo"  )) { System.out.println("text ends with \"llo\"."              ); }
    if (text.matches         ("Hello")) { System.out.println("text matches regex \"Hello\"."       ); }

  }

}
```

2.2.2 Arithmetic

Info [C]

- Arithmetic operators are used to perform mathematical operations.
- Mathematical operations, not supported by arithmetic operators, can be performed using JAVA mathematical functions.
- If one of the operands is float or double the other one is implicitly transformed into that type.
- **Division** returns integer if both operands are evenly divisible integers (or strings that get converted to integers)
  In all other cases float is returned.
- **Modulus** operands are converted to integers (by stripping the decimal part) before processing.
  Result of the modulus has the same sign as the dividend (first parameter a).

Operators

TYPE	NAME
+ b	Plus
- b	Negation
a + b	Addition
a - b	Subtraction
a * b	Multiplication
a / b	Division.
a % b	Modulus.
a++	Post-increment
a--	Post-decrement
++b	Pre-increment
--b	Pre-decrement

Test.java

```
public class Test {

  public static void main(String[] args) {

    //TEST VARIABLES.
    int     x          = 10;
    int     y          = 20;

    //ARITHMETIC OPERATORS.
    int     negate     =  -y;        //-10       = -10
    int     add        = x+y;        //10+20     =  30
    int     subtract   = x-y;        //10-20     = -10
    int     multiply   = x*y;        //10*20     =  200
    int     divide1    = x/y;        //10/20     =  0.5     = 0
    float   divide2    = (float)x/y; //10.0/20.0 =  0.5
    int     reminder   = x%y;        //10%20     =  0*20+10 = 10
    int     increment1 = ++x;        //Increment x by 1 and then store x into increment1.
    int     decrement1 = --x;        //Decrement x by 1 and then store x into decrement1.
    int     increment2 =   x++;      //Store x into increment2 and then increment x by 1.
    int     decrement2 =   x--;      //Store x into decrement2 and then decrement x by 1.

    //DISPLAY RESULTS.
    System.out.println(negate);

  }

}
```

2.2.3 Assignment

Info [C]

- Assignment operators assign value to variable.
- Shortcut assignment operators like `+=` or `*=` perform mathematical operation before making the assignment.

Operators

TYPE	NAME	DESCRIPTION
`a = 5`	Assignment	a equals 5
`a += 5`	Addition-Assignment	a equals a plus 5
`a -= 5`	Subtraction-Assignment	a equals a minus 5
`a *= 5`	Multiplication-Assignment	a equals a multiplied by 5
`a /= 5`	Division-Assignment	a equals a divided by 5
`a %= 5`	Modulus-Assignment	a equals a modulo 5

Test.java

```
public class Test {

 public static void main(String[] args) {

   //TEST VARIABLES.
   int x = 10;

   //ARITHMETIC OPERATORS.
   x += 10; // x = x + 10;
   x -= 10; // x = x - 10;
   x *= 10; // x = x * 10;
   x /= 10; // x = x / 10;
   x %= 10; // x = x % 10;

   //DISPLAY RESULT.
   System.out.println(x);

 }

}
```

2.2.4 Bitwise

Info [C]

- Bitwise operators combine bits within one or two `int` or `long` values.
- If an operand is shorter than an `int`, it is promoted to `int` before doing the operations.
- Right shift `>> n` is Arithmetic Shift equivalent of dividing left operand by 2^n which preserves operand sign.
- AND, OR, XOR and NOT do not change the value of operands.

Operators

TYPE	NAME	DESCRIPTION
`$a & $b`	AND	Set bits that are set in both $a and $b
`$a | $b`	OR	Set bits that are set in either $a or $b.
`$a ^ $b`	XOR	Set bits that are different in $a and $b.
`~ $b`	NOT	Invert bits
`$a << $b`	Shift left	Shift the bits of $a by $b steps to the left (each step means "multiply by two")
`$a >> $b`	Shift right	Shift the bits of $a by $b steps to the right (each step means "divide by two")

Test.java

```
public class Test {

  public static void main(String[] args) {

    //TEST VARIABLES.
    int left        = Integer.parseInt ("11011", 2);
    int right       = Integer.parseInt ("10010", 2);

    //BITWISE OPERATORS.
    int and         =  left & right; //10010.    1 if both   bits are 1.
    int or          =  left | right; //11011.    1 if either bit  is  1.
    int xor         =  left ^ right; //01001.    1 if        bits are different.
    int invert      = ~left;         //00100.    Invert bits.
    int shiftLeft   =  left <<  3;   //11011000. Shift bits to the left  by 3. Fill with 0.
    int shiftRight  =  left >>  2;   //110.      Shift bits to the right by 2. Fill with 0.
    int shiftRight2 =  left >>> 2;   //????

    //DISPLAY RESULTS.
    System.out.println ( Integer.toBinaryString (and        ) );
    System.out.println ( Integer.toBinaryString (or         ) );
    System.out.println ( Integer.toBinaryString (xor        ) );
    System.out.println ( Integer.toBinaryString (invert     ) );
    System.out.println ( Integer.toBinaryString (shiftLeft  ) );
    System.out.println ( Integer.toBinaryString (shiftRight ) );
    System.out.println ( Integer.toBinaryString (shiftRight2) );

  }

}
```

2.2.5 Logical

Info [C]

- Logical operators are used to combine boolean values that can be only TRUE or FALSE.
- This way they provide a method to direct program flow depending on the outcome.

Operators

TYPE	NAME	DESCRIPTION
a && b	AND	True if both 'a' and 'b' are true
a AND b	AND	True if both 'a' and 'b' are true
a || b	OR	True if either 'a' or 'b' is true
a OR b	OR	True if either 'a' or 'b' is true
a XOR b	Exclusive OR	True if only 'a' or only 'b' is true
!b	NOT	True if $b is not true
NOT b	NOT	True if $b is not true

Test.java

```
public class Test {

  public static void main(String[] args) {

    //TEST VARIABLES.
    char left  = 65;
    long right = 70;

    //LOGICAL OPERATORS.
    if ( left> 50 && right==70 ) { System.out.println("left >50 AND right==10"    ); }
    if ( left!=4  || right> 90 ) { System.out.println("left!=50 OR  right> 90"    ); }
    if (          !  (left==80)) { System.out.println("left is  NOT equal to 80"  ); }

  }

}
```

2.3 Statements

Info

- Statement is smallest part of code which can be executed.
- JAVA statements must be terminated with semicolon `;` indicating end of statement, except in some special cases.
- Statements are classified depending on their function.

2.3.1 Conditional

Info

- Conditional Statements define which part of code should be executed depending on some condition.
- Code is executed only once compared to Looping Statements where code can be executed multiple times.

2.3.1.1 Branching

Info

- Branching is used to control program flow.
- Following example contains examples of branching options.

2.3.1.1.1 if

Info [C]

- `if` statement executes specified code if specified condition is true.
- If code contains only one statement curly brackets can be omitted but this can lead to errors when adding statements.

Syntax

```
if (condition) { code }
```

Test.java

```
public class Test {

  public static void main (String arg[]) throws Exception {

    int    value = 10;
    String text  = "bird";

    if (value == 10        ) { System.out.println("Value = 10"      ); }
    if (text.equals("bird")) { System.out.println("text equals bird"); }

  }

}
```

2.3.1.1.2 else

Info [C]

- `else` statement can be used only in combination with `if` statement.
- If condition specified by `if` statement is TRUE then only code defined in `if` statement is executed.
- If condition specified by `if` statement is FALSE then only code defined in `else` statement is executed.
- If code belonging to `if` or `else` statements contains only single statement, curly brackets can be omitted but this can lead to errors when adding statements.

Syntax

```
if   (condition) { code1 }
else             { code2 }
```

Test.java

```
public class Test {

  public static void main (String arg[]) throws Exception {

    int    value = 10;
    String text  = "bird";

    if      (value == 10        ) { System.out.println("Value = 10"      ); }
    else                          { System.out.println("No match found"  ); }

  }

}
```

2.3.1.1.3 else if

Info [C]

- Multiple `else if` statements can be used in combination with `if` statement.
- `if...elseif` combo works as multiple `if` statements.
- Conditions are evaluated in sequence and only code belonging to the first condition which evaluates to true is executed.
- `if...else if` combo can optionally end with `else` statement.
  If so, then if all conditions evaluate to FALSE only code belonging to `else` statement will be executed.

Syntax

```
if      (condition1) { code1  }
else if (condition2) { code2  }
...
else                 { code10 }
```

Test.java

```
public class Test {

  public static void main (String arg[]) throws Exception {

    int    value = 10;
    String text  = "bird";

    if      (value == 10        ) { System.out.println("Value = 10"      ); }
    else if (text.equals("bird")) { System.out.println("text equals bird"); }
    else if (text.equals("dog" )) { System.out.println("text equals dog" ); }
    else                          { System.out.println("No match found"  ); }

  }

}
```

2.3.1.1.4 switch

Info [C]

- `switch` statement uses single `integer` or `char` parameter whose value determines **from** which case to start executing code. All statements that follow are then executed until `break` statement or end of switch block.
- `default:` case is executed if no other cases were true and it can be placed anywhere.
- `switch` statement is used instead `else if` statements when all conditions evaluate the same integer variable.

Syntax

```
switch ( integer|char ) {
  case value1:
    optional code 1
    break;
  case value2:
    optional code 2
  ...
  default:
    optional code 10
}
```

Test.java

```
public class Test {

  public static void main (String arg[]) throws Exception {

    int a = 5;

    switch (a) {

      case 1:
        System.out.println("Value is 1. Exit from switch.");
        break;

      case 5:
        System.out.println("Value is 5. Continue to next case or default.");

      default:
        System.out.println("Default value is selected if no case was true.");
        break;
    }

  }

}
```

2.3.1.1.5 ? :

Info [C]

- This tutorial shows how to use `? :` conditional statement where
  - Expression left from `?` is evaluated.
  - If TRUE expression left from `:` is executed.
  - If FALSE expression right from `:` is executed.

Syntax

```
condition ? code if true : code if false;
```

Test.java

```
public class Test {

  public static void main(String[] args) {

    int a   = 10;
    int b   = 5;

    int max = (a > b) ? a : b;
    System.out.println(max);

  }

}
```

2.3.1.2 Looping

Info

- Depending on given condition Looping Statements can execute specific part of code multiple times.
- Existence of condition makes them similar to Conditional Statements but they can execute code only once.

2.3.1.2.1 for

Info [C]

- Depending on the condition, `for` statement can execute specified code multiple times where
  - expression1 is executed at the beginning of the first iteration
  - expression2 is executed at the end of each iteration
  - condition is evaluated at the beginning of each iteration
    - If it is TRUE code is executed
    - If it is FALSE code is NOT executed and program breaks out of for loop
- Expressions and condition can be left empty in which case condition evaluates to TRUE.
- You can use
  - `break` statement to break out of for loop
  - `continue` statement to skip the rest of the body and continue with next iteration from the beginning of for loop.

Syntax

```
for (expression1; condition; expression2) { body }
```

Test.java

```
public class Test {

  public static void main (String arg[]) throws Exception {

    //SIMPLE.   Output is "1 2 3 4".
    for(int i=1;i<=4;i++) {
      System.out.println(i);
    }

    //COMPLEX.  Output is "1 2 3".
    for(int i=1, j=5 ; i<=4 && j>2 ; i++,j--){
      System.out.println(i);
    }

    //BREAK.    Output is "1 2 3" since we break out of for loop before condition i<4 is reached.
    for(int i=1;i<=4;i++) {
      System.out.println(i);
      if(i==3) { break; }
    }

    //CONTINUE. Output is "1 2 4" since display of number 3 is skipped.
    for(int i=1;i<=4;i++) {
      if(i==3) { continue; };
      System.out.println(i);
    }

  }

}
```

2.3.1.2.2 while

Info [C]

- `while` statement iteratively executes specified code as long as condition is TRUE where
  - condition is evaluated at the beginning of each iteration.
    - If it is TRUE code is executed.
    - If it is FALSE code is NOT executed and program breaks out of while loop.
- You can use
  - `break` statement to break out of for loop
  - `continue` statement to skip the rest of the body and continue with next iteration from the beginning of for loop.

Syntax

```
while (condition) { code }
```

Test.java

```
public class Test {

  public static void main (String arg[]) throws Exception {

    //Initialize variable that will be used as counter.
    int i=0;

    //SIMPLE. Output is "1 2 3 4 5".
    while(i<=4) {
      i++;
      System.out.println(i);
    }

    //BREAK.  Output is "1 2" since we break out of for loop before condition i<=4 is reached.
    while(i<=4){
      i++;
      if(i==3) { break; }
      System.out.println(i);
    }

    //CONTINUE. Output is "1 2 4 5" since display of number 3 is skipped.
    while(i<=4){
      i++;
      if(i==3) { continue; }
      System.out.println(i);
    }

  }

}
```

2.3.1.2.3 do ... while

Info [C]

- `do...while` statement iteratively executes specified code as long as condition is TRUE where
  - condition is evaluated at the end of each iteration.
    - If it evaluates to TRUE code is executed.
    - If it evaluates to FALSE code is NOT executed and program breaks out of do...while loop.
- Unlike `while` statement, code is always executed at least once since `condition` is evaluated at the end of each iteration.
- You can use
  - `break` statement to break out of for loop
  - `continue` statement to skip the rest of the body and continue with next iteration from the beginning of for loop

Syntax

```
do { code } while ( condition )
```

Test.java

```
public class Test {

  public static void main (String arg[]) throws Exception {

    //Initialize variable that will be used as counter.
    int i=0;

    //SIMPLE. Output is "1 2 3 4 5".
    do{
      i++;
      System.out.println(i);
    }while(i<=4);

    //BREAK.  Output is "1 2" since we break out of for loop before condition i<=4 is reached
    do{
      i++;
      if(i==3) { break; }
      System.out.println(i);
    }while(i<=4);

    //CONTINUE. Output is "1 2 4 5" since display of number 3 is skipped.
    do{
      i++;
      if(i==3) { continue; }
      System.out.println(i);
    }while(i<=4);

  }

}
```

2.3.2 Jumping

Info

- Jumping statements are used to unconditionally continue code execution at some other part of code.
- Use of such statements is considered bad practice since they make it harder to follow code execution.
- You should use conditional statements instead.

2.3.2.1 break

Info [C]

- `break` statement is used to end execution of looping statements or switch blocks.
- This means that you can use it to stop execution of for, while, do ... while or switch blocks.

Syntax

```
break;
```

Test.java

```
public class Test {

  public static void main (String arg[]) throws Exception {

    //FOR.  Output is "1 2 3" since we break out of for loop before condition i<4 is reached.
    for(int i=1; i<=4; i++) {
      System.out.println(i);
      if(i==3) { break; }
    }

    //WHILE.  Output is "1 2" since we break out of for loop before condition i<=4 is reached.
    int i=0;
    while(i<=4){
      i++;
      if(i==3) { break; }
      System.out.println(i);
    }

    //DO...WHILE.  Output is "1 2" since we break out of for loop before condition i<=4 is reached
    i=0;
    do {
      i++;
      if(i==3) { break; }
      System.out.println(i);
    } while(i<=4);

    //SWITCH.
    int a = 5;
    switch (a) {
      case 1:
        System.out.println("Value is 1. Exit from switch.");
        break;
      case 5:
        System.out.println("Value is 5. Continue to next case or default.");
      default:
        System.out.println("Default value is selected if no case was true.");
        break;
    }

  }

}
```

2.3.2.2 continue

Info [C]

- `continue` statement is used inside looping statements to stop current iteration and continue with the next one.
- This means that you can use it to stop execution of for, while or do ... while blocks.

Syntax

```
continue;
```

Test.php

```
public class Test {

  public static void main (String arg[]) throws Exception {


    //FOR. Output is "1 2 4" since display of number 3 is skipped.
    for(int i=1; i<=4; i++) {
      if(i==3) { continue; };
      System.out.println(i);
    }

    //WHILE. Output is "1 2 4 5" since display of number 3 is skipped.
    int i=0;
    while(i<=4){
      i++;
      if(i==3) { continue; }
      System.out.println(i);
    }

    //Do...WHILE. Output is "1 2 4 5" since display of number 3 is skipped.
    do{
      i++;
      if(i==3) { continue; }
      System.out.println(i);
    }while(i<=4);

  }

}
```

2.4 Data types

Info [R]

- Data type defines main characteristics of data.

Categorization of Data Types

- Data types can be categorized as
  - Pre defined data types (boolean, char, byte, short, int, long, float, double, array and predefined classes like String)
  - User defined data types (user defined classes or interfaces)
- Data types can be categorized as
  - Value data types which hold data within its own memory allocation
  - Reference data types which hold reference within its own memory allocation that points to actual data
- Data types can be categorized as
  - Scalar data types which can store single value (boolean, char, byte, short, int, long, float, double)
  - Compound data types which can store multiple values of scalar types (array, String, specific classes or interfaces)

Scalar Data Types

TYPE	DESCRIPTION	SIZE
null	Represents undefined/unknown value.	
boolean	Represents logical TRUE or FALSE.	
char	Represents unicode character.	16-bit
byte	Represents integer number (20, -54).	8-bit two's complement
short	Represents integer number (20, -54).	16-bit two's complement
int	Represents integer number (20, -54).	32-bit two's complement
long	Represents integer number (20, -54).	64-bit two's complement
float	Represents real number (-45.87).	32-bit IEEE 754
double	Represents real number (-45.87).	64-bit IEEE 754

Compound Data Types

TYPE	DESCRIPTION
Array	Collection of data of the same data type.
String	Class for storing constant strings.
Class	Custom data type. Collection of data of different data types.
Interface	Custom data type. Collection of data of different data types.

Strongly typed

- JAVA is a strongly typed language meaning that every variable must have a declared data type.
- So when you declare a variable you have to define data type it can store like char, int, array or Class.
- If you try to store data type which is different from declared data type you will get an error.

Examples

```
String name = "John";
int    age  = 25;
```

Classes, Interfaces and Objects

- When you declare a class or interface you are actually declaring new custom compound data type.
- So every declared class or interface is a separate data type, either defined by user or built in into JAVA.
- Object is not a data type, instead it is actual data of specific custom data type defined by its class.
- In the below example class Person is data type and object john is data of that data type.

Examples

```
public class Test {
  public static void main(String[] args) {
    Person john = new Person();
           john.say();
  }
}

class Person {
  String name;
  int    age;
  public void say() { System.out.println("Hello World"); }
}
```

Strings [R]

- String can be considered as a special data type.
- String is declared as a built in class but such that has special powers which no other class can have
  - it is the only class with overloaded `+` operator which is used for concatenating Strings
  - it is the only class whose objects can be created without using `new` by using string literal instead "something"
- String can't be modified.

Examples

```
public class Test {
  public static void main(String[] args) {
    String start = "Hello ";     //Object creation uaing string literal insted of new keyword.
    String end   = "World";
    String greet = start + end;  //Concatenation using overloaded + operator.
    System.out.println(greet);
  }
}
```

Stack vs heap [R]

- Both stack and heap are stored in a computer RAM and are used to store data like variables, classes, etc.

Heap

- In a heap there is no order to the way items are placed so you can reach in and remove items in any order.
- Heap is where global variables (reference types) get created which allows them to be referenced from any function.
- When you declare custom data type like class this declaration is saved in a heap because it has to be globally available so that you could create data of that data type (object) in any method.

Stack

- In a stack items are stacked on top of each other using LIFO (Last In First Out) principle which means
  - You can only add new item at the top of the stack
  - You can only remove item from the top of the stack
- Stack is where local variables (value types)
  - get created when you call a function
  - are removed from when function returns
- Such local variables created on the stack will go out of scope and automatically deallocate and are used to store
  - return address to which program must go when it returns from the function
  - parameters with which function was called
  - local variables created inside the function
- When you create object it goes on top of the stack if it was created as local variable in some method

Implicit conversion of decimals and integer literals

- Any decimal number written in code, like 123.56, is implicitly converted into `double` primitive type.
  This is why you have to explicitly downcast them to smaller container like `float`.
- Any integer number written in code, like 156, is implicitly converted into `int` primitive type.
  This is why you have to explicitly downcast them to smaller containers like `byte` and `short`.

Declaring Integer and Binary

- When declaring Integer and binary constants following rules apply.

Integer constants can be DIRECTLY defined only in decimal, octal or hexadecimal format but not in binary format

```
int value = 65  ;                    //Decimal     value.
          = 0x41;                    //Hexadecimal value.
          = 0101;                    //Octal       value.
```

Binary constants, and all the other constants different from 10, 8 and 16 base can be defined like this

```
int value = Integer.parseInt("54");     //Decimal value since default base is 10.
    value = Integer.parseInt("11", 2);  //Use base two making it binary number: 1*2+1=3.
    value = Integer.parseInt("K0", 25); //Use base 25. F=15. FGHIJK=>K=20. 20*25=500.
```

2.4.1 Scalar

Info

- Scalar datatypes are capable of containing a single item of information.
- Scalar datatypes include: boolean, char, byte, short, int, long, float and double.

2.4.1.1 null

Info

- Null data type can only have `null` value.
- Data of null data type can be called "null data" for short.
- Null data can be created using Null Literal.
- Null data type can only be assigned to reference type variable which are those that hold Classes and Strings.
  You cannot assign null data to primitive variables like boolean, int, float, etc.

Example [C]

- Example uses null literal to create null data of value `null` which is then stored in a variables `name` and `age`.
- Null data can't be stored in variables declared to hold primitive types like boolean, int, float, etc.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //CREATE NULL DATA.
    String  name = null;  //Create null data using null literal.
    Integer age  = null;  //Assign null to Integer class. Different from int primitive.
    int     temp = null;  //Error. Cannot assign null data to primitive variables.

    //DISPLAY.
    System.out.println(name);

  }

}
```

output

```
null
```

2.4.1.2 bool

Info

- Boolean data type can have only two possible values which are logical TRUE or FALSE.
- Data of boolean data type can be called "boolean data" for short.
- Boolean data can be created using Boolean Literal.
- Boolean datatype is named after mathematician George Boole (1815–1864).
- Check out bool cheat sheets containing code samples for working with bool data type.

Example [C]

- Example shows how to create boolean data using
  - boolean literal `true`
  - `getBoolean()` method
- Boolean data is then stored it in variable with identifier `value`.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //CREATE BOOLEAN DATA.
    boolean value = true;                          //Create boolean data using boolean literal.
            value = Boolean.getBoolean("false");   //Create boolean data using getBoolean() method.

    //DISPLAY.
    System.out.println(value);

  }

}
```

output

```
false
```

2.4.1.3 byte

Info

- Byte data type is type of data that represents integer number.
- Data of byte data type can be called "byte data" for short.
- Byte data can be created using Integer Literal which first creates integer data which is then implicitly converted to byte.
- Check out byte cheat sheet containing code samples for working with byte data type.

Example [C]

- Example shows how to create byte data using integer literal which first creates integer data which is then implicitly converted to byte data.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //CREATE CHAR DATA.
    byte value = 65;  //Create byte data using long literal which is then implicitly converted to byte data.

    //DISPLAY.
    System.out.println(value);

  }

}
```

output

```
65
```

2.4.1.4 short

Info

- Short data type is type of data that represents integer number.
- Data of short data type can be called "short data" for short.
- Short data can be created using Integer Literal which first creates integer data which is then implicitly converted to short.
- Check out short cheat sheet containing code samples for working with short data type.

Example [C]

- Example shows how to create short data using integer literal which first creates integer data which is then implicitly converted to short data.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //CREATE SHORT DATA.
    short value = 65;    //Create short data using integer literal which is implicitly converted to short data.

    //DISPLAY.
    System.out.println(value);

  }

}
```

output

```
65
```

2.4.1.5 int

Info

- Int data type is type of data that represents integer number.
- Data of int data type can be called "int data" for short.
- Int data can be created using Integer Literal.
- Check out int cheat sheet containing code samples for working with int data type.

Example [C]

- Example shows how to create int data using
  - integer literal
  - `parseInt()` method

Test.java

```
public class Test {

  public static void main(String[] args) {

    //CREATE INTEGER DATA USING INTEGER LITERAL IN DIFFERENT NOTATIONS.
    int value = 65;                              //Decimal.
        value = 0101;                            //Octal.
        value = 0x41;                            //Hexadecimal.

    //CREATE INTEGER DATA USING method parseInt().
        value = Integer.parseInt("65"        );  //Decimal since default base is 10.
        value = Integer.parseInt("1000001", 2);  //Binary  since base is set to   2.

    //DISPLAY.
    System.out.println(value);

  }

}
```

output

```
65
```

2.4.1.6 long

Info

- Long data type is type of data that represents integer number.
- Data of long data type can be called "long data" for short.
- Long data can be created using Long Literal.
- Check out long cheat sheet containing code samples for working with longdata type.

Example [C]

- Example shows how to create int data using
  - integer literal
  - `parseInt()` method

Test.java

```
public class Test {

  public static void main(String[] args) {

    //CREATE INTEGER DATA USING INTEGER LITERAL IN DIFFERENT NOTATIONS.
    long value = 65;                              //Decimal.
         value = 0101;                            //Octal.
         value = 0x41;                            //Hexadecimal.

    //CREATE INTEGER DATA USING method parseInt().
         value = Long.parseLong("65"        );    //Decimal since default base is 10.
         value = Long.parseLong("1000001", 2);    //Binary  since base is set to   2.

    //DISPLAY.
    System.out.println(value);

  }

}
```

output

```
65
```

2.4.1.7 float

Info

- Float data type is type of data that represents real number.
- Data of float data type can be called "float data" for short.
- Float data can be created using Float Literal.
- Check out float cheat sheet containing code samples for working with float data type.

Example [C]

- Example shows how to create int data using
  - float literal
  - `parseFloat()` method

Test.java

```
public class Test {

  public static void main(String[] args) {

    //CREATE FLOAT DATA USING FLOAT LITERAL IN DIFFERENT NOTATIONS.
    float value = 65.23F;                              //Basic      notation.
          value = 0.6523E2F;                           //Scientific notation.

    //CREATE FLOAT DATA USING METHOD parseFloat().
          value = Float.parseFloat("65.23");

    //DISPLAY.
    System.out.println(value);

  }

}
```

output

```
65.23
```

2.4.1.8 double

Info

- Double data type is type of data that represents real number.
- Data of double data type can be called "double data" for short.
- Double data can be created using Double Literal.
- Check out double cheat sheet containing code samples for working with double data type.

Example [C]

- Example shows how to create double data using
  - double literal
  - `parseFloat()` method

Test.java

```
public class Test {

  public static void main(String[] args) {

    //CREATE DOUBLE DATA USING DOUBLE LITERAL IN DIFFERENT NOTATIONS.
    double value = 65.23F;                              //Basic      notation.
           value = 0.6523E2F;                           //Scientific notation.

    //CREATE DOUBLE DATA USING method parseFloat().
           value = Double.parseDouble("65.23");

    //DISPLAY.
    System.out.println(value);

  }

}
```

output

```
65.23
```

2.4.1.9 char

Info

- Char data type is type of data that uses integer number to represents single character based on ASCII Table.
- Data of char data type can be called "char data" for short.
- Char data can be created using Character Literal in different notations.
- Check out char cheat sheet containing code samples for working with char data type.

Example [C]

- Example shows how to create char data using
  - character literal
  - integer literal with value based on ASCII Table

Test.java

```
public class Test {

  public static void main(String[] args) {

    //CREATE CHAR DATA.
    char letter = 'A';  //Create char data using character literal.
         letter = 65;   //Create char data using integer   literal which is implicitly converted to char data.

    //DISPLAY.
    System.out.println(letter);

  }

}
```

output

```
A
```

2.4.2 Compound

Info

- Compound datatypes allow for multiple items of the same type to be aggregated.
- Compound datatypes include: Array, String and Object.

2.4.2.1 String

Info [R]

- String data type is type of data that represents a string, sequence of characters.
- Data of string data type can be called "string data" for short.
- String data can be created using String Literal.
- Check out String cheat sheet containing code samples for working with String data type.

Example [C]

- Example shows how to create string data
  - using string literal
  - by creating object from string class `new String("text")`

Test.java

```
public class Test {

  public static void main(String[] args) {

    //CREATE STRING DATA USING STRING LITERAL.
    String  text = null;                                  //Calling any function on text throws Exception
            text = "";                                    //Empty string.text.length()=0,text.equals("")==true
            text = "Hello World";                         //Creates string with constant text "text".
            text = "First line. \n Second line.";         //Using escape character '\': \" \t \n \\
            text = "Start" + "End";                       //Connecting Strings.

    //CREATE STRING DATA BY CREATING OBJECT FROM STRING CLASS.
            text= new String("text");
            text= new String(new byte[] {'C','a','r','s'});//Creates String from byte array.

    //DISPLAY.
    System.out.println(text);

  }

}
```

2.4.2.2 array

Info [R]

- Array data type is type of data that represents collection of ordered elements of specific data type.
- Data of array data type can be called "array data" for short.
- Array data can be created using Array Literal.
- Array can't be resized. To change array size create new one and transfer elements or use `ArrayList`, `Vector`, etc.
- Check out array cheat sheet containing code samples for working with array data type.

Example [C]

- Example shows how to create array data
  - using array literal
  - by creating object from string class `new String("text")`

Test.java

```
import java.util.Arrays;

public class Test {

  public static void main(String[] args) {

    //CREATE 1D ARRAY DATA USING ARRAY LITERAL.
    int[]   array1D  = {0,1,2,3};      //Create array data using is array literal {0,1,2,3}.

    //CREATE 2D ARRAY DATA USING ARRAY LITERAL. SUB ARRAYS CAN HAVE DIFFERENT LENGTH.
    int[][] array2D = { {0,1,2,3},
                        {4,5},
                        {6,7,8,9,10,11}
                      };

    //DISPLAY ARRAY.
    System.out.println(Arrays.toString(array1D));    //[0, 1, 2, 3]

  }

}
```

output

```
[0, 1, 2, 3]
```

2.4.2.3 Class

Info

- Class data types are custom made data types that can contain multiple items.

Example [C]

- Example shows how to create object (data) from a class (custom data type).

Test.java

```
//MAIN CLASS.
public class Test {

  public static void main(String[] args) {

    //CREATE OBJECT.
    Person john = new Person();
           john.say();

  }

}

//DECLARE CLASS.
class Person {

  String name;
  int    age;

  public void say() {
    System.out.println("Hello World");
  }

}
```

2.5 Literals

Info **[R]**

- Literal is sequence of characters that represents data by both defining data type and data value.

Literal Types

LITERAL	EXAMPLES	DESCRIPTION	
Null	`null`	Null	literal represents null data type and value which can only be null.
Boolean	`true, false`	Boolean	literal represents boolean data type and value which can be true or false.
Integer	`50`	Integer	literal represents integer data type and value which can be integer.
Long Integer	`50`	Long Integer	literal represents integer data type and value which can be integer.
Floating	`34.75`	Floating	literal represents float data type and value which can be decimal.
Double	`34.75`	Double	literal represents double data type and value which can be decimal.
Character	`'C'`	Character	literal represents character data type and value which can be any character.
String	`"Hello"`	String	literal represents string data type and value which can be any string.
Array			

2.5.1 Null

Info

- Null literal represents null data type.
- Null literal is written by using case sensitive keyword `null`.
- Null literal can be written in just one notation/format as shown below.

Test.php

```
public class Test {

  public static void main(String[] args) {

    //USE NULL LITERAL DO STORE NULL DATA INTO VARIABLE NAME.
    String name = null;

    //DISPLAY NULL DATA.
    System.out.println(name);

  }

}
```

output

```
null
```

2.5.2 Boolean

Info

- Boolean literal represents boolean data type and value.
- Boolean literal is written by using case sensitive `true` or `false`.
- Boolean literal can be written in just one notation/format as shown below.

Test.php

```
public class Test {

  public static void main(String[] args) {

    //USE BOOLEAN LITERAL DO STORE BOOLEAN DATA INTO VARIABLES.
    boolean access = true;
            access = false;

    //DISPLAY BOOLEAN DATA.
    System.out.println(access);

  }

}
```

output

```
true
```

2.5.3 Integer

Info

- Integer literal represents integer data type and value.
- Integer literal can be written in following notations
  - Decimal Notation where integer is written as decimal number in base of 10 like `-41`
  - Octal Notation where integer is written as octal number in base of 8 like `-051`
  - Hexadecimal Notation where integer is written as hexadecimal number in base of 16 like `-0x29`

Test.java

```
public class Test {

  public static void main(String[] args) {

    //USE INTEGER LITERAL DO STORE INTEGER DATA INTO VARIABLE.
    int age =   -41;         //Decimal     notation (base 10).
        age =  -051;         //Octal       notation (base 8 ).
        age = -0x29;         //Hexadecimal notation (base 16).

    //DISPLAY BOOLEAN DATA.
    System.out.println(age);

  }

}
```

output

```
-41
```

2.5.3.1 Decimal Notation

Info

- Integer literal can be written in decimal notation (base 10).

Test.java

```
public class Test {

  public static void main(String[] args) {

    //USE INTEGER LITERAL WRITTEN IN DECIMAL NOTATION.
    int age =  41;    //Positive.
        age = +41;    //Positive.
        age = -41;    //Negative.

    //DISPLAY BOOLEAN DATA.
    System.out.println(age);

  }

}
```

2.5.3.2 Octal Notation

Info

- Integer literal can be written in hexadecimal notation (base 16).

Test.java

```
public class Test {

  public static void main(String[] args) {

    //USE INTEGER LITERAL WRITTEN IN HEXADECIMAL NOTATION.
    int age =  051;    //Positive.
        age = +051;    //Positive.
        age = -051;    //Negative.

    //DISPLAY BOOLEAN DATA.
    System.out.println(age);

  }

}
```

2.5.3.3 Hexadecimal Notation

Info

- Integer literal can be given in hexadecimal notation (base 16).

Test.java

```
public class Test {

  public static void main(String[] args) {

    //USE INTEGER LITERAL WRITTEN IN HEXADECIMAL NOTATION.
    int age =  0x29;    //Positive.
        age = +0x29;    //Positive.
        age = -0x29;    //Negative.

    //DISPLAY BOOLEAN DATA.
    System.out.println(age);

  }

}
```

2.5.4 Long

Info

- Long literal represents long integer data type and value.
- Long literal can be written in following notations
  - Decimal Notation where integer is written as decimal number in base of 10 like `-41l`
  - Octal Notation where integer is written as octal number in base of 8 like `-051L`
  - Hexadecimal Notation where integer is written as hexadecimal number in base of 16 like `-0x29L`

Test.java

```
public class Test {

  public static void main(String[] args) {

    //USE LONG INTEGER LITERAL DO STORE LONG INTEGER DATA INTO VARIABLE.
    long age =   -41l;       //Decimal     notation (base 10).
         age =  -051L;       //Octal       notation (base 8 ).
         age = -0x29L;       //Hexadecimal notation (base 16).

    //DISPLAY INTEGER DATA.
    System.out.println(age);

  }

}
```

output

```
-41
```

2.5.4.1 Decimal Notation

Info

- Long literal can be written in decimal notation (base 10).

Test.java

```
public class Test {

  public static void main(String[] args) {

    //USE LONG INTEGER LITERAL WRITTEN IN DECIMAL NOTATION.
    long age =  41l;    //Positive.
         age = +41L;    //Positive.
         age = -41L;    //Negative.

    //DISPLAY LONG INTEGER DATA.
    System.out.println(age);

  }

}
```

2.5.4.2 Octal Notation

Info

- Long literal can be written in hexadecimal notation (base 16).

Test.java

```
public class Test {

  public static void main(String[] args) {

    //USE LONG INTEGER LITERAL WRITTEN IN OCTAL NOTATION.
    long age =  0511;    //Positive.
         age = +051L;    //Positive.
         age = -051L;    //Negative.

    //DISPLAY LONG INTEGER DATA.
    System.out.println(age);

  }

}
```

2.5.4.3 Hexadecimal Notation

Info

- Long literal can be written in hexadecimal notation (base 16).

Test.java

```
public class Test {

  public static void main(String[] args) {

    //USE LONG INTEGER LITERAL WRITTEN IN HEXADECIMAL NOTATION.
    long age =  0x29l;    //Positive.
         age = +0x29L;    //Positive.
         age = -0x29L;    //Negative.

    //DISPLAY LONG INTEGER DATA.
    System.out.println(age);

  }

}
```

2.5.5 Float

Info [R]

- Float literal represents float data type and value.
- Float literal is used to represent real number.
- Float literal can be written in following notations
  - Basic Notation where integer is written as decimal number in base of 10 like `123.045f`
  - Scientific Notation where integer is written as hexadecimal number in base of 2 like `1.23045E+2F`

Test.java

```
public class Test {

  public static void main(String[] args) {

    //USE DOUBLE LITERAL DO STORE DOUBLE DATA INTO VARIABLES.
    double temp = 123.045f;            //Normal     notation.
           temp = 1.23045E+2F;         //Scientific notation.

    //DISPLAY DOUBLE DATA.
    System.out.println(temp);

  }

}
```

output

```
123.04499816894531
```

2.5.5.1 Basic Notation

Info **[R]**

- Float literal can be written in basic notation by using case insensitive `F` suffix at the end of the decimal number.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //USE NORMAL NOTATION TO WRITE FLOAT LITERAL.
    float temp = 123.045f;  //Explicitly specify float literal using 'f' sufix.
          temp = 123.045F;  //Explicitly specify float literal using 'F' sufix.

    //DISPLAY FLOAT DATA.
    System.out.println(temp);

  }

}
```

2.5.5.2 Scientific Notation

Info **[R]**

- Float literal can be written in scientific notation by using case insensitive E and `F` suffixes at the end of decimal number.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //USE NORMAL NOTATION TO WRITE FLOAT LITERAL.
    float temp = 1.23045E+2F;  //Explicitly specify float literal using f' sufix.
          temp = 1.23045E+2f;  //Explicitly specify float literal using 'F' sufix.

    //DISPLAY FLOAT DATA.
    System.out.println(temp);

  }

}
```

2.5.6 Double

Info **[R]**

- Double literal represents double data type and value.
- Double literal is used to represent real number.
- Double literal can be written in following notations
  - Basic Notation where integer is written as decimal number in base of 10 like `123.045`
  - Scientific Notation where integer is written as hexadecimal number in base of 2 like `1.23045E+2`

Test.java

```
public class Test {

  public static void main(String[] args) {

    //USE DOUBLE LITERAL DO STORE DOUBLE DATA INTO VARIABLES.
    double temp = 123.045;             //Normal     notation.
           temp = 1.23045E+2;          //Scientific notation.

    //DISPLAY DOUBLE DATA.
    System.out.println(temp);

  }

}
```

output

```
123.045
```

2.5.6.1 Basic Notation

Info **[R]**

- Double literal can be written in basic notation with optional `d` or `D` suffix.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //USE NORMAL NOTATION TO WRITE DOUBLE LITERAL.
    double temp = 123.045;   //Floating point literals are by default interpreted as double literal.
           temp = 123.045d;  //Explicitly specify double literal using 'd' sufix.
           temp = 123.045D;  //Explicitly specify double literal using 'D' sufix.

    //DISPLAY DOUBLE DATA.
    System.out.println(temp);

  }

}
```

2.5.6.2 Scientific Notation

Info [R]

- Double literal can be written in scientific notation with optional case insensitive `D` suffix.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //USE NORMAL NOTATION TO WRITE DOUBLE LITERAL.
    double temp = 1.23045E+2;   //Floating point literals are by default interpreted as double literal.
           temp = 1.23045E+2d;  //Explicitly specify double literal using 'd' suffix.
           temp = 1.23045E+2D;  //Explicitly specify double literal using 'D' suffix.

    //DISPLAY DOUBLE DATA.
    System.out.println(temp);

  }

}
```

2.5.7 Character

Info

- Character literal
  - represents character data type and value where value is 16-bit unicode character
  - is constructed by enclosing character or escape sequence inside single quotes
- Escape sequence
  - can represent any character, including special ones, as shown by the table below
  - starts with backslash `/` escape character followed by specific combinations representing a character
- **Using octal or unicode character representations is equivalent to typing that character inside your source code. This differs from escape sequences \n, \' or \" which will not throw error while '\u000D', '\u0027' or '\u0022' might.**

Escape Sequences

SEQUENCE	DESCRIPTION
\n	Linefeed
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\e	Escape
\f	Form feed
\\	Backslash
\$	Dollar sign
\"	Double quote (only for Double Quotes and Heredoc)
\'	Single quote (only for Single Quotes and Nowdoc)
\d	Octal representation of character as defined in ASCII Table (For 'A' it is '\101').
\ud	Unicode representation of character as defined in ASCII Table (For 'A' it is '\u0041').

Test.java

```
public class Test {

  public static void main(String[] args) {

    //BASIC CHARACTER LITERAL.
    char mark = 'A';          //'A' character.
         mark = '\t';         //Tab character.
         mark = '\"';         //Double quote character.
         mark = '\'';         //Single quote character.

    //USING OCTAL ESCAPE SEQUENCE.
    char mark = '\101';       //'A' character.
         mark = '\10';        //Tab quote character.
         mark = '\42';        //Double quote character.
         mark = '\47';        //Single quote character.

    //USING UNICODE ESCAPE SEQUENCE. USES HEXADECIMAL ASCII VALUES WITH LEADING ZEROS.
    char mark = '\u0041';     //'A' character.
         mark = '\u0009';     //Tab character
         mark = '\u0022';     //Double quote character
         mark = '\u0027';     //Single quote character reports error.

    //DISPLAY CHARACTER DATA.
    System.out.println(mark);

  }
}
```

2.5.8 String

Info

- String literal represents string data type and value.
- String literal is used to represent strings.
- String literal is written by enclosing sequence of characters inside double quotes.
- **Using octal or unicode character representations is equivalent to typing that character inside your source code.**
  This differs from escape sequences \n, \' or \" which will not throw error while '\u000D', '\u0027' or '\u0022' might.

Escape Sequences

SEQUENCE	DESCRIPTION
\n	Linefeed
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\e	Escape
\f	Form feed
\\	Backslash
\$	Dollar sign
\"	Double quote (only for Double Quotes and Heredoc)
\'	Single quote (only for Single Quotes and Nowdoc)
\d	Octal representation of character as defined in ASCII Table (For 'A' it is '\101').
\ud	Unicode representation of character as defined in ASCII Table (For 'A' it is '\u0041').

Test.java

```
public class Test {

  public static void main(String[] args) {

    //BASIC STRING LITERAL.
    String text = "Display letter A";                    //"Display letter A" is string literal.
           text = "First line \nSecond \t line.";        //Escape sequences \n and \t for new line and tab.

    //USING OCTAL ESCAPE SEQUENCE.
           text = "Display letter \101";                 //"Display letter A".
           text = "First line \15Second \11 line.";      //"First line \nSecond \t line.".

    //USING OCTAL ESCAPE SEQUENCE.
           text = "Display letter \u0041";               //"Display letter A".
           text = "First line \nSecond \u0009 line.";    //Replacing \n with unicode causes error.

    //DISPLAY CHARACTER DATA.
    System.out.println(text);

  }

}
```

output

```
First line
Second    line.
```

2.5.9 Array

Info

- Array literal represents array data type and value.
- Array literal is written by enclosing sequence of elements inside curly brackets.

Test.java

```
import java.util.Arrays;

public class Test {

  public static void main(String[] args) {

    //ARRAY LITERAL.
    int[] array = {0,1,2,3};                    //{0,1,2,3} is array literal

    //DISPLAY ARRAY.
    System.out.println(Arrays.toString(array));  //[0, 1, 2, 3]


  }

}
```

3 Object Oriented Syntax

Info

- Following tutorials explain Object Oriented JAVA Syntax.
- Below are explanations of some of the terms closely related to object oriented programming (OOP).

Polymorphism

- Polymorphism is capability of having multiple methods with the same name but different implementation.
- There are three types of polymorphism
  - Overloaded methods have the same name but different number or type of parameters.
  - Overridden methods have the same signature but are redefined within class that extends.
  - Dynamic method binding

3.1 Classes

Info

- Class defines type of Object.
- Class is collection of properties and methods which represent certain type of data, for instance: car, animal, food.
- Class is defined by using class keyword.
- If class has defined method main() it can be executed which calls this method `javac Test.java arg0 arg1 arg2`
- Class can hold static objects of its own kind.
- Class keyword is followed by
  - class name
  - optional keyword extends followed by single class which it extends
  - optional keyword implements followed by list of interfaces separated by ','
  - optional constructor
  - optional destructor
  - optional properties (variables, objects)
  - optional methods (functions)

Test.java

```
public class Test {
```

Simple example [C]

- This example defines simple class with defined constructor and single property and method.

Test.java

```
public class Test{
  public static void main(String[] args) {
    Car fiat = new Car(200);
        fiat.startEngine();
  }
}

class Car {
  Car(int topSpeed) { this.topSpeed = topSpeed; }                                  //Constructor.
  private int  topSpeed;                                                           //Property.
  public  void startEngine() { System.out.println("Accelerating to "+topSpeed); } //Method.
}
```

Complex example [C]

- This example defines complex class which extends class Item and implements two interfaces.
- Class constructor calls parent constructor.
- Java doesn't support destructors so none is implemented, instead you can explicitly call some function like close().

Test.java

```
public class Test{
  public static void main(String[] args) {
    Instrument instrument = new Instrument(100);
               instrument.sound();
    Instrument.drums.fall();
  }
}

interface SoundInterface {
  public int  i=10;
  public void sound();
}

interface GravityInterface {
  public double g=9.81;
  public void   fall();
}

class Item {
  int weight;
  Item(int weight) { this.weight = weight; }
  int getWeight() { return weight; };
}

class Instrument extends Item implements SoundInterface, GravityInterface {
  Instrument(int weight) { super(weight); } ;
  static Instrument drums = new Instrument(500); //Static object of this class.
  public void sound() { System.out.println("Playing note"        ); };
  public void fall () { System.out.println("Falling from the sky"); };
}
```

3.1.1 Inheritance

Info [C]

- Inheritance is the capability of class to use the properties & methods of another class while adding its own functionality.
- In Java this is done through the use of extends keyword.
- Keyword `extends` allows class to extend another class, therefore inheriting its properties and methods (inheritance).
- You can extend only from one class since multiple inheritance is not supported (unlike C++).
- Single extend can be combined with implementing multiple interfaces:
- class Bird extends Animal implements interface1, interface2

Syntax

```
class Bird extends Animal { body }
class Bird extends Animal implements interface1, interface2 { body }
```

Test.java

```
public class Test{
  public static void main(String[] args) {
    Bird swan = new Bird("swan", 10);              //Create object calling constructor with two parameters.
         swan.fly();
  }
}

class Animal {
  String name;
  void   eat   ()            { System.out.println(name + "is eating."); };
  public Animal(String name) { this.name = name; } //'this' is used to distinguish 'name' variables.
}

class Bird extends Animal {
  int    wingsSpan;
  void   fly   ()            { System.out.println(name + "is flying."); };
  public Bird(String name, int span)  {
    super(name);                           //Call to Parent constructor must be first statement in constructor.
    wingsSpan = span;
  }
}
```

3.1.2 Reflection

Info [C]

- Reflection enables you to get information of class: name, methods and properties.
- You can run methods and take properties values.

Test.java

```
import java.lang.reflect.*;
import java.lang.Class;

class Unknown {
  public int height = 176;
  public int weight = 75;
  public void noArgs()      {    System.out.println("noArgs>Hello!");  }
  public void oneArg(int i) {    System.out.println("oneArg>"+i);      }
}

public class Test {

  public static void main(String[] args) {

    try {

      Unknown   obj         = new Unknown();
      Class     objectClass = obj.getClass();
      String    className   = objectClass.getName();
      Method[]  classMethods= objectClass.getMethods();
      Field[]   classFields = objectClass.getFields();    //Only public ones are displayed.
      System.out.println("className>"+className);

      //All methods.
      for(int i=0;i<classMethods.length;i++){
        System.out.println("methodName>"+classMethods[i].getName());
      }

      //All fields.
      for(int i=0;i<classFields.length;i++){
        System.out.println("fieldName>"+classFields[i].getName());
      }

      //height.
      Field     height = classFields[0];
      System.out.println("height="+height.get(obj));

      //width.
      Field     width = classFields[1];
      System.out.println("width="+width.get(obj));

      //noArgs.
      Method    noArgs = classMethods[0];
      Object[]  params1 = null;
      noArgs.invoke(obj,params1);

      //oneArgs.
      Method    oneArg = classMethods[1];
      Integer[]  params2 = new Integer[1];
      params2[0] = new Integer(12);
      oneArg.invoke(obj,params2);

    }
    catch(Exception e){}
```

```
  }

}
```

3.1.3 ClassLoader

Info [C]

- MyClassLoader can find class FindMe.class in the same directory and display its methods.

Test.java

```
import java.lang.reflect.Method;

class MyClassLoader extends ClassLoader{}

public class Test {

  public static void main(String[] args) {

    try {
      MyClassLoader myLoader    = new MyClassLoader();
      Class         findMe      = myLoader.loadClass("FindMe");
      Method[]      methods     = findMe.getMethods();
      String        methodName  = methods[0].getName();
      System.out.println("methodName="+methodName);
    }
    catch(Exception e){System.out.println(e.toString());}

  }

}
```

FindMe.java

```
class FindMe {
  public void sayHello() { System.out.println("Helo from FindMe"); }
}
```

3.2 Interface

Info [C]

- Interface is collection of methods which can be used by someone else to communicate.
- In Java interface is declared using keyword `interface`.
- Keyword `interface` declares constants and method declarations, but not method implementations.
  Java interfaces are meant to replace multiple inheritance.
- Keyword `implements` defines that class must implements functions from given list of interfaces.
- To instantiate a class that implements interface, that class has to give body to all functions from interface.
- This ensures that all classes that implement the same interface have the same set of functions (interface).

Test.java

```
public class Test{
  public static void main(String[] args) {
    Instrument instrument = new Instrument();
               instrument.sound();
  }
}

interface SoundInterface {
  public int  i=10;
  public void sound();
}

interface GravityInterface {
  public double g = 9.81;
  public void   fall();
}

class Instrument implements SoundInterface, GravityInterface {
  public void sound() { System.out.println("Playing note"        ); };
  public void fall () { System.out.println("Falling from the sky"); };
}
```

3.3 Objects

Info

- This tutorial shows how to create Java Object.
- In Java objects are created from predefined classes using new keyword followed by class name.
- This will call class constructor which might be given additional parameters.

  Constructor is function with the same name as Class name.

  You can have multiple constructors with different number and type of parameters.

Primitive Types & String [C]

- Primitive types function parameters are given to function by value.
- They are copied into local variables which are visible only inside the function.
- Changes to these local variables inside a function have no affect on the values of variables used as parameters.
- This is also true for Objects of following classes: Integer, String, ...

Test.java

```
public class Test{
  public static void main(String[] args) {
    Car fiat = new Car(200);
        fiat.startEngine();
  }
}

class Car {
  private int  topSpeed;
  public  void startEngine() { System.out.println("Accelerating to "+topSpeed); }
  Car(int topSpeed) { this.topSpeed = topSpeed; } //Constructor.
}
```

3.3.1 Create

Info

- This tutorial shows how to create Java Object and override its properties and methods.
- This means that for this Object overridden methods will behave differently then methods of other Objects of this class.

Override Properties and Methods [C]

- Primitive types function parameters are given to function by value.
- They are copied into local variables which are visible only inside the function.
- Changes to these local variables inside a function have no affect on the values of variables used as parameters.
- This is also true for Objects of following classes: Integer, String, ...

Test.java

```
public class Test{

  public static void main(String[] args) {

    //OVERRIDEN BJECT.
    Car bmw = new Car() {

      //OVERRIDE EXISTING PROPERTIES AND METHODS.
      public int  code = 100;
      public void startEngine() {
        System.out.println("Engine exploded with code "+code);
        sendMail();
      }

      //THESE WILL NOT BE ACCESSIBLE THROUGH OBJECT.
      public int  code2 = 500;
      public void sendMail() { System.out.println("Get new car with code "+code2); }

    };

    bmw.startEngine();
    //bmw.code2 = 101;   //code2 cannot be resolved or is not a field
    //bmw.sendMail();    //The method sendMail() is undefined for the type Car

    //NORMAL BJECT.
    Car fiat = new Car();
        fiat.startEngine();

  }
}

class Car {
  public int  code = 200;
  public void startEngine() { System.out.println("Engine started with code "+code); }
}
```

output

```
Engine exploded with code 100
Get new car with code 500
Engine started with code 200
```

3.3.2 Upcasting

Info

- Upcasting permits an object of a subclass type (Dog) to be treated as an object of any superclass type (Animal).
- Upcasting is done automatically since each subclass extends all the features of superclass.
- So by treating subclass as superclass you are simply using subset of its features.
- By casting you are not changing the object, you are just labeling (treating) it differently.

Example [C]

- In this example we upcast Soldier to Person, we create Object of class Soldier but treat it as if it is of class Person.
- This means that only properties and methods defined in Person will be visible which is why we can't access 'bullets'.
- It is not necessary to explicitly cast to Person since this is done automatically.

Test.java

```
public class Test {
  public static void main(String[] args) {

    Person jill = (Person) new Soldier(); //Excplicit upcasting Soldier to Person.

    Person john =          new Soldier(); //Implicit  upcasting Soldier to Person.
           john.attack();
           //john.bullets = 10;           //Bullets don't exist in Person class

    Soldier jack        =  new Soldier();
    Person  jackPerson  =  (Person) jack; //Excplicit upcasting Soldier to Person.
    Person  jackPerson2 =           jack; //Implicit  upcasting Soldier to Person.

  }
}

class Person {
  void attack() { System.out.println("Using fist."); }
}

class Soldier extends Person {
  int  bullets = 100;
  void attack() { System.out.println("Using gun."); }
}
```

output

```
Using gun.
```

3.3.3 Downcasting

Info [C]

- Downcasting permits an object which is treated as superclass (Animal) to be treated as an object of subclass type (Dog)
- Downcasting must be defined explicitly and it works only if object is being downcasted to one of the classes it extends.
- By casting you are not changing the object, you are just labeling (treating) it differently.

Test.java

```
public class Test {
  public static void main(String[] args) {

    //UPCASTING.
    Person john  = new Soldier();           //Implicit upcasting Soldier to Person.
           john.attack();
           //john.bullets = 10;             //Can't access bullets.

    //DOWNCASTING.
    if(john instanceof Soldier) {
      Soldier johnSoldier = (Soldier) john; //Explicit downcasting Person to Soldier.
               johnSoldier.bullets = 10;    //Can access bullets.
    }

  }
}

class Person {
  void attack() { System.out.println("Using fist."); }
}

class Soldier extends Person {
  public int  bullets = 100;
  void attack() { System.out.println("Using gun."); }
}
```

3.4 Exceptions

Info

- This tutorial shows basic exception handling in Java.

Example [C]

- This tutorial shows simple example of using Exceptions.

Test.java

```
public class MyClass {

  public static void main(String[] args) {

    try{
      Integer i = null;
               i.byteValue();
    }
    catch(Exception e){
      System.out.println(e.toString());
    }

  }

}
```

Print Stack Trace [C]

- This tutorial shows how to get Exception Stack Trace.
- This is a bit complicated since you can't get it as String directly from Exception Object.

Test.java

```
import java.io.PrintWriter;
import java.io.StringWriter;

public class MyClass {

  public static void main(String[] args) {

    try{
      Integer i = null;
               i.byteValue();
    }
    catch(Exception e){
      StringWriter sw = new StringWriter();
      PrintWriter  pw = new PrintWriter (sw);

      e.printStackTrace(pw);

      System.out.println("Error = " + sw.toString());
    }

  }

}
```

Output

```
Error = java.lang.NullPointerException at MyClass.main(MyClass.java:11)
```

3.5 Methods

Info

- This tutorial shows how to use methods.

Purpose

- Methods are used to avoid to copy-paste block of code that needs to be executed numerous times.
- Using methods, every time you need to execute that block of code you can simply
  - call that function by it's name,
  - give it input parameters to work on if it expects any,
  - receive it's return variable if it gives any.

Specifics

- Java function can only exist inside a class, and general term for such function is a method.
- Java function can't have another function inside it, in other words function nesting is not supported.
- Java function can only have single return variable.
  To return more then one value you can return collection, like HashMap, or some custom made Object.
- Values of function parameters are copied into local variables which are visible only inside the function.
  Changes to these local variables have no affect on the values of variables used as parameters.

3.5.1 Create

Info [C]

- This tutorial shows how to create and call methods.
- Keyword void before function name indicates that function returns no value.
- If function returns some Primitive Type or Object, it's name must be placed before function name.
- Keyword return must be used if function returns some value.
- Function parameters are defined after function name inside round brackets.
  For each parameter you must define it's type and name.
- Each function has fixed number of parameters, they all have to be given in function call and can't have default values.

Test.java

```
public class Test {

  public static void main(String[] args) {
                     noReturnValue();
    int    ret      = returnValue  ();     System.out.println(ret     );
    String text     = noParameters ();     System.out.println(text    );
    float  division = parameters   (10,2); System.out.println(division);
  }

  public static void   noReturnValue(){
    System.out.println("This function doesn't return any value.");
  }

  public static int    returnValue  (){
    System.out.print("This function returns int Primitive Type:  ");
    return 10;
  }

  public static String noParameters (){
    System.out.print("This function expects no input parameters. Returns constant String:  ");
    return "Hello";
  }

  public static float  parameters(int p1, int p2){
    System.out.print("This function expects two parameters: "+p1+"/"+p2+"=");
    return p1/p2;
  }

}
```

3.5.2 Parameters Scope

Info

- This tutorial explains scope of function parameters.
- Function parameters are given to function either by value or by reference depending on the parameter type.

Primitive Types & String [C]

- Primitive types function parameters are given to function by value.
- They are copied into local variables which are visible only inside the function.
- Changes to these local variables inside a function have no affect on the values of variables used as parameters.
- This is also true for Objects of following classes: Integer, String, ...

Test.java

```
public class Test {

  public static void main(String[] args) {

    String myString = "Hello";
    int    myInt    = 10;

    myFunction(myString, myInt);

    System.out.println(myString);   //"Hello" since changes made inside a function are not visible.
    System.out.println(myInt   );   //10      since changes made inside a function are not visible.

  }

  static void myFunction(String stringArg, int intArg){
    stringArg = "World";            //Local variable.
    intArg    = 20;                 //Local variable.
  }

}
```

output

```
Hello
10
```

Arrays [C]

- Array function parameters are given to function by reference,
- Changes to them inside a function will be visible also once the function ends.

Test.java

```
public class Test {

  public static void main(String[] args) {
    String[] myString = {"Hello"};
    myFunction(myString);
    System.out.println(myString[0]);
  }

  static void myFunction(String[] stringArg){
    stringArg[0] = "World";
  }

}
```

output

```
World
```

Object [C]

- Object function parameters are given to function by reference,
- Changes to them inside a function will be visible also once the function ends.

Test.java

```
public class Test {

  public static void main(String[] args) {

  Flag flag = new Flag();
         flag.i = 10;

    myFunction(flag);

    System.out.println(flag.i);

  }

  static void myFunction(Flag flagArg){
    flagArg.i = 20;
  }

}

class Flag{
  public int i = 0;
}
```

output

```
20
```

3.5.3 Overloading

Info

- Overloaded methods have the same name but different number or type of parameters.
- Java doesn't support methods with variable number of parameters.
- Instead you have to make separate function for different number of parameters which results in overloading.

Example [C]

- In this example Person constructor (method) is overloaded since two constructors have the same name but different number and type of parameters.

Test.java

```
public class Test {

  public static void main(String[] args) {
    Person john = new Person("John");
           john.say();

    Person jill = new Person("Jill", 25);
           jill.say();
  }

}

class Person {

  String name;
  int    age;
  void say() { System.out.println(name + " is " + age); }

  //CONSTRUCTOR WITH ONE PARAMETER.
  Person(String name) {
    this.name = name;
  }

  //CONSTRUCTOR WITH TWO PARAMETERS.
  Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

}
```

3.5.4 Overriding

Info

- Overridden methods have the same signature but are redefined within class that extends.

Example [C]

- In this example we define some default attack behavior characteristic for all Persons.
- Then we override this behavior for Soldier by giving him a gun.

Test.java

```
public class Test {

  public static void main(String[] args) {
    Soldier john = new Soldier();
            john.attack();
  }

}

class Person {

  void attack() { System.out.println("Using fist."); }

}

class Soldier extends Person {

  void attack() { System.out.println("Using gun.");  }

}
```

4 Advanced Syntax

Info

- Following tutorials explain advanced JAVA syntax which was introduced into JAVA in later versions.

4.1 Generic

Info [R]

- This tutorial shows how to write comments in Java.
- Java supports both single line and multi line (block) comments.

Test.java

```
public class Test {
```

4.1.1 Class

Info [R]

- Generic Class uses one or more symbols in places where actual Class names should be given.
  Which Class names should be used in place of symbols is defined during Object creation.
- In Java 7 constructor call can use diamond `<>` instead of redefining symbol types (type inference)

Syntax

```
Person<String, Integer> john = new Person<>("John", 25);
```

Example [C]

- In this example we define that Generic Class should accept String and Integer.
- Trying to give String to constructor where Integer is expected results in compile time error.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //VALID PARAMETERS.
    Person<String, Integer> john = new Person<String, Integer>("John", 25);
                            john.say();

    //VALID PARAMETERS.
    Person<String, String>  jack = new Person<String, String >("Jack", "25");
                            jack.say();

    //COMPILE TIME ERROR.
    //Second parameter is defined as Integer but constructor gers String.
    //Error message: The constructor Person<String,Integer>(String, String) is undefined
    //Person<String, Integer> jill = new Person<String, Integer>("Jill", "25");

  }

}

//GENERIC CLASS.
class Person<T1, T2> {

  T1 name;
  T2 age;
  Person(T1 name, T2 age) {
    this.name = name;
    this.age  = age;
  }

  public void say() {
    System.out.println(name + " is " + age);
  }

}
```

Example [R] [C]

- Generics Classes are usually used when defining Collection Classes, those that should contain collection of Objects.
- Defining that your Collection Object should contain only Strings you'll get compile time error if you try to add Integer.
- Example of such Collection Class is Vector which, before Generics were introduces, was defined to accept any Object.
- If you were planning to store only Strings in it, error would be caught to late, at run time, instead of compile time.

Test.java

```
import java.util.ArrayList;
import java.util.List;

public class Test {

  public static void main(String[] args) {

    List v = new ArrayList();
         v.add("test");
    Integer i = (Integer)v.get(0); //Run time error.

  }

}
```

Test.java

```
import java.util.ArrayList;
import java.util.List;

public class Test {

  public static void main(String[] args) {

    List<String> v = new ArrayList<String>();
                 v.add("test");
    Integer  i = v.get(0);         //Compile time error. (type error)

  }

}
```

4.1.2 Method

Info [R]

- Generic Method uses type parameters instead of defining of what class input parameters should be.

Example [C]

- In this example we define that Generic Class should accept String and Integer.
- Trying to give String to constructor where Integer is expected results in compile time error.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //VALID CALLS.
    Person.                 say("John", 50);    //SHORT    CALL. Type inference.
    Person.<String, Integer>say("John", 50);    //COMPLETE CALL.

    //INVALID CALL.
    //Person.<String, Integer>say("John","50"); //Expects Integer as second parameter.

  }

}

//GENERIC CLASS.
class Person {

  static public <U, T> void say(U u, T t) {
    System.out.println(u + " is " + t);
  }

}
```

4.1.3 Type inference

Info [R]

- Type inference enables you to invoke generic method or class without specifying type parameters between `<>`.

Generic Method can be called as normal method

```
Person.                 say("John", 50);  //SHORT    CALL. Type inference.
Person.<String, Integer>say("John", 50);  //COMPLETE CALL.
```

Generic Class can be called using diamond <> instead of redefining symbol types

```
Person<String, Integer> john = new Person<>               ("John", 25); //SHORT CALL. Type inference
Person<String, Integer> john = new Person<String, Integer>("John", 25); //COMPLETE CALL.
```

4.1.4 Type parameters

Info [R]

- Type parameters are used to define Generic Methods and Generic Classes.
- When you define generic Class or Method you use type parameters where actual Class names should be given.
- Then when you create Object or call method you define which classes should be substitutes instead type parameters.

Bounded Type Parameters [R]

- Bounded type parameters can be replaced by given Class or its subclass.
- You define upper bound and allowed Classes are connected through inheritance.

To define that parameter can be of type Integer, Number or Object you would write

```
<U extends Integer>
```

Additionally you can define if replacement Class must implement certain interfaces

```
<U extends Integer & MyInterface>
```

4.1.5 Wildcards

Info [R] [R] [R]

- Type parameters are used to define Generic Methods and Generic Classes.
- When you define generic Class or Method you use type parameters where actual Class names should be given.
- Then when you create Object or call method you define which classes should be substitutes instead type parameters.

Type Parameter vs Wildcards

- Type Parameters are used to create template for Generic Class or Method.
  Once you instantiate an Object of generic Class each type parameter is replaced with exactly one specific class.
- Now if you create method which takes Objects of such Generic Class as input parameter you can use Wildcards to more specifically define what kind of type parameter Object should have: <?>, <? extends Number> or <? super Integer>.
- If generic class has multiple type parameters you could define wild cards for each of them

```
<?, ? extends Number, ? super Integer>
```

Unbounded

- `List<?>` represents Generic List with any type parameter like shown below.

```
List<Integer>
List<String>
List<JFrame>
```

With Lower Bound

- `List<? super Integer>` represents Generic List which type parameter is Integer or supertype of Integer like shown below.

```
List<Integer>
List<Number>
List<Object>
```

With Upper Bound

- `List<? extends Number>` represents Generic List which type parameter is Number or extends Number like shown below.
- This allows using both List<Integer> and List<Float> even though there is no inheritance relationship between them.

```
List<Number>
List<Integer>
List<Float>
List<Double>
```

Example [C]

- This example demonstrates usage of wildcards.

Test.java

```
import java.util.ArrayList;
import java.util.List;

public class Test {

  public static void main(String[] args) {

    //CREATE OBJECTS FROM GENERIC LIST INTERFACE.
    List<Object > object   = new ArrayList<Object >();
    List<Number > number   = new ArrayList<Number >();
    List<Integer> integer  = new ArrayList<Integer>();
    List<Float  > floatvar = new ArrayList<Float  >();
    List<String > string   = new ArrayList<String >();

    //UNBOUNDED ACCEPTS ANY LIST.                              List<?>
    unbounded(object);
    unbounded(number);
    unbounded(string);

    //LOWER BOUND ACCEPTS NUMBER OF SUPERTYPE OF NUMBER.       List<? super Integer>
    lowerBound(object );
    lowerBound(number );
  //lowerBound(integer);  //Integer isn't supertype of Number.

    //UPPER BOUND ACCEPTS NUMBER OF CLASS THAT EXTENDS NUMBER. List<? extends Number>
  //upperBound(object  ); //Object doesn't extend from Number.
    upperBound(number  );
    upperBound(integer );
    upperBound(floatvar);

  }

  static void unbounded (List<?               > list) {}
  static void lowerBound(List<? super   Number> list) { }
  static void upperBound(List<? extends Number> list) { }

}
```

Example [C]

- In this example we create generic class person which accepts two type parameters.
- Then we define two function which accepts this generic class and we use wildcards for each type parameter.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //CREATE OBJECTS FROM GENERIC PERSON CLASS.
    Person<String , Integer> stringInteger = new Person<String , Integer>();
    Person<Object , Float  > objectFloat   = new Person<Object , Float  >();
    Person<Integer, Object > integerObject = new Person<Integer, Object >();
    Person<Number , Object > numberObject  = new Person<Number , Object >();

    //TYPE PARAMETERS: UNBOUND, UPPER BOUND.
    unboundedUpper(stringInteger);
    unboundedUpper(objectFloat  );
  //unboundedUpper(integerObject);  //Object doesn't extend Number

    //TYPE PARAMETERS: LOWER BOUND, UPPER BOUND.
    lowerUpper(numberObject );
    lowerUpper(integerObject);
  //lowerUpper(stringInteger);      //String is not supertype of Number

  }

  static void unboundedUpper(Person<?               , ? extends Number> person) { }
  static void lowerUpper    (Person<? extends Number, ? super   String> person) { }

}

//GENERIC CLASS.
class Person<T1, T2> {
  T1 name;
  T2 age;
}
```

5 Built in functionalities

Info

- Following tutorials show how to use core JAVA APIs which are built-in PHP functions.

5.1 Collections

Info

- Collections are structures which contain number of other variables.
- Arrays are the only structures in JAVA which are not Objects.
- All other collections, like HashMaps or Vectors, are created by instantiating objects of some class.

5.1.1 HashMap

Info [C]

- HashMap is structure containing key-value pairs.
- HashMap can't contain duplicate keys.
- Both keys and values can be any Object, Primitive Type or null value.
- Use HashMap instead Hashtable since
  - it allows null values for both keys and values
  - it's iterator is fail-safe throwing Exception if another thread tries to modify collection "structurally" while iterating
  - it is not thread-safe making it faster and giving you an option to synchronize it or not

Test.java

```
import java.util.HashMap;

public class Test {

  public static void main (String arg[]) throws Exception {

    //PUT ELEMENTS.-----------------------------------------
    HashMap   hashmap         = new HashMap();
              hashmap.put("key"           , "element"          ); //Add key-value pair.
              hashmap.put("age"           , 33                 );
              hashmap.put(new Integer(34) , false              );
              hashmap.put(1               , new Character('A') );

    //GET ELEMENTS.-----------------------------------------
    String    name            =   (String   ) hashmap.get ("key"           );
    int       age             = ( (Integer  ) hashmap.get ("age"           ) ).intValue     ();
    boolean   flag            = ( (Boolean  ) hashmap.get (new Integer(34) ) ).booleanValue ();
    char      stuff           = ( (Character) hashmap.get (1               ) ).charValue    ();

    //STATISTICS.-------------------------------------------
    boolean   containsElement = hashmap.containsValue("element");
    boolean   containsKey     = hashmap.containsKey  ("key"    );

    boolean   isEmpty         = hashmap.isEmpty      (         );
    int       size            = hashmap.size         (         );

    String    removed = (String) hashmap.remove       ("key"    );
                                 hashmap.clear        (         );

    //DISPLAY STATISTICS.-----------------------------------
    System.out.println ("Contains 'element' = " + containsElement);
    System.out.println ("isEmpty            = " + isEmpty        );
    System.out.println ("size               = " + size           );
    System.out.println ("removed element    = " + removed        );

  }

}
```

5.1.2 Vectors

Info [C]

- This tutorial shows how to use Vectors.
- Vector is structure that holds indexed elements where index is integer starting with 0 and can't be negative.
- Elements can only be inserted at the end or at the already existing index.
  Existing element at that index and all elements with bigger indexes are shifted to the right by increasing their index by 1
  Size of vector is increased by 1.
- When element is removed all elements with bigger indexes are shifted to the left by decreasing their index by 1.
  Size of vector is decreased by 1.

Test.java

```
import java.util.Vector;

public class Test {

  public static void main (String arg[]) throws Exception {

    //PUT ELEMENTS.-------------------------------------
    Vector   vector           = new Vector();
             vector.addElement(   "element"         ); //Append element to the end of vector.
             vector.add       (   33                ); //Append element to the end of vector.
             vector.add       (   false             ); //Elements can be of different Raw Type
             vector.add       (   new Character('A')); //or Object.
             vector.add       (3, "Shift others"    ); //Insert element at index 3 and add 1 to
                                                       //indexes of elements on index 3 or above.

    //GET ELEMENTS.-------------------------------------
    String   index_0          =   (String   ) vector.elementAt(0);
    int      index_1          = ( (Integer  ) vector.get      (1)).intValue    ();
    boolean  index_2          = ( (Boolean  ) vector.get      (2)).booleanValue();
    String   index_3          =   (String   ) vector.get      (3);
    char     index_4          = ( (Character) vector.get      (4)).charValue   ();

    //STATISTICS.---------------------------------------
    boolean  containsElement  = vector.contains(33);

    boolean  isEmpty          = vector.isEmpty ( );
    int      size             = vector.size    ( );

    String   removed = (String) vector.remove  (3 ); //Decrase size & indexes of above elements by 1
                                vector.clear   ( );

    //DISPLAY ELEMENTS.---------------------------------
    Iterator iterator         = vector.iterator();
    while( iterator.hasNext() == true ) {
      System.out.println(iterator.next());
      iterator.remove();
    }

    //DISPLAY STATISTICS.-------------------------------
    System.out.println ("Contains 'element' = " + containsElement);
    System.out.println ("isEmpty            = " + isEmpty        );
    System.out.println ("size               = " + size           );
    System.out.println ("removed element    = " + removed        );

  }

}
```

5.1.3 Iterator

Info

- Iterators are used to iterate through collection of elements with ability, in some cases, to remove last element returned.
- Iterators are part of Java Collection Framework
- Iterators should be used instead of Enumerator since they provide extended functionality and shorter function names.
- Iterator can't be instantiated but only returned by some other collection class like Vectors.

5.1.4 Hashtables

Info [R] [C]

- This tutorial shows how to use HashTables in JAVA.
- HashTable is structure containing key-value pairs.
- HashTable can't contain duplicate keys, meaning that each key can map to at most one value.
- In JAVA HashTables are implemented as Objects.
- Both keys and values can be any Object or Primitive Type.
- Use HashMap instead Hashtable since
  - it allows null values for both keys and values
  - it's iterator is fail-safe throwing Exception if another thread tries to modify collection "structurally" while iterating
  - it is not thread-safe making it faster and giving you an option to synchronize it or not

Test.java

```
import java.util.Enumeration;
import java.util.Hashtable;

public class Test {

  public static void main (String arg[]) throws Exception {

    //PUT ELEMENTS.-----------------------------------------
    Hashtable   hashtable        = new Hashtable();
                hashtable.put("key"           , "element"          ); //Add key-value pair.
                hashtable.put("age"           , 33                 );
                hashtable.put(new Integer(34) , false              );
                hashtable.put(1               , new Character('A') );

    //GET ELEMENTS.-----------------------------------------
    String      name             =   (String   ) hashtable.get ("key"           );
    int         age              = ( (Integer  ) hashtable.get ("age"           ) ).intValue     ();
    boolean     flag             = ( (Boolean  ) hashtable.get (new Integer(34) ) ).booleanValue ();
    char        stuff            = ( (Character) hashtable.get (1               ) ).charValue    ();

    //STATISTICS.-------------------------------------------
    boolean     containsElement  = hashtable.contains     ("element");
    boolean     containsElement2 = hashtable.containsValue("element");
    boolean     containsKey      = hashtable.containsKey  ("key"    );

    boolean     isEmpty          = hashtable.isEmpty      (         );
    int         size             = hashtable.size         (         );

    String      removed = (String) hashtable.remove       ("key"    );
                                   hashtable.clear        (         );

    //DISPLAY KEYS.-----------------------------------------
    Enumeration keys             = hashtable.keys();
    while( keys    .hasMoreElements()== true ) {
      System.out.println(keys.nextElement());
    }

    //DISPLAY ELEMENTS.-------------------------------------
    Enumeration elements         = hashtable.elements();
    while( elements.hasMoreElements() == true ) {
      System.out.println(elements.nextElement());
    }

    //DISPLAY STATISTICS.-----------------------------------
    System.out.println ("Contains 'element' = " + containsElement );
    System.out.println ("Contains 'element' = " + containsElement2);
```

```
    System.out.println ("isEmpty            = " + isEmpty         );
    System.out.println ("size               = " + size            );
    System.out.println ("removed element    = " + removed         );

  }

}
```

5.1.5 Enumerator

Info

- Enumerators are used to iterate through collection of elements.
- Enumerators are legacy class which should be replaced by Iterators which are part of Java Collection Framework.
- Enumerators should be replaced by Iterator since they provide extended functionality and shorter function names.
- Enumerators can't be instantiated but only returned by some other Collection classes like Hashtables.

5.1.6 Date/Time

Info [R]

- This tutorial shows how to work with Date & Time in Java.

Calendar [C]

- This tutorial shows how to use Calendar class.

Test.java

```
import java.util.Calendar;

public class Test {

  public static void main(String[] args) {

    //GET CURRENT DATE.
    Calendar  calendar = Calendar.getInstance();           //Get current date.
              calendar.set(2012,1,13);                     //Set date. year,month,dayofmonth
               calendar.set(2012,1,13, 23,45,12);          //year,month,dayofmonth, hours24,minutes,seconds
              calendar.clear();

    //GET DATE COMPONENTS.
    int year        = calendar.get(Calendar.YEAR        ); //2012
    int month       = calendar.get(Calendar.MONTH       ); //1     [0, 12]
    int dayOfMWeek  = calendar.get(Calendar.DAY_OF_WEEK ); //1     [1-Sunday,...,7-Suterday]
    int dayOfMonth  = calendar.get(Calendar.DAY_OF_MONTH); //13    [1, 31]
    int hour12      = calendar.get(Calendar.HOUR        ); //11    [0, 11]
    int hour24      = calendar.get(Calendar.HOUR_OF_DAY ); //23    [0, 23]
    int minutes     = calendar.get(Calendar.MINUTE      ); //45    [0, 59]
    int seconds     = calendar.get(Calendar.SECOND      ); //12    [0, 59]
    int miliseconds = calendar.get(Calendar.MILLISECOND ); //312   [0, 999]

    //CALCULATE.
    calendar.add(Calendar.YEAR        ,  1);               //2013  Add        1 year.   Next year
    calendar.add(Calendar.MONTH       ,  2);               //3     Add        2 months.
    calendar.add(Calendar.DAY_OF_MONTH, -1);               //12    Substract  1 month.  Yesterday.
    calendar.add(Calendar.HOUR        , -1);               //22    Substract  1 hour.
    calendar.add(Calendar.MINUTE      , 20);               //5     Add       20 minutes.
    calendar.add(Calendar.SECOND      , 20);               //32    Add       20 seconds.
    calendar.add(Calendar.MILLISECOND , 20);               //32    Add       20 miliseconds.

    //DISPLAY.
    System.out.println("year       ="+year       );
    System.out.println("month      ="+month      );
    System.out.println("dayOfMonth ="+dayOfMonth );
    System.out.println("dayOfMWeek ="+dayOfMWeek );
    System.out.println("hour12     ="+hour12     );
    System.out.println("hour24     ="+hour24     );
    System.out.println("minutes    ="+minutes    );
    System.out.println("seconds    ="+seconds    );
    System.out.println("miliseconds="+miliseconds);

  }

}
```

Date [C]

- This tutorial shows how to use Date class which is deprecated and should be replaced with Calendar described above.

Test.java

```
import java.util.Date;
import java.text.SimpleDateFormat;

public class Test {

  public static void main(String[] args) {

    //MILISECONDS.
    long   miliSecs = System.currentTimeMillis();        //Current time in miliseconds.
    System.out.println(miliSecs);                       //1107879156077

    //DISPLAY COMPONENTS OF DATE & TIME.
    Date   date              = new Date           ();   //Current date and time.
    String text              = date.toLocaleString();   //Tue Feb 08 17:04:07 CET 2005
    int    dayOfWeek         = date.getDay        ();   //0-Sunday,...,6-Suterday  Day of week.
    int    dayOfMonth        = date.getDate       ();   //[1,31]  Day of the
    int    month             = date.getMonth      ();   //[0,11]
    int    year              = date.getYear  ()+1900;   //2012
    int    hours             = date.getHours      ();   //[0,23]
    int    monutes           = date.getMinutes    ();   //34
    int    seconds           = date.getSeconds    ();   //12
    long   yearInMiliseconds = date.getTime       ();   //1107879156077 current time in miliseconds.

    //CREATE DATE OBJECT FROM.
    Date date1 = new Date();                            //Current date
    Date date2 = new Date(miliSecs);                    //miliseconds
    Date date3 = new Date(105, 0, 12);                  //(year,month,day) year=wanted-1900 month=[0,11]
    Date date4 = new Date(105, 02, 12, 12, 59, 1);      //(year,moth,day,hrs,min,sec) integers day=[1,31]

    //DISPLAY FORMATED DATE.
    SimpleDateFormat   formatter    = new SimpleDateFormat("dd:MM:yyyy HH:mm:ss.SSS");
    String             formatedDate = formatter.format(new Date());
    System.out.println(formatedDate);                   //26:03:2009 10:53:40.030

    //DISPLAY.
    System.out.println(text);                           //Tue Feb 08 17:04:07 CET 2005
    System.out.println(dayOfWeek        );              //0-Sunday,...,6-Suterday  Day of the week  [0, 6]
    System.out.println(dayOfMonth       );              //0-Sunday,...,6-Suterday  Day of the month [1,31]
    System.out.println(month            );              //[0,11]
    System.out.println(year        +1900);              //2012
    System.out.println(hours            );              //[0,23]
    System.out.println(monutes          );              //34
    System.out.println(seconds          );              //12
    System.out.println(yearInMiliseconds);              //1107879156077 urrent time in miliseconds.

    System.out.println(date1.getTime());                //1107879156077
    System.out.println(date2.getTime());                //1107879156077
    System.out.println(date3);                          //Sat Jan 12 00:00:00 CET 2005
    System.out.println(date4);                          //Sat Mar 12 12:59:01 CET 2005

  }

}
```

5.2 Serialization

Info

- Serialization is process of storing objects data as file on hard disk.
- This way data will not get lost once the application exits.
- Deserialization is process of loading objects data from file into memory.
- To make custom class serializable just add implements Serializable.

Example [C]

- In this example built in class Vector, which implements Serializable, is Serialized and Deserialized.

Test.java

```
import java.util.Vector;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class Test {

  public static void main(String[] args) throws IOException, ClassNotFoundException {

    //CREATE OBJECT.
    Vector v = new Vector();
           v.add("First");
           v.add("Second");

    //SERIALIZE OBJECT. Save object to file.
    FileOutputStream   fout = new FileOutputStream("test.dat");
    ObjectOutputStream oos  = new ObjectOutputStream(fout);
                       oos.writeObject(v);
                       oos.close();

    //DELETE VECTOR DATA.
    v.clear();

    //DESERIALIZE OBJECT. Load object from file.
    FileInputStream    fin = new FileInputStream("test.dat");
    ObjectInputStream  ois = new ObjectInputStream(fin);
    v = (Vector)       ois.readObject();
                       ois.close();

    //DISPLAY VECTOR DATA.
    System.out.println(v.get(0));
    System.out.println(v.get(1));

  }

}
```

6 Appendix

Info

- Following tutorials show how to use core JAVA APIs which are built-in PHP functions.

6.1 ASCII Table

Info **[R]**

- Following tables show ASCII values of characters which can be used with octal and unicode Escape Sequences like.
- For instance capital letter 'A' can be represented in octal or unicode Escape Sequences as `'\101'` or `'\u0041'` respectively.
- Using these character representations is equivalent to typing that character inside your source code.
  This differs from escape sequences \n, \' or \" which will not throw error while '\u000D', '\u0027' or '\u0022' will.

Control characters

DEC	HEX	BINARY	CHAR	DESCRIPTION
0	00	00000000	NUL	null
1	01	00000001	SOH	start of header
2	02	00000010	STX	start of text
3	03	00000011	ETX	end of text
4	04	00000100	EOT	end of transmission
5	05	00000101	ENQ	enquiry
6	06	00000110	ACK	acknowledge
7	07	00000111	BEL	bell
8	08	00001000	BS	backspace
9	09	00001001	HT	horizontal tab
10	0A	00001010	LF	line feed
11	0B	00001011	VT	vertical tab
12	0C	00001100	FF	form feed
13	0D	00001101	CR	enter / carriage return
14	0E	00001110	SO	shift out
15	0F	00001111	SI	shift in
16	10	00010000	DLE	data link escape
17	11	00010001	DC1	device control 1
18	12	00010010	DC2	device control 2
19	13	00010011	DC3	device control 3
20	14	00010100	DC4	device control 4
21	15	00010101	NAK	negative acknowledge
22	16	00010110	SYN	synchronize
23	17	00010111	ETB	end of trans. block
24	18	00011000	CAN	cancel
25	19	00011001	EM	end of medium
26	1A	00011010	SUB	substitute
27	1B	00011011	ESC	escape
28	1C	00011100	FS	file separator
29	1D	00011101	GS	group separator
30	1E	00011110	RS	record separator
31	1F	00011111	US	unit separator
127	7F	01111111	DEL	delete

Special characters

DEC	HEX	BINARY	CHAR	DESCRIPTION
32	20	00100000	Space	space
33	21	00100001	!	exclamation mark
34	22	00100010	"	double quote
35	23	00100011	#	number
36	24	00100100	$	dollar
37	25	00100101	%	percent
38	26	00100110	&	ampersand
39	27	00100111	'	single quote
40	28	00101000	(	left parenthesis
41	29	00101001	)	right parenthesis
42	2A	00101010	*	asterisk
43	2B	00101011	+	plus
44	2C	00101100	,	comma
45	2D	00101101	-	minus
46	2E	00101110	.	period
47	2F	00101111	/	slash

Digits

DEC	HEX	BINARY	CHAR	DESCRIPTION
48	30	00110000	0	zero
49	31	00110001	1	one
50	32	00110010	2	two
51	33	00110011	3	three
52	34	00110100	4	four
53	35	00110101	5	five
54	36	00110110	6	six
55	37	00110111	7	seven
56	38	00111000	8	eight
57	39	00111001	9	nine

Special characters

DEC	HEX	BINARY	CHAR	DESCRIPTION
58	3A	00111010	:	colon
59	3B	00111011	;	semicolon
60	3C	00111100	<	less than
61	3D	00111101	=	equality sign
62	3E	00111110	>	greater than
63	3F	00111111	?	question mark
64	40	01000000	@	at sign

Special characters

DEC	HEX	BINARY	CHAR	DESCRIPTION
91	5B	01011011	[	left square bracket
92	5C	01011100	\	backslash
93	5D	01011101	]	right square bracket
94	5E	01011110	^	caret / circumflex
95	5F	01011111	_	underscore
96	60	01100000	`	grave / accent

Capital letters

DEC	HEX	BINARY	CHAR	DESCRIPTION
65	41	01000001	A	
66	42	01000010	B	
67	43	01000011	C	
68	44	01000100	D	
69	45	01000101	E	
70	46	01000110	F	
71	47	01000111	G	
72	48	01001000	H	
73	49	01001001	I	
74	4A	01001010	J	
75	4B	01001011	K	
76	4C	01001100	L	
77	4D	01001101	M	
78	4E	01001110	N	
79	4F	01001111	O	
80	50	01010000	P	
81	51	01010001	Q	
82	52	01010010	R	
83	53	01010011	S	
84	54	01010100	T	
85	55	01010101	U	
86	56	01010110	V	
87	57	01010111	W	
88	58	01011000	X	
89	59	01011001	Y	
90	5A	01011010	Z	

Small letters

DEC	HEX	BINARY	CHAR	DESCRIPTION
97	61	01100001	a	
98	62	01100010	b	
99	63	01100011	c	
100	64	01100100	d	
101	65	01100101	e	
102	66	01100110	f	
103	67	01100111	g	
104	68	01101000	h	
105	69	01101001	i	
106	6A	01101010	j	
107	6B	01101011	k	
108	6C	01101100	l	
109	6D	01101101	m	
110	6E	01101110	n	
111	6F	01101111	o	
112	70	01110000	p	
113	71	01110001	q	
114	72	01110010	r	
115	73	01110011	s	
116	74	01110100	t	
117	75	01110101	u	
118	76	01110110	v	
119	77	01110111	w	
120	78	01111000	x	
121	79	01111001	y	
122	7A	01111010	z	

Special characters

DEC	HEX	BINARY	CHAR	DESCRIPTION
123	7B	01111011	{	left curly bracket
124	7C	01111100	|	vertical bar
125	7D	01111101	}	right curly bracket
126	7E	01111110	~	tilde
127	7F	01111111	DEL	delete

6.2 Cheat Sheets

Info

- Following tutorials contain code samples for working with different data types.

6.2.1 bool

Info

- This tutorial contains code samples for working with bool data type.
- Converting boolean true/false from number, character or string gives 1/0, t/f or "true"/"false" respectively.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //=============================================================================================
    //boolean.
    //=============================================================================================

    //CREATE.------------------------------------------------------------------------------------
    boolean value    = true;
            value    = Boolean.getBoolean  ("false");
            value    = Boolean.parseBoolean("true" );

    //CONVERT TO.--------------------------------------------------------------------------------
    byte    convert1 = value ? (byte)  1:0  ; //  1   -true ,  0   -false.
    char    convert2 = value ?       't':'f'; // 't'  -true , 'f'  -false.
    short   convert3 = value ? (short) 1:0  ; //  1   -true ,  0   -false.
    int     convert4 = value ?         1:0  ; //  1   -true ,  0   -false.
    long    convert5 = value ?         1:0  ; //  1   -true ,  0   -false.
    float   convert6 = value ?         1:0  ; // 1.0  -true , 0.0  -false.
    double  convert7 = value ?         1:0  ; // 1.0  -true , 0.0  -false.
    String  convert8 = String.valueOf(value); //Explicit conversion to String object: "A"
            convert8 =  ""  +  value;         //Implicit ocnversion while concatenating.

    //DISPLAY.-----------------------------------------------------------------------------------
    System.out.println(value);
    System.out.println(Boolean.toString(value));

    //=============================================================================================
    //Boolean.
    //=============================================================================================

    //CREATE.------------------------------------------------------------------------------------
    Boolean object   = new Boolean     (value);    //Constructor accepting boolean argument.
            object   = new Boolean     ("false");  //Constructor accepting String  argument.

            object   = Boolean.valueOf (value);    //boolean argument.
            object   = Boolean.valueOf ("true");   //String  argument.

    //CONVERT TO.--------------------------------------------------------------------------------
    boolean primitive= object.booleanValue();
    String  string   = object.toString();

    //DISPLAY.-----------------------------------------------------------------------------------
    System.out.println(object);
    System.out.println(object.toString());

  }

}
```

6.2.2 byte

Info

- This tutorial contains code samples for working with byte data type.
- Explicit conversion is needed to convert `byte` to `char` since `char` is not used to store numbers.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //============================================================================================
    //byte.
    //============================================================================================

    //CREATE.-------------------------------------------------------------------------------------
    byte    value    =  65;                          //Decimal.     Implicit conversion from int.
            value    =  0101;                        //Octal.       Implicit conversion from int.
            value    =  0x41;                        //Hexadecimal. Implicit conversion from int.

            value    = Byte.parseByte("65"        );  //Decimal     since default base is 10.
            value    = Byte.parseByte("1000001", 2);  //Binary      since base is set to   2.
            value    = Byte.parseByte("101"    , 8);  //Octal       since base is set to   8.
            value    = Byte.parseByte("41"     ,16);  //Hexadecimal since base is set to  16.

    //CONVERT TO.---------------------------------------------------------------------------------
    boolean convert1 = (value == 1) ;                 //True only if left byte argument is 1.
    char    convert2 = (char)  value;                 //Excplicit conversion since not all char fit into.
    short   convert3 = (short) value;                 //Implicit conversion since all byte fit into short.
    int     convert4 =         value;                 //Implicit conversion since all byte fit into int.
    long    convert5 =         value;                 //Implicit conversion since all byte fit into long.
    float   convert6 =         value;                 //Implicit conversion since all byte fit into long:65.0
    double  convert7 =         value;                 //Implicit conversion since all byte fit into long:65.0
    String  convert8 = String.valueOf(value);         //Explicit conversion to String object.
            convert8 =  ""  +  value;                 //Implicit ocnversion while concatenating.

    //DISPLAY.------------------------------------------------------------------------------------
    System.out.println(value);
    System.out.println(Byte.toString(value));

    //============================================================================================
    //Byte.
    //============================================================================================

    //CREATE.-------------------------------------------------------------------------------------
    Byte    object   = new Byte      (value);         //byte argument.
            object   = new Byte      ("65" );         //Decimal.

            object   = Byte.valueOf  (value       );  //byte argument.
            object   = Byte.valueOf  ("65"        );  //Decimal     since default base is 10.
            object   = Byte.valueOf  ("1000001", 2);  //Binary      since base is set to   2.
            object   = Byte.valueOf  ("101"    , 8);  //Octal       since base is set to   8.
            object   = Byte.valueOf  ("41"     ,16);  //Hexadecimal since base is set to  16.

    //CONVERT TO.---------------------------------------------------------------------------------
    byte    primitive= object.byteValue();
    String  string   = object.toString();

    //DISPLAY.------------------------------------------------------------------------------------
    System.out.println(object);
    System.out.println(object.toString());
```

```
  }

}
```

6.2.3 short

Info

- This tutorial contains code samples for working with short data type.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //============================================================================================
    //short.
    //============================================================================================

    //CREATE.-------------------------------------------------------------------------------------
    short   value    = 65;                            //Decimal.     Implicit conversion from int.
            value    = 0101;                          //Octal.       Implicit conversion from int.
            value    = 0x41;                          //Hexadecimal. Implicit conversion from int.

            value    = Short.parseShort("65"        ); //Decimal     since default base is 10.
            value    = Short.parseShort("1000001", 2); //Binary      since base is set to   2.
            value    = Short.parseShort("101"    , 8); //Octal       since base is set to   8.
            value    = Short.parseShort("41"     ,16); //Hexadecimal since base is set to  16.

    //CONVERT To.---------------------------------------------------------------------------------
    boolean convert1 = (value == 1) ;                 //True only if left byte argument is 1.
    char    convert2 = (char)  value;                 //Explicit conversion since not all short fit into char
    byte    convert3 = (byte)  value;                 //Implicit conversion since all short fit into byte.
    int     convert4 =         value;                 //Implicit conversion since all short fit into int.
    long    convert5 =         value;                 //Implicit conversion since all short fit into long.
    float   convert6 =         value;                 //Implicit conversion since all short fit into long:65
    double  convert7 =         value;                 //Implicit conversion since all short fit into long:65
    String  convert8 = String.valueOf(value);         //Explicit conversion to String object.
            convert8 =  ""  +  value;                 //Implicit conversion while concatenating.

    //DISPLAY.------------------------------------------------------------------------------------
    System.out.println(value);
    System.out.println(Short.toString(value));

    //============================================================================================
    //Short.
    //============================================================================================

    //CREATE.-------------------------------------------------------------------------------------
    Short   object   = new Short       (value);       //byte argument.
            object   = new Short       ("65" );       //Decimal.

            object   = Short.valueOf   (value       ); //byte argument.
            object   = Short.valueOf   ("65"        ); //Decimal     since default base is 10.
            object   = Short.valueOf   ("1000001", 2); //Binary      since base is set to   2.
            object   = Short.valueOf   ("101"    , 8); //Octal       since base is set to   8.
            object   = Short.valueOf   ("41"     ,16); //Hexadecimal since base is set to  16.

    //CONVERT TO.---------------------------------------------------------------------------------
    short   primitive= object.shortValue();
    String  string   = object.toString();

    //DISPLAY.------------------------------------------------------------------------------------
    System.out.println(object);
    System.out.println(object.toString());

  }
```

```
}
```

6.2.4 int

Info

- This tutorial contains code samples for working with int data type.
- Explicit conversion is needed for converting `int` to primitive types with smaller containers.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //=============================================================================================
    //int.
    //=============================================================================================

    //CREATE.--------------------------------------------------------------------------------------
    int     value    = 65;                              //Decimal.
            value    = 0101;                            //Octal.
            value    = 0x41;                            //Hexadecimal.

            value    = Integer.parseInt ("65"        ); //Decimal     since default base is 10.
            value    = Integer.parseInt ("1000001", 2); //Binary      since base is set to   2.
            value    = Integer.parseInt ("101"    , 8); //Octal       since base is set to   8.
            value    = Integer.parseInt ("41"     ,16); //Hexadecimal since base is set to  16.

    //CONVERT TO.----------------------------------------------------------------------------------
    boolean convert1 = (value == 1) ;                   //True only if left byte argument is 1.
    char    convert2 = (char)  value;                   //Explicit conversion since not all int fit into char.
    byte    convert3 = (byte)  value;                   //Implicit conversion since all int fit into byte.
    short   convert4 = (short) value;                   //Implicit conversion since all int fit into short.
    long    convert5 =         value;                   //Implicit conversion since all int fit into long.
    float   convert6 =         value;                   //Implicit conversion since all int fit into long: 65.0
    double  convert7 =         value;                   //Implicit conversion since all int fit into long: 65.0
    String  convert8 =  ""  +  value;                   //Decimal.
            convert8 = String .valueOf       (value);   //Decimal.
            convert8 = Integer.toString      (value);   //Decimal.
            convert8 = Integer.toBinaryString(value);   //Binary.
            convert8 = Integer.toOctalString (value);   //Octal.
            convert8 = Integer.toHexString   (value);   //Hexadecimal.
            convert8 = Integer.toString      (value, 5);//Base 5.

    //DISPLAY.-------------------------------------------------------------------------------------
    System.out.println(value);                          //Decimal.

    //=============================================================================================
    //Integer.
    //=============================================================================================

    //CREATE.--------------------------------------------------------------------------------------
    Integer object   = new Integer      (value);        //int argument.
            object   = new Integer      ("65" );        //Decimal.

            object   = Integer.valueOf  (value       ); //int argument.
            object   = Integer.valueOf  ("65"        ); //Decimal     since default base is 10.
            object   = Integer.valueOf  ("1000001", 2); //Binary      since base is set to   2.
            object   = Integer.valueOf  ("101"    , 8); //Octal       since base is set to   8.
            object   = Integer.valueOf  ("41"     ,16); //Hexadecimal since base is set to  16.

    //CONVERT TO.----------------------------------------------------------------------------------
    int     primitive= object.intValue();
    String  conv1    = object.toString();               //Decimal.
```

```
    //DISPLAY.-----------------------------------------------------------------------------------
    System.out.println(object);

  }

}
```

6.2.5 long

Info

- This tutorial contains code samples for working with long data type.
- Explicit conversion is needed for converting `long` to primitive types with smaller containers.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //============================================================================================
    //long.
    //============================================================================================

    //CREATE.-------------------------------------------------------------------------------------
    long    value    = 65;                          //Decimal.
            value    = 0101;                        //Octal.
            value    = 0x41;                        //Hexadecimal.

            value    = Long.parseLong ("65"        );  //Decimal     since default base is 10.
            value    = Long.parseLong ("1000001", 2);  //Binary      since base is set to   2.
            value    = Long.parseLong ("101"    , 8);  //Octal       since base is set to   8.
            value    = Long.parseLong ("41"     ,16);  //Hexadecimal since base is set to  16.


    //CONVERT TO.---------------------------------------------------------------------------------
    boolean convert1 = (value == 1) ;                  //True only if left byte argument is 1.
    char    convert2 = (char)  value;                  //Explicit conversion since not all long fit into char.
    byte    convert3 = (byte)  value;                  //Implicit conversion since not all long fit into byte.
    short   convert4 = (short) value;                  //Implicit conversion since not all long fit into short
    int     convert5 = (int)   value;                  //Implicit conversion since not all long fit into long.
    float   convert6 =         value;                  //Implicit conversion since     all long fit into long:
    double  convert7 =         value;                  //Implicit conversion since     all long fit into long:
    String  convert8 =  ""  +  value;                  //Decimal.
            convert8 = String.valueOf(value);          //Decimal.
            convert8 = Long  .toString      (value);   //Decimal.
            convert8 = Long  .toBinaryString(value);   //Binary.
            convert8 = Long  .toOctalString (value);   //Octal.
            convert8 = Long  .toHexString   (value);   //Hexadecimal.
            convert8 = Long  .toString      (value, 5);//Base 5.

    //DISPLAY.------------------------------------------------------------------------------------
    System.out.println(value);

    //============================================================================================
    //Long.
    //============================================================================================

    //CREATE.-------------------------------------------------------------------------------------
    Long    object   = new Long       (value);         //int argument.
            object   = new Long       ("65" );         //Decimal.

            object   = Long.valueOf   (value       );  //int argument.
            object   = Long.valueOf   ("65"        );  //Decimal     since default base is 10.
            object   = Long.valueOf   ("1000001", 2);  //Binary      since base is set to   2.
            object   = Long.valueOf   ("101"    , 8);  //Octal       since base is set to   8.
            object   = Long.valueOf   ("41"     ,16);  //Hexadecimal since base is set to  16.

    //CONVERT TO.---------------------------------------------------------------------------------
    long    primitive= object.intValue();              //Decimal.
```

```
    String  string   = object.toString();              //Decimal.

    //DISPLAY.-------------------------------------------------------------------------------
    System.out.println(object);

  }

}
```

6.2.6 float

Info

- This tutorial contains code samples for working with float data type.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //=============================================================================================
    //float.
    //=============================================================================================

    //CREATE.------------------------------------------------------------------------------------
    float   value  =  65;                       //Decimal.
            value  =  0101;                     //Octal.
            value  =  0x41;                     //Hexadecimal.

            value  = Float.parseFloat ("65.0" );

    //CONVERT TO.--------------------------------------------------------------------------------
    boolean conv1  = (value == 1) ;             //True only if left byte argument is 1.
    char    conv2  = (char)  value;             //Explicit conversion since not all float fit into char.
    byte    conv3  = (byte)  value;             //Implicit conversion since not all float fit into byte.
    short   conv4  = (short) value;             //Implicit conversion since not all float fit into short
    int     conv5  = (int)   value;             //Implicit conversion since not all float fit into int.
    long    conv6  = (long)  value;             //Implicit conversion since     all float fit into long.
    double  conv7  =         value;             //Implicit conversion since     all float fit into doubl
    String  conv8  = String.valueOf(value);     //Explicit conversion to String object.
            conv8  =  ""  +  value;             //Implicit conversion while concatenating.

    //DISPLAY.-----------------------------------------------------------------------------------
    System.out.println(value);
    System.out.println(Float.toString   (value));
    System.out.println(Float.toHexString(value));

    //=============================================================================================
    //Float.
    //=============================================================================================

    //CREATE.------------------------------------------------------------------------------------
    Float   object = new Float        (value);    //int argument.
            object = new Float        ("65.0" );  //Decimal.

            object = Float.valueOf    (value);    //int argument.
            object = Float.valueOf    ("65" );    //Decimal since default base is 10.

    //CONVERT TO.--------------------------------------------------------------------------------
    float   primitive = object.intValue();
    String  string    = object.toString();

    //DISPLAY.-----------------------------------------------------------------------------------
    System.out.println(object);
    System.out.println(object.toString());

  }

}
```

6.2.7 double

Info

- This tutorial contains code samples for working with double data type.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //============================================================================================
    //double.
    //============================================================================================

    //CREATE.-------------------------------------------------------------------------------------
    double  value  =  65;                          //Decimal.
            value  =  0101;                        //Octal.
            value  =  0x41;                        //Hexadecimal.

            value  = Double.parseDouble ("65.0" );

    //CONVERT TO.---------------------------------------------------------------------------------

    boolean conv1  = (value == 1) ;                //True only if left byte argument is 1.
    char    conv2  = (char)  value;                //Explicit conversion since not all double fit into char
    byte    conv3  = (byte)  value;                //Implicit conversion since not all double fit into byte
    short   conv4  = (short) value;                //Implicit conversion since not all double fit into shor
    int     conv5  = (int)   value;                //Implicit conversion since not all double fit into int.
    long    conv6  = (long)  value;                //Implicit conversion since     all double fit into long
    float   conv7  = (float) value;                //Implicit conversion since     all double fit into floa
    String  conv8  = String.valueOf(value);        //Explicit conversion to String object.
            conv8  =  ""  +  value;                //Implicit conversion while concatenating.

    //DISPLAY.------------------------------------------------------------------------------------
    System.out.println(value);
    System.out.println(Double.toHexString(value));
    System.out.println(Double.toHexString(value));

    //============================================================================================
    //Double.
    //============================================================================================

    //CREATE.-------------------------------------------------------------------------------------
    Double  object = new Double        (value);    //int argument.
            object = new Double        ("65.0" );  //Decimal.

            object = Double.valueOf    (value);    //int argument.
            object = Double.valueOf    ("65" );    //Decimal since default base is 10.

    //CONVERT TO.---------------------------------------------------------------------------------
    double  primitive= object.intValue();
    String  string   = object.toString();

    //DISPLAY.------------------------------------------------------------------------------------
    System.out.println(object);
    System.out.println(object.toString());

  }

}
```

6.2.8 char

Info

- This tutorial contains code samples for working with char data type.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //=============================================================================================
    //char.
    //=============================================================================================

    //CREATE.--------------------------------------------------------------------------------------
    char      value    =  'A';
              value    =  65;              //Decimal     ASCII value.
              value    =  0101;            //Octal       ASCII value.
              value    =  0x41;            //Hexadecimal ASCII value.

    //CONVERT TO.----------------------------------------------------------------------------------
    boolean   convert1 = (value == 1) ;     //True only if left byte argument is 1.
    byte      convert2 = (byte)  value;     //Explicit conversion since not all char fit into byte.
    short     convert3 = (short) value;     //Explicit conversion since not all char fit into short.
    int       convert4 =         value;     //Implicit conversion since     all byte fit into int.
    long      convert5 =         value;     //Implicit conversion since     all byte fit into long.
    float     convert6 =         value;     //Implicit conversion since     all byte fit into float.
    double    convert7 =         value;     //Implicit conversion since     all byte fit into double
    String    convert8 = String.valueOf(value);  //Explicit conversion to String object: "A"
              convert8 =  ""  +  value;     //Implicit ocnversion while concatenating.

    //DISPLAY.-------------------------------------------------------------------------------------
    System.out.println(value);

    //=============================================================================================
    //Character.
    //=============================================================================================

    //CREATE.--------------------------------------------------------------------------------------
    Character object   = new Character( value    ); //Constructor accepting char argument.
              object   = new Character((char)65  ); //Decimal     ASCII value.
              object   = new Character((char)0101); //Octal       ASCII value.
              object   = new Character((char)0x41); //Hexadecimal ASCII value.

              object   = Character.valueOf(value);  //char argument.

    //CONVERT TO.----------------------------------------------------------------------------------
    char      primitive= object.charValue();
   String    string   = object.toString();

    //DISPLAY.-------------------------------------------------------------------------------------
    System.out.println(object);
    System.out.println(object.toString());

  }

}
```

6.2.9 String

Info

- This tutorial contains code samples for working with String data type.

Test.java

```
public class Test {

  public static void main(String[] args) {

    //CREATE.-------------------------------------------------------------------------------------
    String  text     = null;                                //Calling any function on text throws Exception
            text     = "";                                  //Empty string.text.length()=0,text.equals("")==true
            text     = "text";                              //Creates string with constant text "text".
            text     = "Name\t\"Jo\"\n";                    //Using escape character '\': \" \t \n \\
            text     = "Start"+"End";                       //Connecting Strings.

            text     = new String();                        //Empty string.
            text     = new String("text");                  //Creates string with constant text "text".
            text     = new String        (new byte[] {'C','a','r','s'}); //Creates String from byte array.
            text     = new String        (new char[] {'C','a','r','s'}); //Creates String from char array.

            text     = String.copyTextOf(new char[] {'C','a','r','s'}); //Creates String from char array.
            text     = String.copyTextOf(new char[] {'C','a','r','s'}, 1, 2); //From index=1 take 2.

            text     = String.textOf (       false);        //boolean
            text     = String.textOf (        'A' );        //char
            text     = String.textOf (        65  );        //int
            text     = String.textOf ((long)  65  );        //long
            text     = String.textOf ((float) 65.0);        //float
            text     = String.valueOf(        65.0);        //double

    //DISPLAY.------------------------------------------------------------------------------------
    System.out.println(value);

    //CONCATENATE.--------------------------------------------------------------------------------
    String  text     = "START"+"END";                       //Connects two strings into new one.
            text    += text;                                //Connects string with itself.

    //CONVERT TO.---------------------------------------------------------------------------------
    boolean convert1 = Boolean.parseBoolean("true");        //All else is false including null.

    char    convert2 = "A"    .charAt      (0);

    byte    convert3 = new Byte   ("10").byteValue();       //Decimal     since default base is 10.
            convert3 = Byte   .parseByte   ("10");          //Decimal     since default base is 10.
            convert3 = Byte   .parseByte   ("1000001", 2); //Binary      since base is set to   2.
            convert3 = Byte   .parseByte   ("101"    , 8); //Octal       since base is set to   8.
            convert3 = Byte   .parseByte   ("41"     ,16); //Hexadecimal since base is set to  16.

    short   convert4 = new Short  ("10").shortValue();      //Decimal     since default base is 10.
            convert4 = Short  .parseShort  ("10");          //Decimal     since default base is 10.
            convert4 = Short  .parseShort  ("1000001", 2); //Binary      since base is set to   2.
            convert4 = Short  .parseShort  ("101"    , 8); //Octal       since base is set to   8.
            convert4 = Short  .parseShort  ("41"     ,16); //Hexadecimal since base is set to  16.

    int     convert5 = new Integer("10").intValue();        //Decimal     since default base is 10.
            convert5 = Integer.parseInt    ("10");          //Decimal     since default base is 10.
            convert5 = Integer.parseInt    ("1000001", 2); //Binary      since base is set to   2.
            convert5 = Integer.parseInt    ("101"    , 8); //Octal       since base is set to   8.
            convert5 = Integer.parseInt    ("41"     ,16); //Hexadecimal since base is set to  16.
```

```
    long    convert6 = new Long   ("10").longValue();      //Decimal     since default base is 10.
            convert6 = Long   .parseLong   ("10");         //Decimal     since default base is 10.
            convert6 = Long   .parseLong   ("1000001", 2); //Binary      since base is set to   2.
            convert6 = Long   .parseLong   ("101"    , 8); //Octal       since base is set to   8.
            convert6 = Long   .parseLong   ("41"     ,16); //Hexadecimal since base is set to  16.

    float   convert7 = Float  .parseFloat  ("2.5");

    double  convert8 = Double .parseDouble ("2.5");

    //ANALYZE.-------------------------------------------------------------------------------------
    String  text    = "012345";
    int     index   = text.indexOf           ('3'     );   //First occurence of  '3'                 or -1.
            index   = text.indexOf           ('3' , 2 );   //First occurence of  '3' starting from 2 or -1.
            index   = text.indexOf           ("23"    );   //First occurence of "23"                 or -1.
            index   = text.indexOf           ("23", 2 );   //First occurence of "23" starting from 2 or -1.
            index   = text.lastIndexOf       ('3' , 2 );   //Last  occurence of  '3' starting from 2 or -1.
            index   = text.lastIndexOf       ('3' , 2 );   //Last  occurence of  '3' starting from 2 or -1.
            index   = text.lastIndexOf       ("23"    );   //Last  occurence of "23"                 or -1.
            index   = text.lastIndexOf       ("23", 2 );   //Last  occurence of "23" starting from 2 or -1.

    boolean test    = text.startsWith        ("0123"  );   //true if text starts with "0123". Else false.
            test    = text.startsWith        ("234", 2);   //true if there is string "234" at index 2.
            test    = text.endsWith          ("345"   );   //true if text ends with "345". Else false.
            test    = text.equals            ("012345");   //true if strings are equal.
            test    = text.equalsIgnoreCase  ("012345");   //true if strings are equal ignoring case.

    int     cmp     = text.compareTo         ("012345");   //0 if equal, <0 if text<"Second", else >0
            cmp     = text.compareToIgnoreCase("012345");  //Same as above ignoring case.

    int     length  = text.length();                       //Returns number of characters in string.

    //GET SUBSTRINGS.------------------------------------------------------------------------------
            text    = "012345";
    String  sub     = text.substring (2);     //Take substring from index 2 till the end. Result is "2345"
            sub     = text.substring (2,5);   //Take substring from index 2 till 5-1=4.   Result is "234".

    String  changed = text.trim       ();     //Erases leading and trailing whitespaces: \t \n

            text    = "zero<->one<->two<->three";
    String[] tokens = text.split("<->"  );    //Split string around "<->" regex. Gives tokens[0]="one";
            tokens  = text.split("<->",3);    //Create maximum 3 tokens. Gives tokens[2]="two<->three";
            tokens  = text.split("[135]");    //Split string around 1,3 and 5.
            tokens  = text.split("/"    );    //Split string around '/'.
            tokens  = text.split("\\."  );    //Split string around '.'.
            tokens  = text.split("\\\\" );    //Split string around '\'.
            tokens  = ""  .split(";"    );    //One token  is  returned resulting in tokens.length=1.
            tokens  = ";" .split(";"    );    //No  tokens are returned resulting in tokens.length=0.

    //MODIFY.--------------------------------------------------------------------------------------
            text    = "012345  \n";
    String  changed = text.toUpperCase ();          //Converts all characters to upper case.
            changed = text.toLowerCase ();          //Converts all characters to lower case.
            changed = text.replace     ('3', 'A' ); //Replace ALL   occurrences of char   '3'  with 'A'.
            changed = text.replaceFirst("23","AB"); //Replace FIRST occurrences of regex  "23" with "AB".
            changed = text.replaceAll  ("23","AB"); //Replace ALL   occurrences of regex  "23" with "AB".
            changed = text.replaceAll  ("[\n\r]","");//Remove all \n and \r characters.

  }

}
```

6.2.10 array

Info

- Following tutorials contains code samples for working with array data type.

6.2.10.1 Single dimensional

Info

- This tutorial contains code samples for working with single dimension array data type.

Array

Create with [] after array Type

```
int[] array;                                //Declared. Used to define scope.
int[] array = null;                         //Declared, set to null.
int[] array = new int[5];                   //Declared, memory reserved, initial values.
int[] array = new int[ ] {0,1,2,3};         //Declared, memory reserved, elements added.
int[] array = {0,1,2,3};                    //Declared, memory reserved, elements added.
```

Create with [] after array Name

```
int array[];                                //Declared. Used to define scope.
int array[] = null;                         //Declared, set to null.
int array[] = new int[5];                   //Declared, memory reserved, inital values.
int array[] = new int[ ] {0,1,2,3};         //Declared, memory reserved, elements added.
int array[] = {0,1,2,3};                    //Declared, memory reserved, elements added.
```

Create with Initial Values

```
boolean[] array = new boolean[5];           //Elements set to initial value false.
char   [] array = new char   [5];           //Elements set to initial value 0x0000.
byte   [] array = new byte   [5];           //Elements set to initial value 0.
short  [] array = new short  [5];           //Elements set to initial value 0.
int    [] array = new int    [5];           //Elements set to initial value 0.
long   [] array = new long   [5];           //Elements set to initial value 0.
float  [] array = new float  [5];           //Elements set to initial value 0.0.
double [] array = new double [5];           //Elements set to initial value 0.0.
String [] array = new String [5];           //Object elements are set to initial value null.
```

Modify

```
array = null;                               //Set to null.
array = new int[5];                         //Memory reserved, initial values.
array = new int[ ] {0,1,2,3};               //Memory reserved, elements added.
array = array2;                             //array gets dimension & all elements from array2.
```

Analyze

```
int length = array.length;                  //Number of elements of Array array.
int length = Array.getLength(array);        //Number of elements of Array array.
```

Display

```
System.out.println(Arrays.toString(array));    //[0, 1, 2, 3]
```

Element

Add

```
array[0] = 5;                                 //Add value 5 at index 0.
Array.set       (array, 0, 5);                //Add value 5 at index 0.
Array.setBoolean(array, 0, true);             //Exception if can't convert to array type.
Array.setChar   (array, 0, 'A');              //-||-.
Array.setByte   (array, 0, (byte)  0xFF);     //-||-. Convert from int to byte.
Array.setShort  (array, 0, (short) 0xFFFF);   //-||-. Convert from int to short.
Array.setInt    (array, 0, 5);                //-||-.
Array.setLong   (array, 0, 5);                //-||-.
Array.setFloat  (array, 0, (float) 5.5);      //-||-. Convert from double to float.
Array.setDouble (array, 0, 5.5);              //-||-.
```

Get

```
int     element = array[0];                   //Get element at index 0.
int     element = Array.get        (array, 0); //Get element at index=0 from Array array.

boolean element = Array.getBoolean (array, 0); //Exception if can't convert from array to element.
char    element = Array.getChar    (array, 0); //-||-.
byte    element = Array.getByte    (array, 0); //-||-.
short   element = Array.getShort   (array, 0); //-||-.
int     element = Array.getInt     (array, 0); //-||-.
long    element = Array.getLong    (array, 0); //-||-.
float   element = Array.getFloat   (array, 0); //-||-.
double  element = Array.getDouble  (array, 0); //-||-.
```

6.2.10.2 Multi Dimensional

Info

- This tutorial contains code samples for working with multi dimensional array data type.
- Multi dimensional arrays are created simply by using arrays as elements for another array.
- Each array that is element of another array has it's own length throwing Exception if you over-index it.

Array

Create int array

```
int[][]  array;
int[][]  array   = null;
int[][]  array   = new int[2][4];
int[][]  array   = new int[ ][ ] { {0,1,2,3}, {4,5}, {6,7,8,9,10,11} };
int[][]  array   = { {0,1,2,3},
                     {4,5}
                     {6,7,8,9,10,11}
                   };
```

Create String array

```
String[][][] multyArray =
  {                                        //Top level array containing two middle level arrays.
    {                                      //First middle layer array containing two arrays.
      {"There","is"},                      //Lowest level array 1.
      {"something","fishy","going","on."}  //Lowest level array 2.
    },
    {                                      //Second middle layer array containing two arrays.
      {"These","are","just","arrays."},    //Lowest level array 3.
      {"No","way."}                        //Lowest level array 4.
    }
  };
```

Element

Add

```
array     [1][0]    = 7;                    //Element at second row and first column.
multyArray[0][1][3] = "on.";                //Element at first table, second row & forth column = "on."
```

Get

```
int      element = array[0][2];             //Get element at second row and first column=2.
```