Object-Oriented Programming Using Java

Structured Programming

Quan Thai Ha

HUS

February 20, 2024

Presentation Outline

1 Basics of computer programming

2 Java Programming Introduction

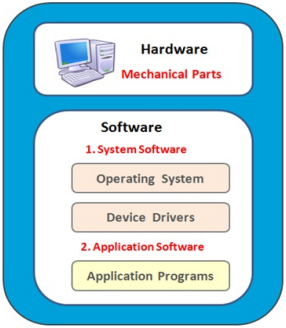
3 Java Programming Basics

4 Memory Model

5 Parameter Passing Mechanism

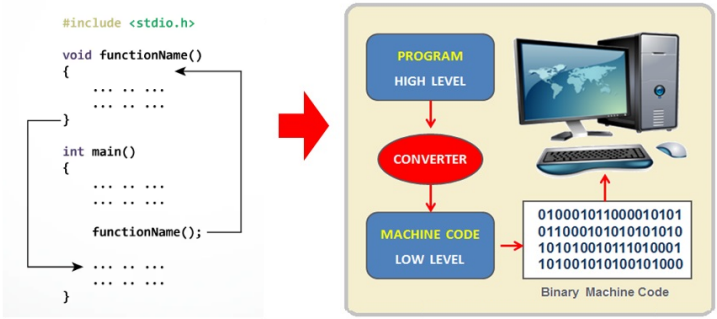
6 References

Computer Hardware Components



Computer Components

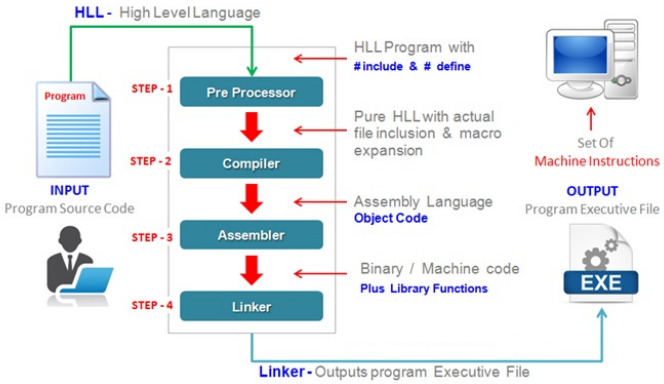
- The computer system consist of both software and hardware components. The operating system is a system software that manages all the crucial functions of the computer system.
- A computer system is a digital electronic device that needs to be programmed to perform any meaningful task. The computer program directs the computer microprocessor (CPU) to perform the desired operations as per the program instructions.



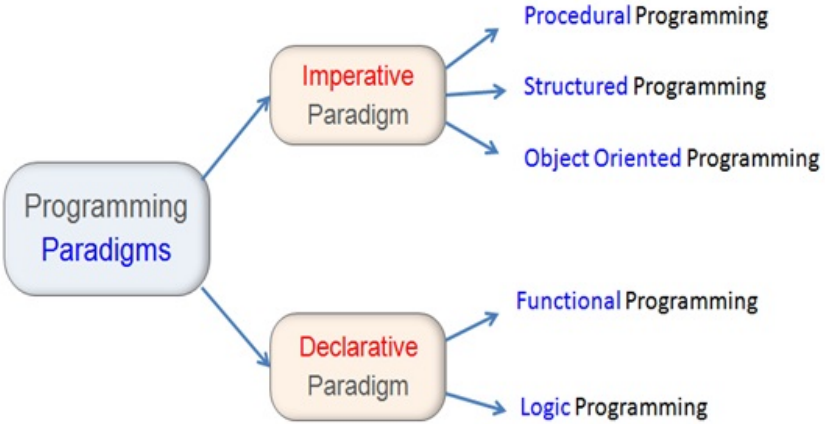
Computer Program

- The computer program consist of set of program instructions and each of these instruction performs a specific task. The computer program can be written using any high level programming language.
- Some of the most popular and commonly used programming languages include C language, C++, Java, Python, JavaScript, C# (C Sharp), and many other high level languages.

- When the user initiates the program execution, the operating system allocates the necessary resources in terms of processor time and the main memory (RAM). The CPU is the brain of the computer system. The CPU reads these instructions from the memory and executes the program instructions one by one.



- The computer system can understand and execute only machine code instructions in binary. The binary code consist of only two numbers that is 0 (zero) and 1 (one).
- And therefore, all the programs written in any programming language must be first converted to machine code instructions in binary which computer CPU can interpret and execute. This conversion process is known as program compilation.

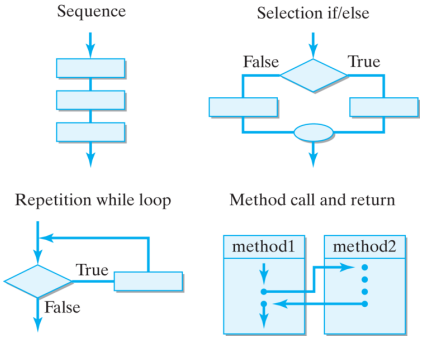

Programming Paradigms
Imperative Paradigm
Declarative Paradigm
Procedural Programming
Structured Programming
Object Oriented Programming
Functional Programming
Logic Programming

- The programming paradigms are simply different approaches to organize and structure the program code. Alternately paradigms can also be referred as methodology, approach or a particular style to write and organize the program code.
- The computer science has evolved both in the hardware and software domain. Similarly, the computer programming approaches and problem solving techniques has also evolved over the period of last few decades.
- There are a number of alternative approaches to program methodology and the manner in which the program is written and organized while building a software. Each of these programming style offers different advantages and limitations.
- These different programming styles are referred to as programming paradigms. Each of these paradigm represent fundamentally different approaches to building solutions to specific types of problems using programming.
- Different programming languages came in to existence which allow the programmers to write the code as per one paradigm. Most programming languages fall under one paradigm, but some languages have elements of multiple paradigms.

Non-Structured Programming

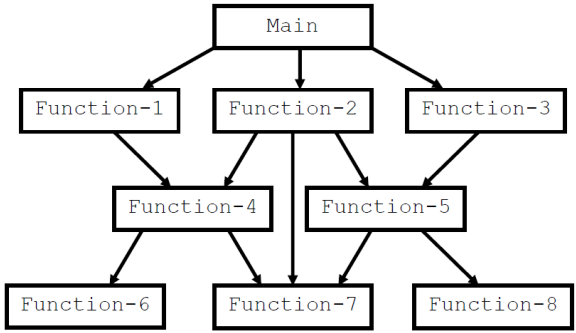
- The non-structured programming paradigm was used in the first generation of programming languages. Basic, COBOL and Assembly language are examples of non-structured programming.
- As per non-structured programming style of programming, the code is organized as collection of program statement which gets sequentially executed from first statement till the last statement.
- In non-structured programming paradigm the code is written in a single continuous file. Each program statement has a statement number or label.
- One of the major limitation of non-structured approach was the difficulties due to spaghetti code which makes the program code difficult to debug.
- The spaghetti code is a badly entangled code due to liberal use of GOTO statement. In large and complex code it is difficult to track the flow of control due multiple use of GOTO statements.
- The non–structured programming also had major limitation in terms of difficulty to reuse the program code. Further, it was not possible to secure the data from unintended changes in this approach.

- The next generation of programming languages like C language and Pascal were based on the concept of structured programming.
- In structured programming, the program code is modularized in the form of functions. The function once defined can be called number of times. A large chunk of code can be replaced by a single function.



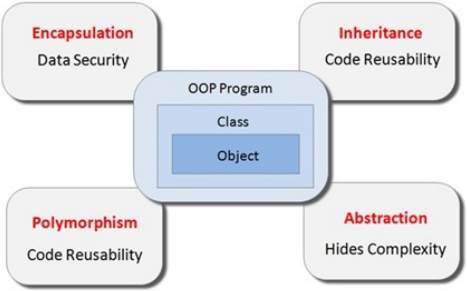
- The structured programming approach also made it easy to debug the program code. For example the logical errors in the program code can be easily fixed by making the necessary corrections in the function code.
- A structured language has constructs like IF-THEN-ELSE condition statement, WHILE-NEXT conditional loop, SWITCH-CASE statements, DO-WHILE loop, and FOR-NEXT loop. These statements helps to simplify the program algorithm.

- Although structured programming approach was a major improvement over its predecessor. However, this approach also had some major limitations.
- In structured programming approach, it is difficult to perfectly model the real world objects (entities).



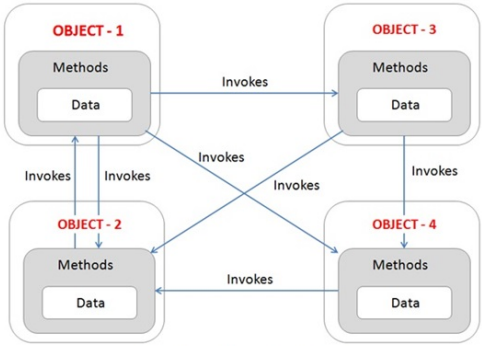
- In structured programming approach, it was difficult to secure the data and it was difficult to protect the data was unwarranted operations. In large and complex code, it is difficult to track the operations of various functions operating on the same data which can lead to spaghetti code.

- The Object-Oriented Programming (OOP) approach was next generation of programming paradigm developed to overcome the limitations of structured programming approach. The OOP methodology addressed two major limitations of the structured programming.



- In structured programming, first it was difficult to secure the data and second, it was also difficult to model the real world entities.
- The OOP methodology does not provide direct access to the program data. The OOP allows the data to be treated as private and the programmer can restrict the data access.
- Further, with object-oriented approach, allows the programmers to easily and realistically model the real world entities.

- The Object-Oriented Programming (OOP) is an approach to the software application development where all application components are treated as objects.
- An object is a smallest component of a program that has some properties and behavior. The object behavior defines how to perform certain actions and interact with other elements of the program.



- The objects are the basic units of OOP. In OOP methodology, the program code is organized in the form of set of classes. The class defines the object in the program code.
- Each and every class represents a real world entity. In other words, the entities are objects that need to be represented in the program code.

1 Basics of computer programming

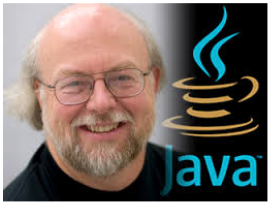
2 Java Programming Introduction

3 Java Programming Basics

4 Memory Model

5 Parameter Passing Mechanism

6 References



James Gosling
1995, Sun Microsystems

Use C/C++ as foundation
- "Simpler" in syntax
- Less low-level machine interaction

- Write Once, Run Everywhere™
- Extensive and well documented standard library

- Less efficient

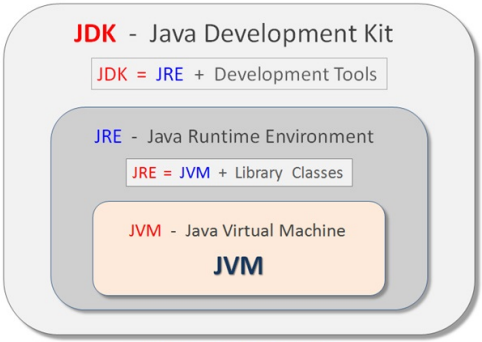
History of Java

- Java was developed by James Gosling and his colleagues while working on a project at Sun Microsystems in year 1991. The Java language was initially developed for developing embedded software for electronic gadgets.
- The Java language development project started as a project called "Oak" by James Gosling. However, due to some legal issues this language was later on renamed as Java. Java was officially released in year 1995.
- The Java language syntax is quite similar to the syntax of C language and C++. However, some crucial features of these languages were excluded in Java syntax in order to make the Java more secure language.
- The primary motive of the development of Java was to create language that is platform independent. This means Java program should have the capability of "Write Once, Run Anywhere".
- The Java's birth also coincided with the advent of smart mobile handsets. The Java's capabilities were tailor made for mobile and touch screen devices. Java soon became one of the most popular, versatile and widely used language which now powers over 3 billion devices.

What Is Java?

- Java is high level, platform independent, object-oriented programming language. Java is a versatile language which is used for creating different types of software applications. Java is used for building desktop applications, enterprise level applications, web applications, android applications for mobile platforms.
- The Sun Microsystems where Java originated defines Java as simple, object-oriented, network-savvy, interpreted, robust, secure, architecture-neutral, portable, high-performance, multi-threaded, dynamic computer language.
- We can think of Java as a general-purpose, object-oriented language that looks a lot like C and C++, but Java is much easier to use and far more versatile. Java lets you create more secure and robust programs that can run on multiple platforms.
- Java comes in three editions. Java standard edition is used to create general purpose software. The enterprise edition of Java is used to create far more complex and distributed enterprise level applications.
- Whereas the micro edition of Java is a light weight that is optimized for mobile app and embedded system development.

- The Java Environment refers to the runtime environment provided by the Java platform needed for the development and the execution of Java programs.
- The Java Environment has three major components. These components include:
  - Java Development Kit (JDK)
  - Java Runtime Environment (JRE)
  - Java Virtual Machine (JVM)



- The Java developers would typically need all the three components required for the development and also for testing (JDK = JRE + Tools) the Java applications.
- Whereas the end user (client) machine would need only components (JRE = JVM + Class Libraries) necessary for execution of Java program.

- The Java Development Kit (JDK) as the name suggest is primarily required for development of Java applications. The JDK consist of two components. These two components are Java Runtime Environment and some other application development tools.
- The JDK is intended for java developers. The JDK includes all the three components required for the development and also for testing the Java applications (JDK = JRE + Tools).
- The Java Runtime Environment (JRE) as the name suggest is primarily required for providing runtime environment to the Java applications. The JRE consist of two components. These two components are Java Virtual Machine and some class libraries.
- The JRE is installed on the end user machines for providing runtime environment to the Java software application needed for software deployment (JRE = JVM + Class Libraries).
- The Java Virtual Machine (JVM) is a vital component of the JRE which provides runtime environment to the Java application.
- The JVM accepts the platform independent Bytecode (dot class file) generated by the Java compiler (javac) as input. The JVM converts the Bytecode into platform specific native machine code and executes this code line by line.

JDK: Java Development Kit

JDK provides required environment to develop, debug and run java program. JDK = JRE + Development Tools

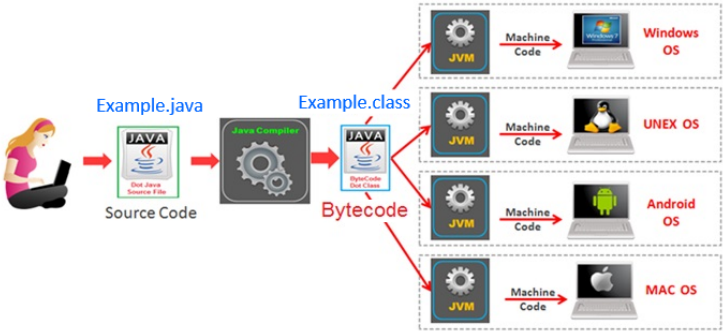
JRE: Java Runtime Environment

JRE provides required environment to run the java program on specific machine. JRE = JVM + Library Classes

JVM: Java Virtual Machine

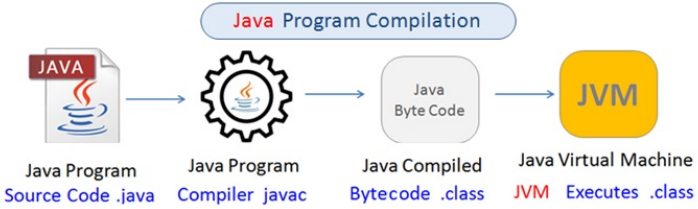
JVM is crucial component which makes the java platform independent. JVM accepts Bytecode (.class) file as input And provides Native Machine Code as output specific to the platform.

- Java is a high level programming language. The Java program gets compiled in two stages. The Java compiler does not directly compile the Java program to native machine code.

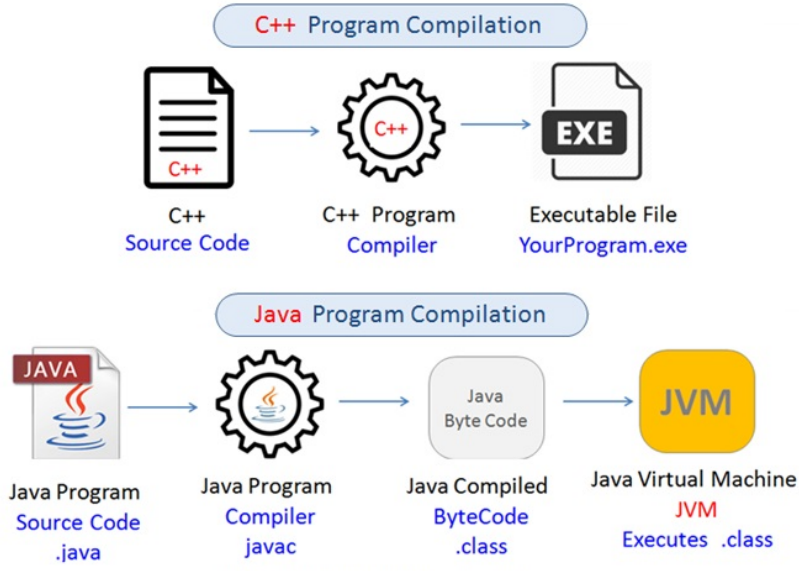


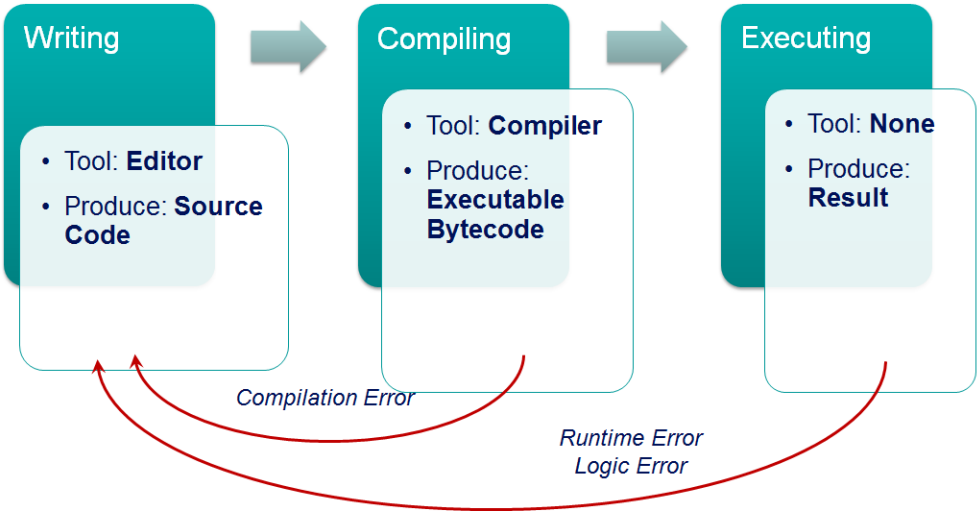
- Instead, the Java program first gets compiled into a platform independent Bytecode. In the first stage we use the Java compiler (javac) which compiles the high level Java program code into intermediate Bytecode (dot class file).
- This Bytecode is a platform independent code that gets further compiled to native machine code. The JVM then executes this platform specific native machine code line by line.

- The Java program is said to be platform independent because the Java program is compiled and executed in two stages.

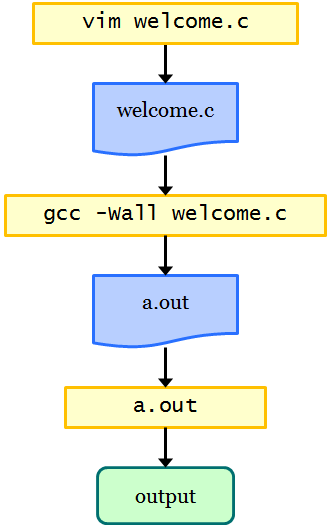


- In the first stage the Java compiler (javac) converts the Java program source code file (YourProgram.java) into a Bytecode (also referred as dot class file YourProgram.class).
- This Java Bytecode is a platform independent executable code that can run on any platform (Windows, Mac, Linux) which has Java Runtime Environment (JRE) installed on it.
- The Java Virtual Machine (JVM) is an essential component of the JRE. The JVM is an interpreter and responsible to execute the Java Bytecode (.class file) line by line.
- And therefore, Java program is platform independent but the JVM part of JRE is platform dependent. And for this reason we need to select the correct version Java during the Java download depending upon the platform.


C++ Program Compilation
C++
C++
EXE
C++
Source Code
C++ Program
Compiler
Executable File
YourProgram.exe
Java Program Compilation
JAVA
Java
Byte Code
JVM
Java Program
Source Code
.java
Java Program
Compiler
javac
Java Compiled
ByteCode
.class
Java Virtual Machine
JVM
Executes .class

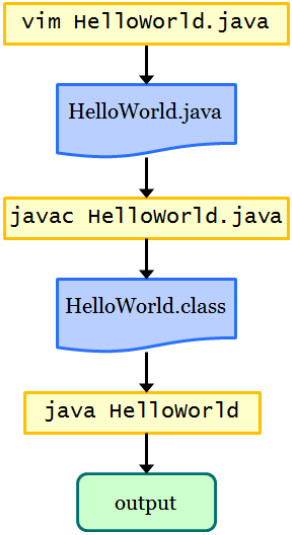

Writing
• Tool: Editor
• Produce: Source Code
Compiling
• Tool: Compiler
• Produce: Executable Bytecode
Executing
• Tool: None
• Produce: Result
Compilation Error
Runtime Error
Logic Error

- Writing/Editing Program
  - Use an editor, e.g. vim
  - Source code must have a .c extension
- Compiling Program
  - Use a C compiler, e.g. gcc
  - Default executable file a.out
- Executing binary
  - Type name of executable file

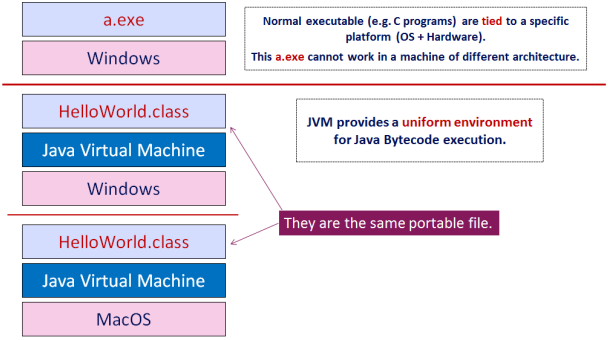


- Writing/Editing Program
  - Use an text editor, e.g. vim
  - Compiled binary has .class extension
- Compiling Program
  - Use a Java compiler, e.g. javac
  - Compiled binary has .class extension
  - The binary is also known as Java executable Bytecode
- Executing binary
  - Run on a Java Virtual Machine (JVM)
    - e.g. java HelloWorld

Note the difference here compared to C executable.



- Normal executable files are directly dependent on the OS/Hardware.
  - Hence, an executable file is usually not executable on different platforms.
  - E.g. the a.out file compiled on Linux is not executable on your Windows computer.



- Java overcomes this by running the executable on an uniform hardware environment simulated by software.
  - The hardware environment is know as the Java Virtual Machine (JVM).
  - So, we only need a specific JVM for a particular platform to execute all Java bytecodes without recompilation.

- In order to develop the Java programs you need first install the Java Development Kit (JDK) on your computer system.
- In addition to JDK you will also need a special software commonly referred as Integrated Development Environment (IDE).

Some of the popular and free Java IDE.

- The IDE provides fully automated environment for Java program development. The IDE provides a complete set of software and tools to write, debug and deploy professional level Java software applications.
- So, initially spend some time on the text editor writing your first few initial Java programs. And once you understand the whole process of how Java program works, then you can always switch over to any suitable IDE.

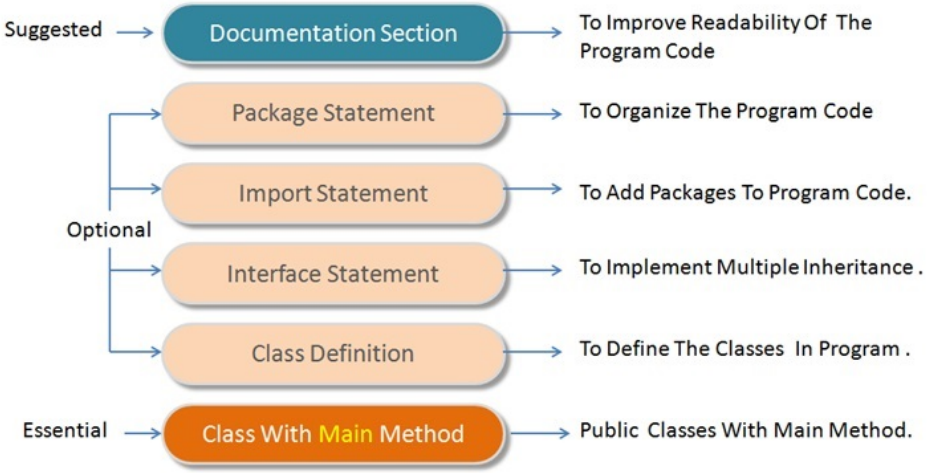

```c
#include <stdio.h>

int main(void) {
    printf("Hello World!\n");

    return 0;
}
```

HelloWorld.c


```java
import java.lang.*;  // optional

public class HelloWorld {

  public static void main(String[] args) {

     System.out.println("Hello World!");

  }

}
```

HelloWorld.java

Beginners' common mistake:
Public class name not identical to program's file name.


Suggested
Documentation Section
To Improve Readability Of The Program Code
Optional
Package Statement
To Organize The Program Code
Import Statement
To Add Packages To Program Code.
Interface Statement
To Implement Multiple Inheritance .
Class Definition
To Define The Classes In Program .
Essential
Class With Main Method
Public Classes With Main Method.

- The documentation section is optional and contains the comments in the java program code. The comments are part of the program that are ignored and not compiled by the compiler.
- The main purpose of the comments in the documentation section is to improve the readability and the maintainability of the java program code.
- The comments are added to the program code especially in the large and complex project where number of programmers are writing the program code. The comments inserted in the program code makes it easier to understand the program code.
- Example:
  - Single Line Comment: // Example Comment Text
  - Multiple Line Comment:
    ```
    /* Example Multi-line Comment Line 1
       This Is comment Line 2 */
    ```
  - Documentation Comment:
    ```
    /** Example Documentation Comment
        Generated Automatically */
    ```

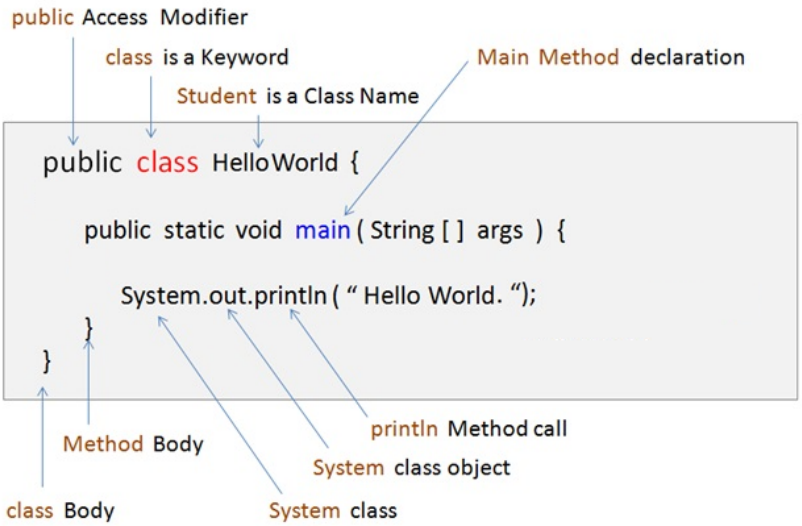
- The Java package allows the programmer to create a group of similar types of classes, interfaces and sub-packages. These packages once created can be easily imported into the program code.
- The Java packages allows the code re-usability and the programmer can simply add the required classes by importing the package that contains these classes. The Java packages can be either built-in packages or user-defined package.
- The package statement in Java program is optional and identifies the package that a Java program is a part of that package. The program belongs to the default package if the Java program does not include a package statement.
- If program does not include the package statement then the program belongs to the default package, which is simply a package without any name.
- Example:

```
package vn.hus.oop;
```

- The Java programmer can use the classes defined in other packages by directly importing the packages into the program code using import statement. The programmer can either import some classes or all the classes present in the package.
- Java import statement is optional. The Java programming language provide number of built-in packages which programmer can use to add additional functionality to the program.
- Once the required package is imported, the classes in the package become available in the program code and can be referred to directly by using only its name. The import statement is very useful and frequently used in the Java program.
- Example:

```
import java.io.* ;
```

- In Java program, all the entities that need to be represented in the program in the form of objects are included as class. And therefore, a Java program may contain several classes defined in the program.
- The class declarations are an important element of the Java program code and the Java program contains number of classes .
- The Java program can contain number of classes. However, the Java program file which will be compiled to generate the Bytecode must have one public class that contains the main method.
- The Java program file must be saved with the same class name that contains the main method. This main method is the starting point of the program execution.


public Access Modifier
class is a Keyword
Student is a Class Name
Main Method declaration
public class HelloWorld {
public static void main (String [ ] args) {
System.out.println (" Hello World. ");
}
}
println Method call
Method Body
System class object
class Body
System class

- Let us now create and understand our first Java program HelloWorld. You can write this program using any text editor (such as Notepad) and then compile this program on the command prompt.
- In order to write and successfully execute the Java program HelloWorld, you need to follow the following steps.
  - Install the JDK on your computer.
  - Set the JDK path with Environment Variable.
  - Write the program in text editor.
  - Save the program file with class name (HelloWorld.java).
  - Compile the program: javac HelloWorld.java
  - This will create Java Bytecode (HelloWorld.class)
  - Execute the program: java HelloWorld
  - This will print: Hello World.

1 Basics of computer programming

2 Java Programming Introduction

3 Java Programming Basics
- Variables In Java
- Data Types in Java
- Keywords in Java
- Operators in Java
- Java Control Statements
- Java Methods

4 Memory Model

5 Parameter Passing Mechanism

- The Java program is essentially a collection of objects which are represented in the program code in the form of a classes. These objects communicate with each other to perform the various tasks as per the program instructions.
- Like any other programming language, the Java program code must be written as per the set of rules and principles that govern the structure of the Java program code and the statements.
- Java is a class based language. And therefore, all the program code is enclosed in a class except the package import statements, documentation statements and package declaration.
- The Java program code can be organized in various packages. The package is a collection of related classes. These pages can be imported into Java program code using import keyword.
- Java Is Strongly Types Language.
  - Java is said to be a strongly typed language because as the Java syntax rules all the variables used in the program code must be declared and the datatype should be specified.

- Java program can have any number of classes. These classes are organized in packages to improve the readability of the program code.
- The class should be declared as per Java syntax rules. The class is declared using the Java keyword class followed by the class name. The initial letter of the class name is capitalized by convention.
- The Java source code file with .java extension can contain only one public class. The Java main method should be declared in this public class which is starting point of program execution. The Java source code file should be saved with the same name as Java public class name.

```java
public class MyJavaProgram {  // Saved as MyJavaProgram.java
    public static void main (String [ ] args) {
        // Main Method Body
    }
}
```

1 Basics of computer programming

2 Java Programming Introduction

3 Java Programming Basics
- Variables In Java
- Data Types in Java
- Keywords in Java
- Operators in Java
- Java Control Statements
- Java Methods

4 Memory Model

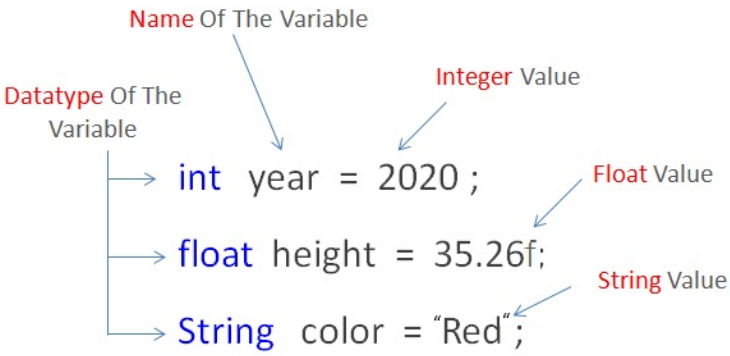
5 Parameter Passing Mechanism

- The data is an important element of the Java program. During the program execution, the data needs to be stored into the memory (RAM) so that the CPU can operate on the data as per the program instructions.
- The variables are named memory locations with specific data type assigned to it which acts as a container for temporarily storing the values in the memory during the program execution.
- The variable is temporary memory space created during the program execution that can store some value as per the data type defined in a program instruction. This memory space can be accessed using a variable name as specified in the program code.
- A variable can be declared with only one data type. Once the variable is declared of a specific data type then during the program execution different values of same data type can be stored into the variable.
- Each data type has specific range of the minimum and maximum value that can be stored into the variable. Depending upon the variable data type the required bytes of memory is allocated by the operating system.

Datatype	Memory Size	Default Value	Type Of Value Stored
bype	1 Byte	0	Integer
short	2 Byte	0	Integer
int	4 Byte	0	Integer
long	8 Byte	0 L	Integer
char	2 Byte	'\u0000'	Character
float	4 Byte	0.0f	Decimal
double	8 Byte	0.0d	Decimal
boolean	1 Bit	false	True Or False

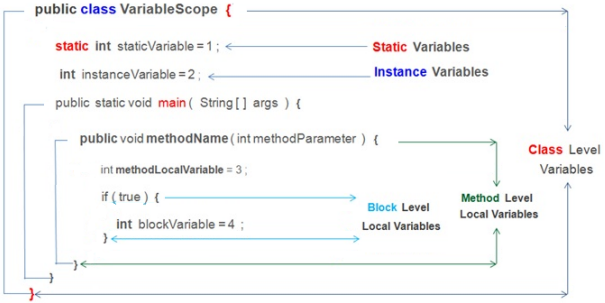
Java Primitive Datatypes And Memory Size

- The variables are used to store the data during the program execution. The variables are referred in the program code by user specified variable names.
- And therefore, the program defines the names to these memory locations and the specific data type is also assigned to these named memory locations. The program data is stored into these named memory locations depending upon the data type.
- The Java program variables must be declared in the program code along with its name and data type. The suitable name is given to the variable as per the variable naming convention (CamelCase).
- The variables are created during the program execution and the operating system allocates the required memory as per the data type of the variable.
- The scope and the visibility of the variable depends upon the type of the variable (local, instance or static variable). The type of the variable depends upon where it is declared within a class and the type of the value assigned to the variable.



Java Variable Declaration

- The variables scope defines the visibility of the Java variable within the program code. The scope of the variable defines the area within which the variable value can be accessed.



- The Java variable is said to be in scope if the variable value can be accessed and operated upon within the segment of the program code.
- The Java variable scope depends upon the location where the variable is declared and the access modifier that precedes the variable name in the variable declaration statement.
- The Java programmer can control the scope (visibility) of the program variables by using the access modifiers (public, private, protected and default) and the location where the variable is declared.

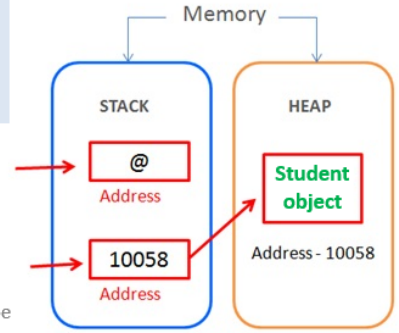
- The Java programming language makes use of different types of variables. The Java variable types can be broadly grouped into two categories based on the type of the value stored into the variable and the scope of the variable.
- The first category of the Java variable type is based on the type of the data stored in a variable. As per this criterion, the Java variable can either be of reference type or it can be of non-reference (primitive) type.
  - Primitive variable
  - Reference variable
- The second category of variable type is based on the scope and the life span of the variable. As per this criterion, the Java variables can be categorized into three types which includes local variable, instance variable and static variable.
  - Local variable
  - Instance variable
  - Static variable

- **Primitive variable**
  - The Java despite being an object-oriented language, the Java supports primitive data type and the programmer is allowed to use the primitive data types in the Java program code.
- **Reference variable**
  - The Java programming language is said to be extensible because you can create a class and declare a variable of that class type. Then we can use the Java new keyword and invoke the constructor to create an object.
  - Once the class is declared, we can create a variable of that class type and such variables are called reference variable. The reference variable can be assigned the address of an object. And therefore, when a reference variable contains the memory location of an object, we say that it refers (points) to an object.
  - In Java, an object variable (that is, a variable whose type is a class) does not actually hold an object. It merely holds the memory location of an object. The object itself is stored elsewhere (Heap).
  - There is a reason for this behavior. Objects can be very large. It is more efficient to store only the memory location instead of the entire object.

- Let us consider one example of a Student class. We can create class Student using Java keyword class. After creating the Student class, now we can create a reference variable Student class (s1) which holds the address of an object and this reference variable (s1) points to the actual object.

```
class Student {
    // Student class declaration
}
```

```
1st Statement : Student s1 ;
// Declare reference variable

2nd Statement :
Studen s1 = new Student ( ) ;
// Create object s1 of Student type
```



- A local variable is said to be a block level variable. The local variables are declared and created within a block. A block is a group of Java statements enclosed within a curly braces. For example variable declaration inside a methods, constructor and blocks.
- A local variable in Java is a variable that is declared within the body of a method or within a instance block or in a constructor body. The local variable once declared can exist only within the method or constructor body or inside a instance block.
- The local variables are not initialized and are not given the initial default values. These variables are created only during the method execution or when the constructor is invoked.

```java
public int createLocalVariable(int argument) {
    int localVariable;
    localVariable = argument;
    return localVariable;
}
```

- The local variables are not given the initial default values and therefore local variables must be first initialized and assigned a value before it is used in any program statement.
- The memory (Stack) is allocated to the local variable when a method is invoked (called) since the local variables are declared inside the method body. The memory to the local variable is taken away and the local variables come to an end when the method execution is over.
- The local variables cannot use any of the access modifiers as they exist only inside the method body. But the local variable can be declared as final using non-access modifier.
- The scope of the local variables is only within a method body and the local variables cannot be accessed and used outside the method, constructor or block body.

```java
public int createLocalVariable(int argument) {
    int localVariable = 1;
}
```

- The instance variables are variables that are declared inside a class but outside the body of a method, constructor or a block without using static keyword. The instance variables belongs to the an object.
- The instance variables are said to be an object level variable. The instance variables belongs to an instance (object) of a class. Each object will have its own copy of the instance variable.
- The instance variables are created when the object is created and destroyed when object is destroyed. The object is created when a class is instantiated by using a Java new keyword and by invoking the constructor. This is then assigned to a reference variable of that class type.
- The instance variable initialization is not compulsory. However, if the instance variable is not initialized then its default value is zero (0). The instance variables can be accessed by creating an object and by using the dot operator with reference to an object created.
- The instance variables for an object defines the state of an object and stores values pertaining to the state of an object. The instance variables can be declared using any one of the four access modifiers (public, private, protected and default).

```java
public class Bicycle {
  // The Bicycle class has three instance variables
  private int cadence;
  private int gear;
  private int speed;

  // The Bicycle class has one constructor
  public Bicycle(int startCadence, int startSpeed, int startGear) {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
  }

  public void setCadence(int newValue) {
    cadence = newValue;
  }

  ...
}

// Create object
Bicyle bicycle = new Bicyle(9, 10, 1);
```

- The static keyword is a non-access modifier in Java that can be used while declaring a method or a variable. The Java keyword static indicates that the method or variable belongs to the class and can be accessed without creating an object of the class.
- The static variables are variables that are declared inside the class but outside the body of a method, constructor or a block using the Java static keyword.
- The static variables are said to be a class variable. The static variables belongs to a class and all the objects share a common copy of the static variable regardless of the number of objects.
- The static variables are created when the JVM loads the Java program dot class (Bytecode) file into the memory for the execution. The static variables are destroyed when the Bytecode execution is over.
- The memory allocation to the static variables happens only once when class is loaded into the memory. It is the JVM which loads the class (Bytecode) into the memory.
- The static variables belong to the class and can be accessed anywhere within the class. And therefore, the static variable cannot be declared inside a method in Java.

Static Variable Example

```
public class Bicycle {
  private static int numberOfBicycles = 0;  // Static variable

  // Three instance variables
  private int cadence;
  private int gear;
  private int speed;

  public Bicycle(int startCadence, int startSpeed, int startGear) {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
  }

  public void setCadence(int newValue) {
    cadence = newValue;
  }

  ...
}

// Create object
Bicyle bicycle = new Bicyle(9, 10, 1);
```

1 Basics of computer programming

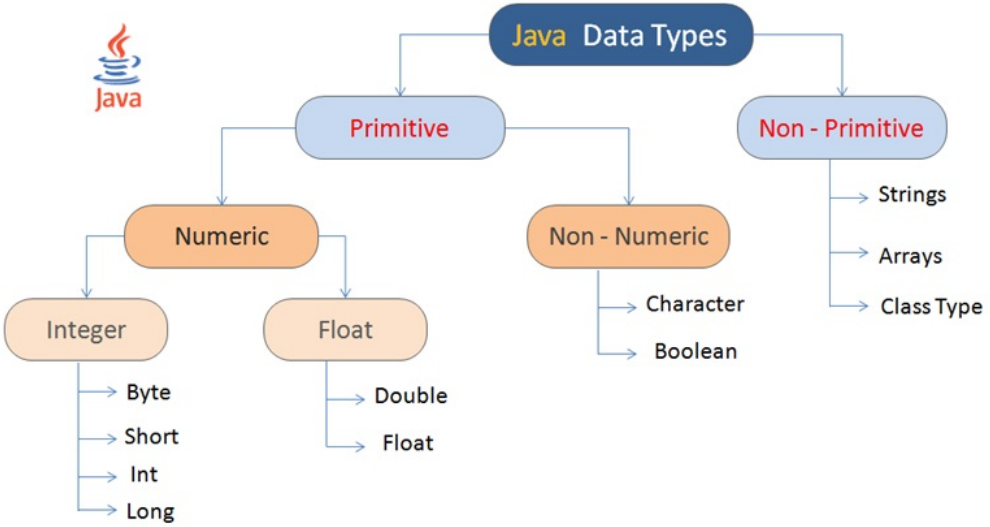
2 Java Programming Introduction

3 Java Programming Basics
- Variables In Java
- Data Types in Java
- Keywords in Java
- Operators in Java
- Java Control Statements
- Java Methods

4 Memory Model

5 Parameter Passing Mechanism

- The variables are created and declared in the Java program to temporarily store the values during the program execution. The programmer also needs to specify the data type of variables in the variable declaration statements.
- Based on the data type of a variable, the operating system allocates the required memory. The data type of the variable decides what kind of values that can be stored into this named memory location.
- For example, the integer type variables can store integer values (such as 4, 7, 23, 1004), float type variables can store floating point type values (such as 3.14, 245.009, 0.7). The string type variables can store String type values (such as "Color", "Red"). The class type reference variables can store object references.
- The data types in Java can be broadly grouped into two types primitive types and non-primitive types. The primitive data types can be further grouped into two types that is numeric and non-numeric. The non-numeric data types include character and boolean type. The non-primitive data types include strings, arrays and class types.

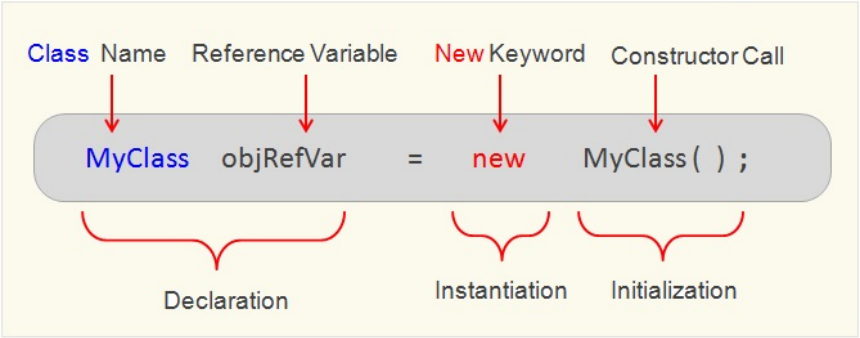

Java
Java Data Types
Primitive
Non - Primitive
Numeric
Non - Numeric
Integer
Float
Byte
Short
Int
Long
Double
Float
Character
Boolean
Strings
Arrays
Class Type

- There are eight primitive data types supported by Java. The primitive data types are predefined by the Java language and each data type is named by a keyword.
- The primitive data types are the most commonly used and a special group of data types that is used extensively in the Java programming. Although Java is object-oriented but supports the primitive data types with the help of wrapper classes.
- The sizes of the primitive data types is the same for all the operating system. This is one of the key features of the Java language that makes Java so portable.

```java
int age = 10;                 // Integer whole number
float distance = 83.7f;       // Float point number
char charDemo = 'J';          // Character data type
boolean booleanDemo = true;   // Boolean data type
String stringDemo = "Java";   // String data type
```

Primitive Type	Size	Minimum Value	Maximum Value	Wrapper Type
char	16-bit	Unicode 0	Unicode 216-1	Character
byte	8-bit	-128	+127	Byte
short	16-bit	-215 (-32,768)	+215-1 (32,767)	Short
int	32-bit	-231 (-2,147,483,648)	+231-1 (2,147,483,647)	Integer
long	64-bit	-263 (-9,223,372,036,854,775,808)	+263-1 (9,223,372,036,854,775,807)	Long
float	32-bit	Approx range 1.4e-045 to 3.4e+038		Float
double	64-bit	Approx range 4.9e-324 to 1.8e+308		Double
boolean	1-bit	true or false		Boolean

- The non-primitive data type in Java include string data types, arrays and class types also referred to as reference data types.
- The Java programming language is extensible because the programmer can create classes and then define a reference variable of that class type. The reference data type do not store the values rather they store the address which points to an object.
- The non-primitive data types are created by programmers as per the need of the program. The non-primitive data types are not predefined in the Java language like primitive data types.
- When we define a reference variable of non-primitive data types, it points to a memory location where data is actually stored in the Heap memory where an object is actually stored.
- For example, we can use arrays when we need to create a large number of variables of the same data type. An array is a single object that contains multiple values of the same data type that can be referred to with the same array name.



objRefVar is a Reference Variable of MyClass Type .

new is a Java Keyword used To Instantiate a Class .

Java reference variable and reference data type

1 Basics of computer programming

2 Java Programming Introduction

3 Java Programming Basics
- Variables In Java
- Data Types in Java
- Keywords in Java
- Operators in Java
- Java Control Statements
- Java Methods

4 Memory Model

5 Parameter Passing Mechanism

Category	Java Keywords
Access Modifiers	private, protected, public
Class, Method, Variable Modifiers	abstract, class, extends, final, implements, interface, native, new, static, strictfp, synchronized, transient, volatile
Flow Control	break, case, continue, default, do, else, for, if, instanceof, return, switch, while
Package Control	import, package
Primitive Types	boolean, byte, char, double, float, int, long, short
Error Handling	assert , catch , finally, throw, throws, try
Enumeration	enum
Others	super, this, void
Unused	const , goto

1 Basics of computer programming

2 Java Programming Introduction

3 Java Programming Basics
- Variables In Java
- Data Types in Java
- Keywords in Java
- Operators in Java
- Java Control Statements
- Java Methods

4 Memory Model

5 Parameter Passing Mechanism

- The operator in Java is a symbol that is used to perform specific operation as defined in the Java language syntax.
- The Java program statement or expression can contain number of operators. An operator in Java is a special symbol that directs the compiler to perform some specific mathematical or non-mathematical operations on one or more operands.
- The Java language provides different types of operators to perform various operations. For example unary operator, arithmetic operator, relational operator, shift operator, bitwise operator, ternary operator and assignment operator.
- The Java language supports eight different types of operators.
  - Arithmetic
  - Uniry
  - Assignment
  - Relational
  - Logical
  - Ternary
  - Bitwise
  - Shift

Java Operator		Examples
1	Arithmetic	+ , – , / , * , %
2	Unary	++ , – – , !
3	Assignment	= , += , -= , *= , /= , %= , ^=
4	Relational	==, != , < , >, <= , >=
5	Logical	&& , ||
6	Ternary	(Condition) **?** (Statement1) **:** (Statement2) ;
7	Bitwise	& , | , ^ , ~
8	Shift	<< , >> , >>>

1 Basics of computer programming

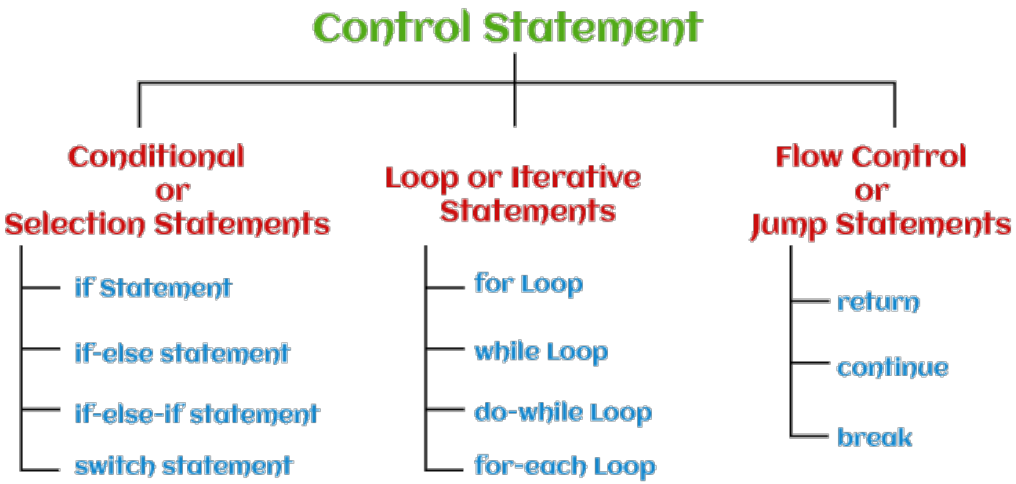
2 Java Programming Introduction

3 Java Programming Basics
- Variables In Java
- Data Types in Java
- Keywords in Java
- Operators in Java
- Java Control Statements
- Java Methods

4 Memory Model

5 Parameter Passing Mechanism

- The Java program is a collection of program statements. As the name suggests, the control statements allows the programmer to control the logical flow of the program as per the program logic.
- The conditional control statements allow to first test the condition (usually an mathematical expression) and then execute the statement block depending upon the outcome of the condition.
- The loop statements are also a type of program statement used in structured programming languages to repeat (reiteration) some program statements for required number of times. The loop will reiterate the statement block till the time the looping condition is evaluated to either true or false.
- Java control statement types:
  - Conditional Control Statements
  - Looping Control Statements
  - Jumping Control Statements


Control Statement
Conditional or Selection Statements
if Statement
if-else statement
if-else-if statement
switch statement
Loop or Iterative Statements
for Loop
while Loop
do-while Loop
for-each Loop
Flow Control or Jump Statements
return
continue
break

- The control statements are used to control the flow of the program. The conditional control statements allow to first test the condition (usually an mathematical expression) and then execute the statement block depending upon the outcome of the condition (either true or false).
- The control flow statements allow the Java program execution either sequentially, or conditionally (with the help of if-else, and switch) or iteratively (with the help of loops) or by using the combination of various control statements depending upon the program logic.
- Conditional statement include:
  - if statement
  - if-else Statement
  - Nested if-else statement
  - switch statement

- The Java looping statements are used to repeat the program statement or a block of statements code either certain number of times or till the time the looping condition is evaluated to either true or false.
- The Java programming language provides three types of looping constructs which can be used by the programmer whenever a block of statements is required to be executed number of times.
- The Java looping control statements include:
  - while loop statement
  - do-while loop statement
  - for loop statement

- A while loop is a control flow statement that allows the block of statements to be executed repetitively based on a given boolean condition is evaluated to either true or false.
- A while loop provides a exit mechanism such as loop counter variable which will keep the track of the number of iterations after being incremented or decremented and the loop will be terminated once condition is evaluated to false.
- The while loop will always evaluate the looping condition first and then execute the code if the boolean condition is evaluated to true.

```
while (expression) {
   statement(s)
}
```

- A do-while loop is a control flow statement that allows the block of statements to be executed repetitively based on a given boolean condition is evaluated to either true or false.
- The do-while loop will always first execute the statements in the loop body and then it will test the looping condition.
- The do-while execute the code at least once. The do while loop will continue the repeated execution till boolean condition is evaluated to true. The loop will exit when the boolean condition is evaluated to false.

```
do {
    statement(s)
} while (expression);
```

- A do while loop must provide a exit mechanism such as loop counter variable which will keep the track of the number of iterations after being incremented or decremented.
- The main difference between while loop and do-while loop is the order of execution of the loop condition. The while loop always first test the loop condition whereas the do-while loop will always first execute the statements in the loop body and then test the loop condition.

```java
class DoWhileDemo {
  public static void main(String[] args) {
    int count = 1;
    do {
      System.out.println("Count is: " + count);
      count++;
    } while (count < 11);
  }
}
```

- The for loop will keep executing the code within its body for the specified number of times as defined in the looping condition expression and will continue till the condition expression is evaluated to true.
- The control will exit the for loop either when the looping condition expression is evaluated to false or the code executes a break (keyword) statement which terminates the for loop.
- The for loop is mainly used in the program code where the number of loop iterations are known before the loop starts and the loop counter is defined in the condition expression to track the current number of the iteration completed by the for loop.

```
for (initialization; condition; update) {
  statement(s)
}
```

How Java For Loop Works?

1. The for loop starts with first initialization of the loop counter variable which is executed only once in the beginning of the loop.
2. The loop condition expression is then evaluated with current value of the loop counter.
3. The statements within loop body are executed if the loop condition boolean expression result to true.
4. The control is sent to the loop counter update section (increment / decrement) where the counter is modified.
5. The control then again checks the loop condition with reference to the current value of the loop counter variable.
6. The loop will exit if condition loop expression is evaluated to false.

1 Basics of computer programming

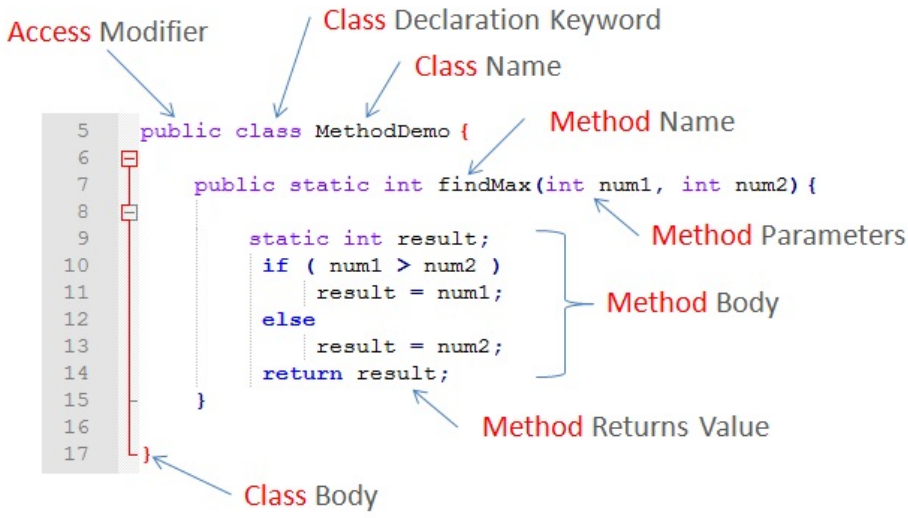
2 Java Programming Introduction

3 Java Programming Basics
- Variables In Java
- Data Types in Java
- Keywords in Java
- Operators in Java
- Java Control Statements
- Java Methods

4 Memory Model

5 Parameter Passing Mechanism

- The Java program contains classes and a class contains member variables and functions (methods). A method is a named block of code which performs a specific task as per the program statements.
- The variables within a class defines the state of an object whereas the methods defines the behavior of an object. The methods operate on the data (variables) to perform various operations.
- The method needs to be explicitly called (invoked) in the program code along with the arguments. The methods are also alternately referred as functions in other language but both means the same.
- The Java programmer can make use of either built-in methods or user defined methods created as per the program needs.
- The built-in methods are part of the classes that program can readily use by importing the package which contains the class in which these methods have been defined.


Access Modifier
Class Declaration Keyword
Class Name
Method Name
Method Parameters
Method Body
Method Returns Value
Class Body
5 public class MethodDemo {
6
7 public static int findMax(int num1, int num2) {
8
9 static int result;
10 if (num1 > num2)
11 result = num1;
12 else
13 result = num2;
14 return result;
15 }
16
17 }

- The methods can be classified in Java in number of ways. The Java methods can either be standard library methods (built-in methods) or user defined methods.
- The Java languages provides extensive library of standard methods which are predefined in the Java language. The programmer can simply import the package which contains the class in which the method is defined to use in the program.
- The programmer can also create and define a method as per the program requirements. The programmer can use these methods by organizing the classes containing these methods into various packages.
  - Pre-defined (built-in) methods
  - User-defined methods

- The Java methods can also be classified on the basis of scope of the method into two types such as instance method and static methods.
- **Instance method**
  - The programmer can also create a method by declaring a method within a class body. The method declared inside a class is referred as instance method. Which means each object (instance) of the class will have this method.
  - The instance method can be called (invoked) by creating an object of the class and by using the object reference and the dot operator.
- **Static method**
  - A method can also be declared at the class level by using the java static keyword. The static method will be one single method common for all objects. The static method can access only static member variables within a class.
  - The static method can be called (invoked) without creating an object of the class and by using the class name and the dot operator. The main method is an example of the static method.

```java
public class Calculator {
  public static int add(int a, int b) {
    return a + b;
  }

  public static int subtract(int a, int b) {
    return a - b;
  }

  public static int multiply(int a, int b) {
    return a * b;
  }

  public static int divide(int a, int b) {
    return a / b;
  }

  public static void main(String[] args) {
    ...
  }
}
```

```java
public class Calculator {
  ...

  public static void main(String[] args) {
    int a = 20;
    int b = 4;

    int result = add(a, b);
    // result is 24

    result = subtract(a, b);
    // result is 16

    result = multiply(a, b);
    // result is 80

    result = divide(a, b);
    // result is 5
  }
}
```

1 Basics of computer programming

2 Java Programming Introduction

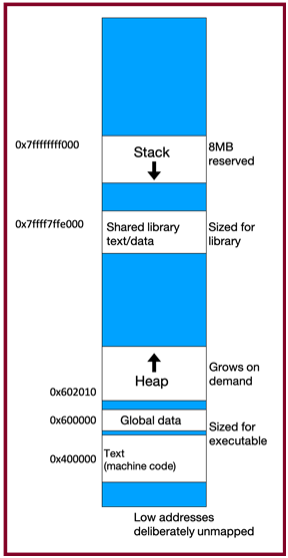
3 Java Programming Basics

4 Memory Model
- Stack
- Heap
- Variables and Program Memory in Java

5 Parameter Passing Mechanism

6 References

- We are going to dive deeper into different areas of memory used by a computer program.
- The Stack is the place where all local variables and parameters live for each function. A function's Stack frame goes away when the function returns.
- The Stack grows downwards when a new function is called, and shrinks upwards when the function is finished.



1 Basics of computer programming

2 Java Programming Introduction

3 Java Programming Basics

4 Memory Model
- Stack
- Heap
- Variables and Program Memory in Java

5 Parameter Passing Mechanism

6 References

- Stack is an abstract data type, a linear data structure that holds a collection of elements that are added or removed in a Last In First Out (LIFO) manner. This means the element added at the top will be removed first, just like we have a pile of plates one on top of the other and you will take out the plate at the top first.

  Stack supports two main operations:

  - **Push**: Adding a data item to the array or list.
  - **Pop**: Removing the most recently added data item.

  The size of the stack gets adjusted based on Push or Pop operation.

- Stack memory gets allocated to adjacent blocks or chunks of memory. Since this allocation of memory happens in a function called Stack, it is named as Stack memory allocation.
- Therefore, the popular use of Stack at the architecture level is memory allocation. A Stack is that part of a computer memory that is used for contiguous, temporary memory allocation. Stack has a fixed starting location, but variable size.
- This memory allocation is temporary in nature and stores local variables along with those arguments which are passed through a function along with their return addresses. All the data belonging to the function which completes execution is quickly removed from the Stack.
- What this really means is that the value stored in Stack memory is available only until the execution is still running and it will automatically erase the Stack memory after the task is completed.

Stack Memory

- The Stack Memory is an important component in the memory management of Java programs. The Stack Memory in Java is used for static memory allocation and the execution of a thread. It contains primitive values that are specific to a method and references to objects referred from the method that are in a Heap.
- Whenever we call a new method, a new block is created on top of the Stack which contains values specific to that method, like primitive variables and references to objects. These local variables are declared inside the method they're pushed onto the Stack. These local variables pertaining to a given method are grouped together in what's called a Stack Frame.
- When a method finishes executing, the corresponding Stack Frame is popped from the Stack, meaning all of the variables contained within are removed together and become unavailable, and space becomes available for the next method.
- This results in the Stack Frame from the previous method being at the top of the Stack, and, consequently, the local variables it contains being in scope - this is how variable scope is managed in Java.

Key Features of Stack Memory

- It grows and shrinks as new methods are called and returned, respectively.
- Variables inside the stack exist only as long as the method that created them is running.
- It's automatically allocated and deallocated when the method finishes execution.
- If this memory is full, Java throws java.lang.StackOverFlowError.
- Access to this memory is fast when compared to Heap memory.
- This memory is threadsafe, as each thread operates in its own Stack.

Benefits

- Stack memory offers multiple advantages to the programmer at the time of compilation of the code.
- The process of allocation and deallocation of memory can be controlled.
- The capability to manage data in LIFO (Last In First Out) gives the stack an edge over the heap.
- Stack offers auto clean-up objects in memory and variables cannot be resized.
- The local variables are stored in the “called function” in the stack and are quickly terminated on return.

Drawbacks

- Stack has a limited size for memory, which makes it unsuitable in case of the requirement of large memory size.
- During the compilation of code, stack overflow can happen if the number of objects exceeds the size of the stack.

1 Basics of computer programming

2 Java Programming Introduction

3 Java Programming Basics

4 Memory Model
- Stack
- Heap
- Variables and Program Memory in Java

5 Parameter Passing Mechanism

6 References

Heap Space

- Heap space is used for the dynamic memory allocation of Java objects and JRE classes at runtime. New objects are always created in Heap space, and the references to these objects are stored in Stack memory.
- These objects have global access and we can access them from anywhere in the application.
- We can break this memory model down into smaller parts, called generations, which are:
  1. **Young Generation** – this is where all new objects are allocated and aged. A minor Garbage collection occurs when this fills up.
  2. **Old or Tenured Generation** – this is where long surviving objects are stored. When objects are stored in the Young Generation, a threshold for the object's age is set, and when that threshold is reached, the object is moved to the old generation.
  3. **Permanent Generation** – this consists of JVM metadata for the runtime classes and application methods.
- The Garbage Collector (GC) is a program that manages the objects on the Heap, it deletes objects that are no longer being used - freeing memory for future allocations. The GC frees programmers of these memory management responsibilities.

- The allocation of Heap memory takes place at the time of execution of the instructions of the programmer. The term Heap refers to a collection of memory that can be allocated and deallocated by the programmer. Therefore, the Heap has no relation to the heap data structure.
- It is important to understand that while the construction of an object happens in a Heap, however, the corresponding information for these objects is saved in Stack memory. Heap memory often suffers from security issues due to the visibility and accessibility of data stored in all threads.
- This, sometimes, can lead to a situation of memory leak in the application if the programmer misses handling Heap memory with care.
- In Heap, data is stored in a hierarchical manner which leads to slow access as compared to Stack. Do you remember how the old platter hard drives used to get clogged due to fragmentation? Something similar happens with Heap memory as well. Fragmentation leads to the clogging of Heap memory.
- One of the major advantages of heap Hemory lies in the fact that there is no limitation on the size of the memory and it also allows for resizing of variables as and when needed.
- Heap memory is stored randomly, and that explains the slow speed of access because the data will have to be pulled from multiple random places on the chip.

Key Features of Java Heap Memory

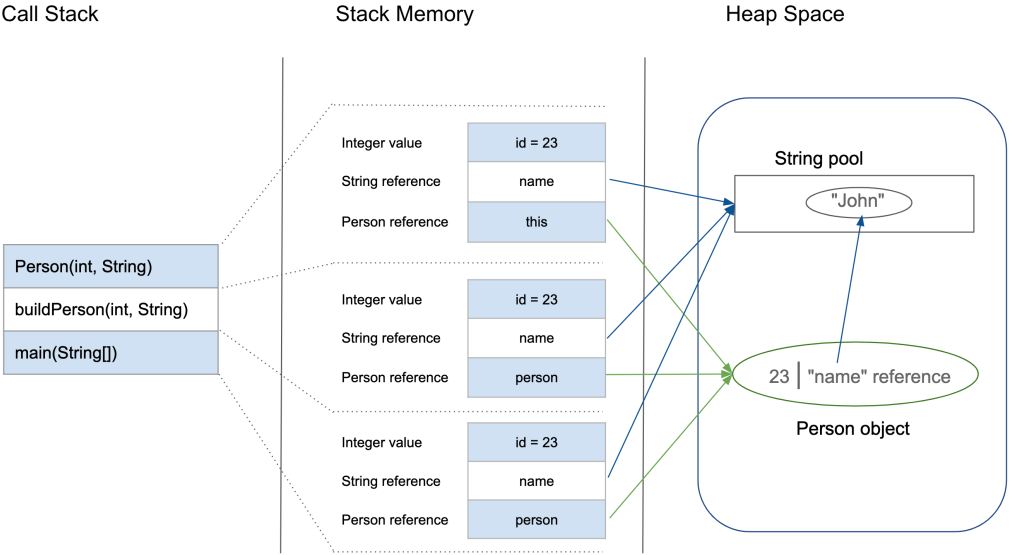
- It's accessed via complex memory management techniques that include the Young Generation, Old or Tenured Generation, and Permanent Generation.
- If Heap space is full, Java throws java.lang.OutOfMemoryError.
- Access to this memory is comparatively slower than Stack memory.
- This memory, in contrast to Stack, isn't automatically deallocated.
- It needs Garbage Collector to free up unused objects so as to keep the efficiency of the memory usage.
- Unlike Stack, a Heap isn't threadsafe and needs to be guarded by properly synchronizing the code.

- **Advantages**
  - The heap does not have any limitation on the size of memory. This feature gives the Heap an added advantage over the Stack.
  - The variables in Heap memory can be accessed globally and can also be resized based on requirements.
- **Disadvantages**
  - As compared to the Stack, Heap not only has a slower execution time but also the management of memory is a complicated process.
  - The computation process is also slow as compared to the stack. Continuous use of heap memory can consume all the RAM from the computer.

```java
class Person {
  private int id;
  private String name;

  public Person(int id, String name) {
    this.id = id;
    this.name = name;
  }
}

public class PersonBuilder {
  private static Person buildPerson(int id, String name) {
    Person person = new Person(id, name);
    return person;
  }

  public static void main(String[] args) {
    int id = 23;
    String name = "John";
    Person person = null;
    person = buildPerson(id, name);
  }
}
```

1. When we enter the main() method, a space in Stack memory is created to store primitives and references of this method.
   - Stack memory directly stores the primitive value of integer id.
   - The reference variable person of type Person will also be created in Stack memory, which will point to the actual object in the Heap.
2. The call to the parameterized constructor Person(int, String) from main() will allocate further memory on top of the previous Stack. This will store:
   - The this object reference of the calling object in Stack memory.
   - The primitive value id in the Stack memory.
   - The reference variable of String argument name, which will point to the actual string from String Pool in Heap memory.
3. The main method is further calling the buildPerson() static method, for which further allocation will take place in Stack memory on top of the previous one. This will again store variables in the manner described above.
4. However, Heap memory will store all instance variables for the newly created object person of type Person.

Java Program Structure Example



Parameter	Stack Memory	Heap Space
Cost of usage	Low	High
Deallocation of variable	Not necessary	Clear deallocation is important
Access time	Quick	Slow as compared to stack
Resizing required	Variable size is fixed	Resizing of variable is possible
Drawbacks	Limited memory	Fragmented memory due to allocation and later freeing up of memory
Allocation of memory	Automatic	Manual
Order for allocation of memory	Allocated as a contiguous block	Allocated in a random order
Size of memory	Limited memory and dependent on OS	Unlimited memory size

1 Basics of computer programming

2 Java Programming Introduction

3 Java Programming Basics

4 Memory Model
- Stack
- Heap
- Variables and Program Memory in Java

5 Parameter Passing Mechanism

6 References

- A computer program has memory allocated to it by the OS that is used to hold the data that the program uses (i.e. to store variables) and other things.
- We can think of that memory as being divided into different parts: the Stack, the Heap, and everything else in the program's memory (like the program's machine instructions).
- The Stack and the Heap store the values of the program's variables – i.e. they store the configuration of bits that a variable refers to. Thinking in this way: a variable is essentially an association between a name in the source code and a block of memory which either lives on the Stack or the Heap, with the contents of the block (some configuration of bits) being the value of the variable.
- Each Java variable has a specific type, and the contents of the memory block associated with a given variable depend on its type.
- Java variables fall into two distinct type categories – value types and reference types, which are handled differently in memory. The type of a variable – specifically whether it's a reference or value type – and the context in which it was declared, determine whether it is stored on the Stack or Heap.

```java
int x = 2;                      // Value type variable
Car car = new Car("Audi");      // Reference type variable
```

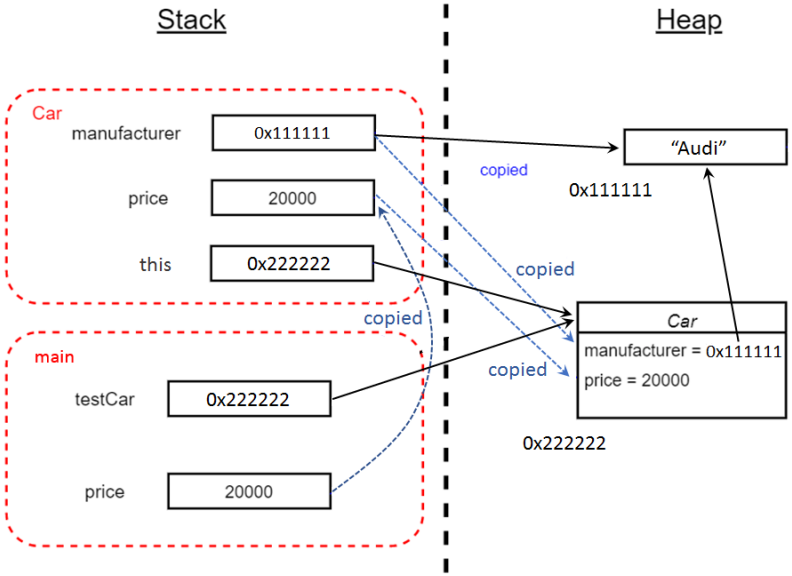
- **Value types**
  - A value type variable does hold the value to which it is associated.

- **Value types**
  - The value of a reference type is a reference (or null), meaning the value of a reference type is the memory address of the object to which it refers. So, a reference type variable does not hold the value of the object it refers to, it holds a reference to that object.

Where do Java variables get stored?

- Java variables are stored on either the Stack or Heap, which one depends on whether the variable is of reference or value type, and on the context in which the variable is declared.
- Local variables (i.e. those that are declared inside methods) are stored on the Stack. This means their values are stored on the Stack, therefore meaning that local reference type variables have references stored on the Stack and local value type variables have actual values stored on the Stack.
- Objects of reference type variables (i.e. the things that references point to) always live on the Heap.
- Instance variables that are part of a reference type instance (e.g. a field on a class) are stored on the Heap with the object itself.
- Instance variables that are part of a value type instance are stored in the same context as the variable that declares the value type. This means that a variable that is declared in a method will live on the Stack, whilst a variable that is declared inside a class (i.e. a field on the class) will live on the Heap.

```java
public class Car {
  private String manufacturer;
  private int price;

  public Car(String manufacturer, int price) {
    this.manufacturer = manufacturer;
    this.price = price;
  }

  public static void main(String[] args) {
    int price = 20000;
    Car testCar = new Car("Audi", price);
  }
}
```

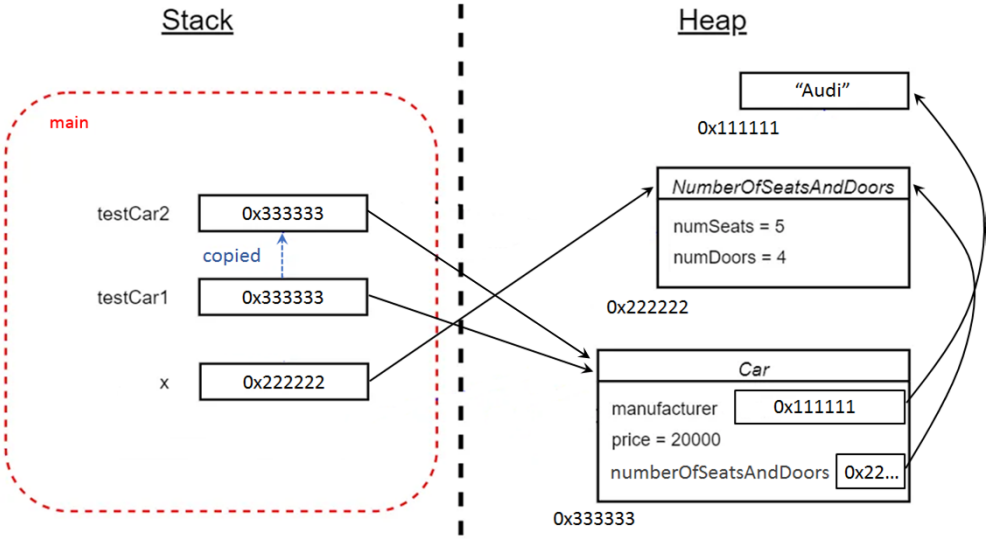

Stack
Heap
Car
manufacturer
0x111111
price
20000
this
0x222222
main
testCar
0x222222
price
20000
copied
copied
copied
copied
"Audi"
0x111111
Car
manufacturer = 0x111111
price = 20000
0x222222

- In the main() method we're declaring and initializing a local variable, price, of type int (value type), we're then declaring a variable of type Car (reference type), testCar, and initializing it with a reference to a Car object. price is a local variable of value type, therefore it will live on the Stack; testCar is a local variable of reference type, therefore it will live on the Stack, but, again, it's a reference type and so the thing stored on the Stack is a reference (i.e. a memory address), as depicted in the diagram above. The actual Car object that the reference in testCar points to lives on the Heap.
- When the right-hand side of Car testCar = new Car("Audi", price) runs, the constructor to the Car class runs, which is a method and so a new Stack frame is added to the Stack to hold the variables for that method. The Car constructor is being called with the price variable as an argument for the price parameter, therefore the int value from price (in main() Stack frame) is getting copied into the price variable in the Car Stack frame (method parameters are local variables too).
- There is also the manufacturer parameter on the constructor, which is being passed a new string instance directly inside the call to the Car constructor, therefore there is a variable, manufacturer, of type String on the Stack, living in the Car Stack frame. The Car constructor then copies the values of these two variables from the Car Stack frame into the corresponding fields on the Car object on the Heap.

Example of Allocating Object That Contains Another Object

```java
public class NumberOfSeatsAndDoors {
  private int numSeats;
  private int numDoors;

  public NumberOfSeatsAndDoors(int numSeats, int numDoors) { ... }
}

public class Car {
  private String manufacturer;
  private int price;
  private NumberOfSeatsAndDoors numberOfSeatsAndDoors;

  public Car(String manufacturer, int price, NumberOfSeatsAndDoors numberOfSeatsAndDoors) {
    this.manufacturer = manufacturer;
    this.price = price;
    this.numberOfSeatsAndDoors = numberOfSeatsAndDoors;
  }

  public static void main(string[] args) {
    NumberOfSeatsAndDoors x = new NumberOfSeatsAndDoors(5, 4);
    Car testCar1 = new Car("Audi", 20000, x);
    Car testCar2 = testCar1;
  }
}
```

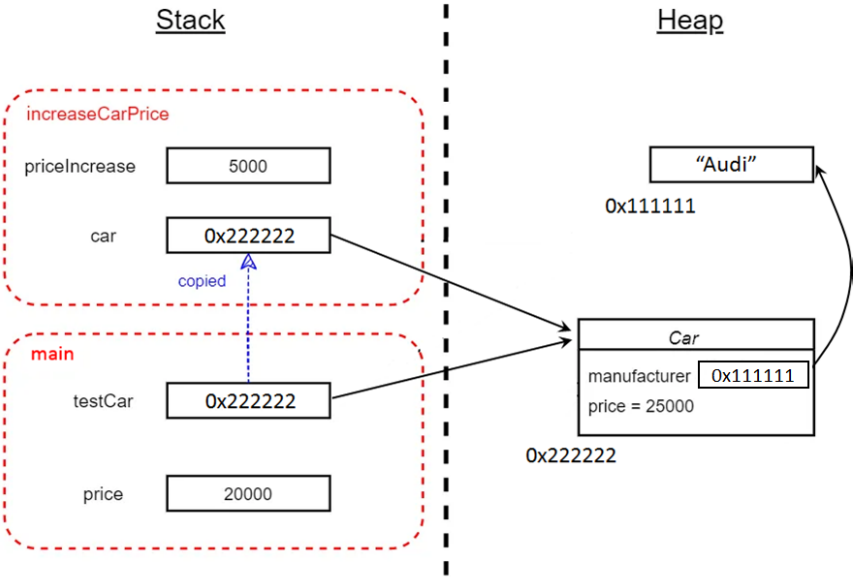
Example of Allocating Object That Contains Another Object



- In the main() method we're creating an instance of the class type NumberOfSeatsAndDoors and assigning it into variable x, which is a local variable of reference type and will therefore live on the Stack.
- Next, we're creating an instance of Car and assigning it into the local variable testCar1, this is a local variable of reference type and will therefore live on the Stack and contain a reference to the Car object, which lives on the Heap.
- To build the Car object, the variable x is being passed to the constructor; x is a reference type and therefore its value – a reference of NumberOfSeatsAndDoors instance - will be copied from the Stack into the numberOfSeatsAndDoors field on the Car object.
- Finally, another variable of type Car, testCar2, is being declared and initialised with the value of testCar1. These two variables are reference types, therefore the reference that testCar1 holds is being copied into the testCar2 variable, resulting in them both referring to the same Car object on the Heap.

```java
public class Car {
  public String manufacturer;
  public int price;

  public Car(String manufacturer, int price) {
    this.manufacturer = manufacturer;
    this.price = price;
  }
}

public class Program {
  public static void main(string[] args) {
    int price = 20000;
    Car testCar = new Car("Audi", price);
    inreaseCarPrice(testCar, 5000);
  }

  public static void increaseCarPrice(Car car, int priceIncrease) {
    car.price += priceIncrease;
  }
}
```

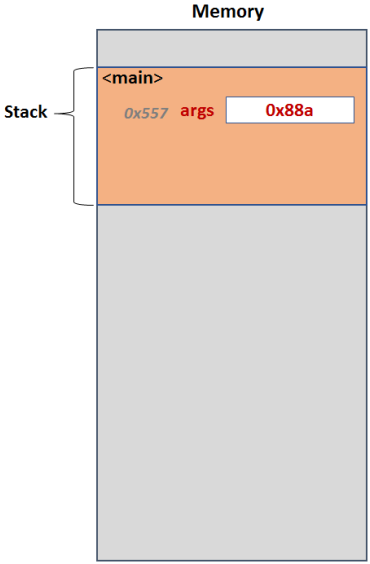
Example of Allocating Memory When Calling a Funtion



- We're creating an int and assigning it into a local variable price, and declaring a variable, testCar, and initializing it with a reference to a Car object, both of these variables live on the Stack, whilst the Car object lives on the Heap.
- Next, we're calling the increaseCarPrice() method, which results in a new Stack frame being pushed onto the Stack to contain the local variables of that method. testCar is being passed as an argument for the car parameter, meaning the reference that it holds is being copied into the local variable car in the Stack frame of the increaseCarPrice() method; a new int of value 5000 is being created and assigned into the priceIncrease parameter. As a result, the Stack frame associated with increaseCarPrice() contains an integer with value 5000 and a reference pointing to a Car object on the Heap. The line of code in the increaseCarPrice() method is then being executed which changes the price field on the Car object on the Heap.
- When the program reaches the closing curly brace of the increaseCarPrice() method, the corresponding Stack frame is popped off the Stack, resulting in the earlier Stack frame - corresponding to the main() method - being at the top and its contents being in scope.

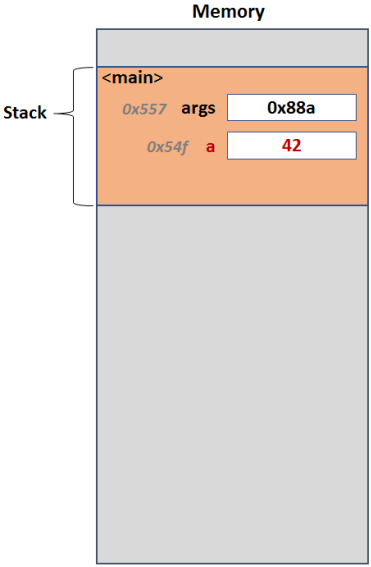
Example of Stack Memory Allocation

```java
public class StackDemo {
  public static void main(String[] args) {
    int a = 42;
    int b = 17;
    func1();
    System.out.println("Done.");
  }

  public static void func1() {
    int c = 99;
    func2();
  }

  public static void func2() {
    int d = 0;
  }
}
```



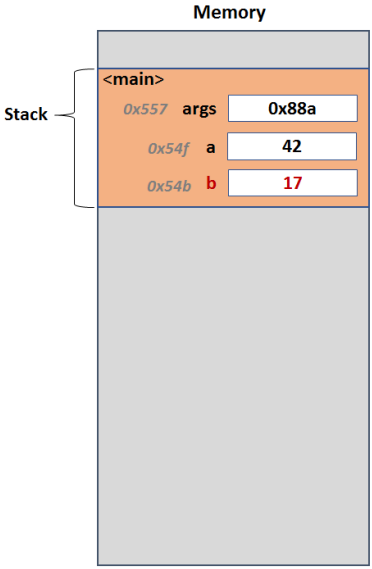
Example of Stack Memory Allocation

```java
public class StackDemo {
  public static void main(String[] args) {
    int a = 42;
    int b = 17;
    func1();
    System.out.println("Done.");
  }

  public static void func1() {
    int c = 99;
    func2();
  }

  public static void func2() {
    int d = 0;
  }
}
```



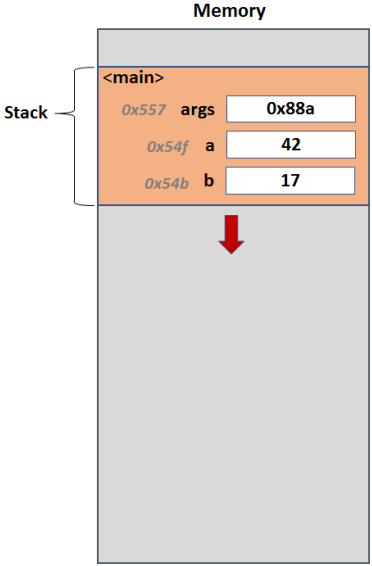
Example of Stack Memory Allocation

```java
public class StackDemo {
  public static void main(String[] args) {
    int a = 42;
    int b = 17;
    func1();
    System.out.println("Done.");
  }

  public static void func1() {
    int c = 99;
    func2();
  }

  public static void func2() {
    int d = 0;
  }
}
```



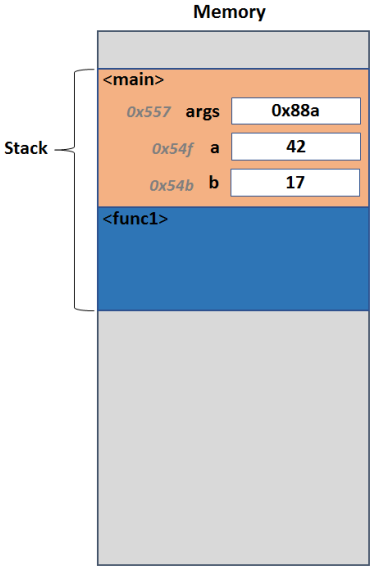
Example of Stack Memory Allocation

```java
public class StackDemo {
  public static void main(String[] args) {
    int a = 42;
    int b = 17;
    func1();
    System.out.println("Done.");
  }

  public static void func1() {
    int c = 99;
    func2();
  }

  public static void func2() {
    int d = 0;
  }
}
```



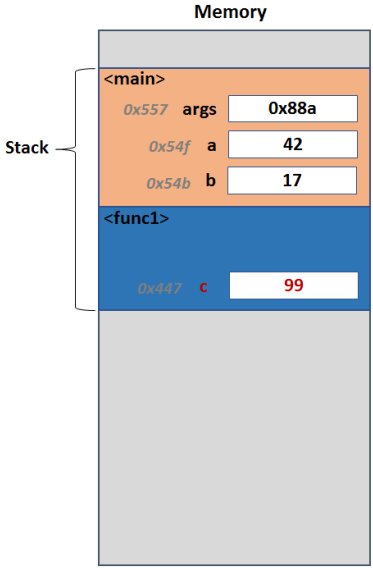
Example of Stack Memory Allocation

```java
public class StackDemo {
  public static void main(String[] args) {
    int a = 42;
    int b = 17;
    func1();
    System.out.println("Done.");
  }

  public static void func1() {
    int c = 99;
    func2();
  }

  public static void func2() {
    int d = 0;
  }
}
```



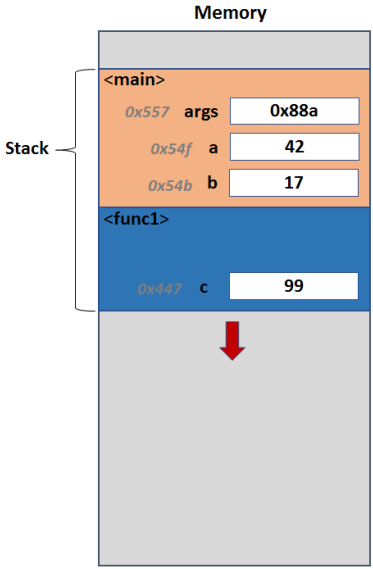
Example of Stack Memory Allocation

```java
public class StackDemo {
  public static void main(String[] args) {
    int a = 42;
    int b = 17;
    func1();
    System.out.println("Done.");
  }

  public static void func1() {
    int c = 99;
    func2();
  }

  public static void func2() {
    int d = 0;
  }
}
```



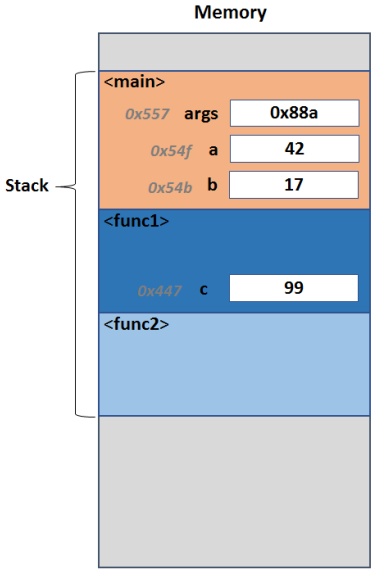
Example of Stack Memory Allocation

```java
public class StackDemo {
  public static void main(String[] args) {
    int a = 42;
    int b = 17;
    func1();
    System.out.println("Done.");
  }

  public static void func1() {
    int c = 99;
    func2();
  }

  public static void func2() {
    int d = 0;
  }
}
```



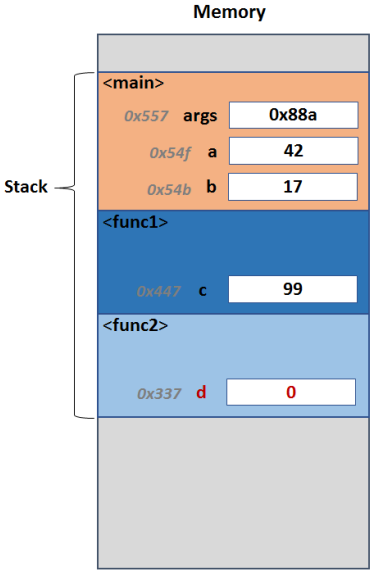
Example of Stack Memory Allocation

```java
public class StackDemo {
  public static void main(String[] args) {
    int a = 42;
    int b = 17;
    func1();
    System.out.println("Done.");
  }

  public static void func1() {
    int c = 99;
    func2();
  }

  public static void func2() {
    int d = 0;
  }
}
```



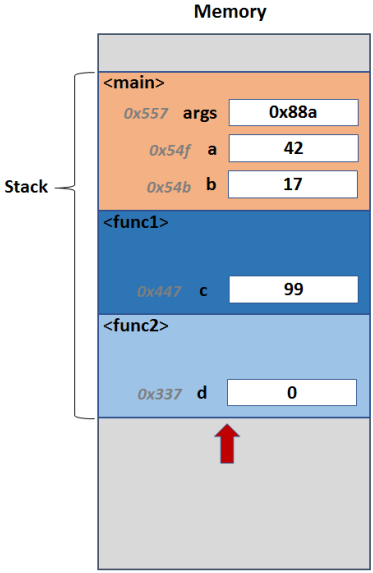
Example of Stack Memory Allocation

```
1  public class StackDemo {
2    public static void main(String[] args) {
3      int a = 42;
4      int b = 17;
5      func1();
6      System.out.println("Done.");
7    }
8
9    public static void func1() {
10     int c = 99;
11     func2();
12   }
13
14   public static void func2() {
15     int d = 0;
16   }
17 }
```



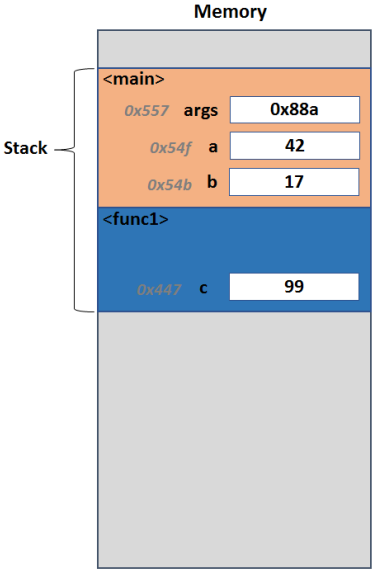
Example of Stack Memory Allocation

```
1  public class StackDemo {
2    public static void main(String[] args) {
3      int a = 42;
4      int b = 17;
5      func1();
6      System.out.println("Done.");
7    }
8
9    public static void func1() {
10     int c = 99;
11     func2();
12   }
13
14   public static void func2() {
15     int d = 0;
16   }
17 }
```



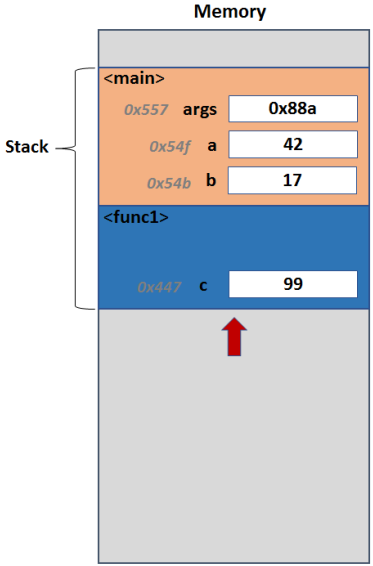
Example of Stack Memory Allocation

```java
public class StackDemo {
  public static void main(String[] args) {
    int a = 42;
    int b = 17;
    func1();
    System.out.println("Done.");
  }

  public static void func1() {
    int c = 99;
    func2();
  }

  public static void func2() {
    int d = 0;
  }
}
```



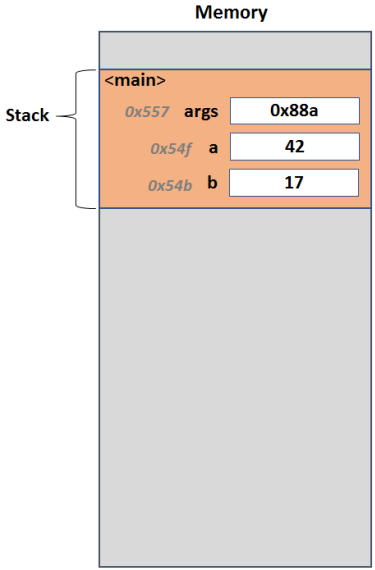
Example of Stack Memory Allocation

```java
public class StackDemo {
  public static void main(String[] args) {
    int a = 42;
    int b = 17;
    func1();
    System.out.println("Done.");
  }

  public static void func1() {
    int c = 99;
    func2();
  }

  public static void func2() {
    int d = 0;
  }
}
```



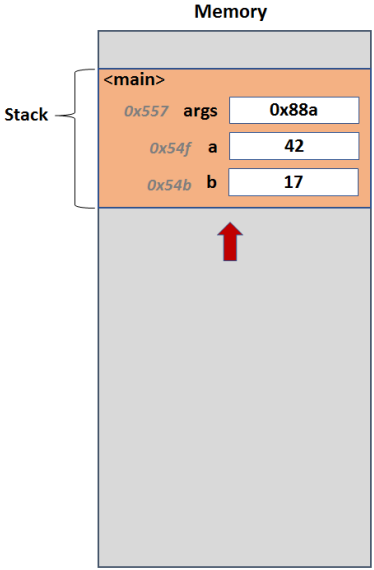
Example of Stack Memory Allocation

```java
public class StackDemo {
  public static void main(String[] args) {
    int a = 42;
    int b = 17;
    func1();
    System.out.println("Done.");
  }

  public static void func1() {
    int c = 99;
    func2();
  }

  public static void func2() {
    int d = 0;
  }
}
```



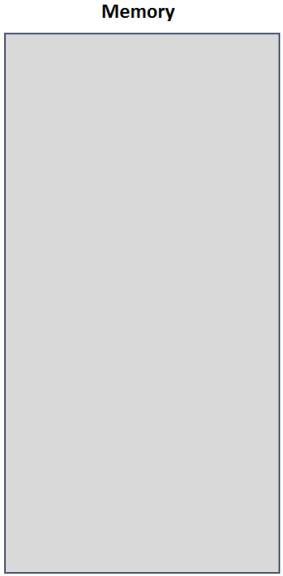
Example of Stack Memory Allocation

```java
public class StackDemo {
  public static void main(String[] args) {
    int a = 42;
    int b = 17;
    func1();
    System.out.println("Done.");
  }

  public static void func1() {
    int c = 99;
    func2();
  }

  public static void func2() {
    int d = 0;
  }
}
```



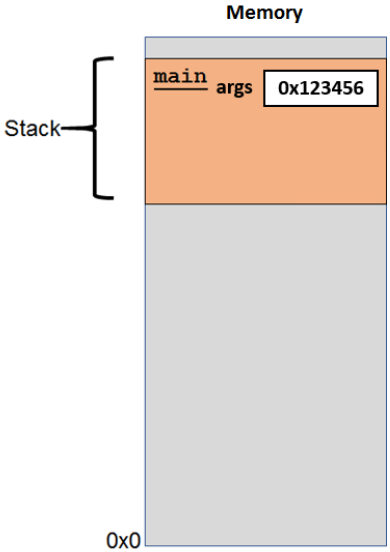
Example of Stack Memory Allocation

```java
public class StackDemo {
  public static void main(String[] args) {
    int a = 42;
    int b = 17;
    func1();
    System.out.println("Done.");
  }

  public static void func1() {
    int c = 99;
    func2();
  }

  public static void func2() {
    int d = 0;
  }
}
```

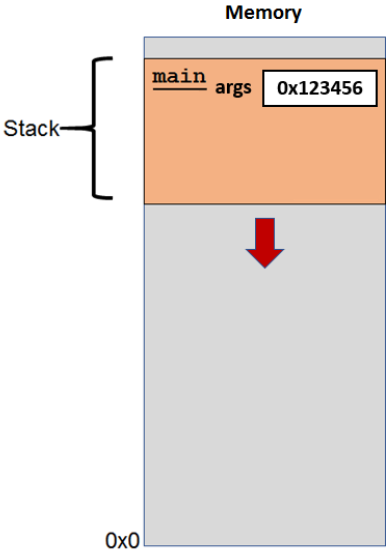


Example of Stack Memory Allocation with Recursive Calls

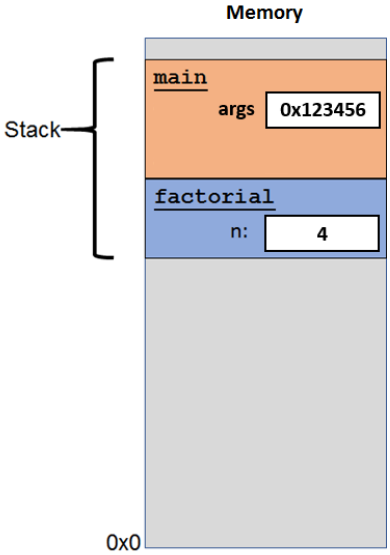
```java
public class FactorialDemo {
    public static void main(String[] args) {
        System.out.println(factorial(4));
    }

    public static int factorial(int n) {
        if (n == 1) {
            return 1;
        }

        return n * factorial(n - 1);
    }
}
```



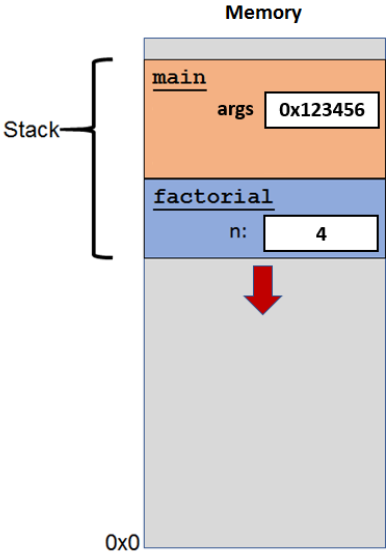
```java
public class FactorialDemo {
    public static void main(String[] args) {
        System.out.println(factorial(4));
    }

    public static int factorial(int n) {
        if (n == 1) {
            return 1;
        }

        return n * factorial(n - 1);
    }
}
```



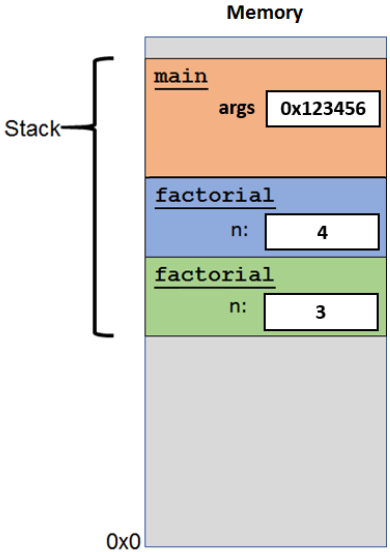
```java
public class FactorialDemo {
  public static void main(String[] args) {
    System.out.println(factorial(4));
  }

  public static int factorial(int n) {
    if (n == 1) {
      return 1;
    }

    return n * factorial(n - 1);
  }
}
```



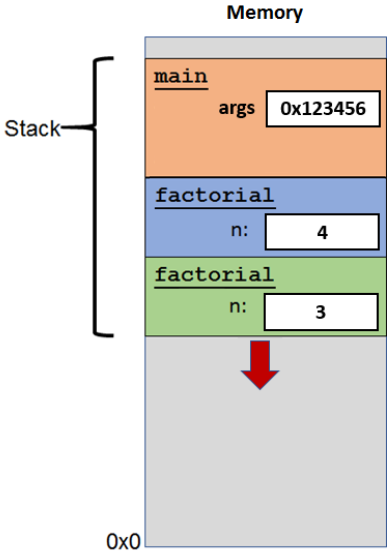
```java
public class FactorialDemo {
  public static void main(String[] args) {
    System.out.println(factorial(4));
  }

  public static int factorial(int n) {
    if (n == 1) {
      return 1;
    }

    return n *factorial(n - 1);
  }
}
```



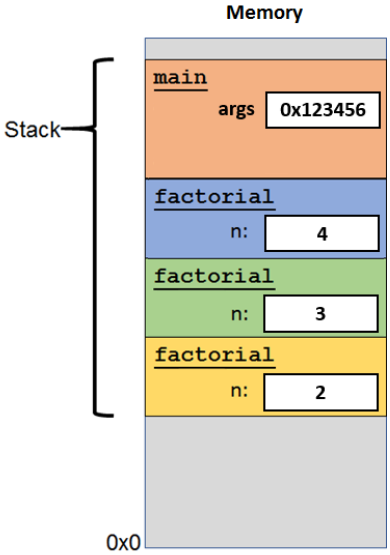
```java
public class FactorialDemo {
    public static void main(String[] args) {
        System.out.println(factorial(4));
    }

    public static int factorial(int n) {
        if (n == 1) {
            return 1;
        }

        return n * factorial(n - 1);
    }
}
```



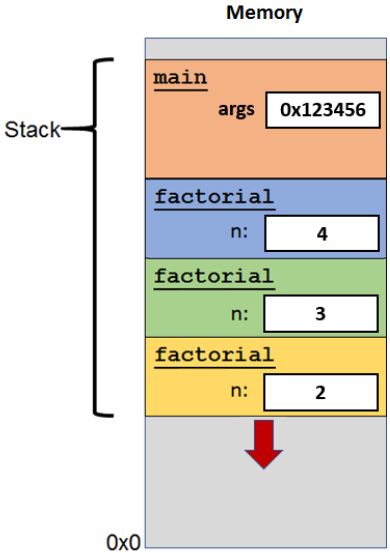
```java
public class FactorialDemo {
    public static void main(String[] args) {
        System.out.println(factorial(4));
    }

    public static int factorial(int n) {
        if (n == 1) {
            return 1;
        }

        return n *factorial(n - 1);
    }
}
```



```java
public class FactorialDemo {
  public static void main(String[] args) {
    System.out.println(factorial(4));
  }

  public static int factorial(int n) {
    if (n == 1) {
      return 1;
    }

    return n * factorial(n - 1);
  }
}
```

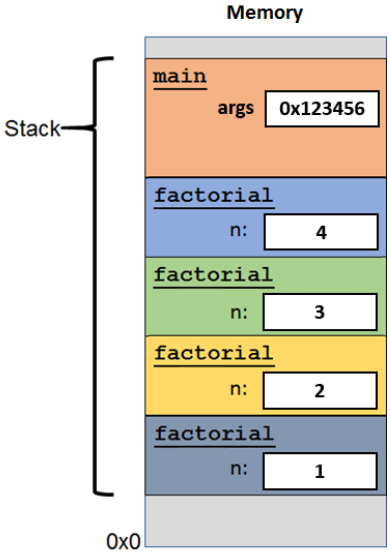


```java
public class FactorialDemo {
    public static void main(String[] args) {
        System.out.println(factorial(4));
    }

    public static int factorial(int n) {
        if (n == 1) {
            return 1;
        }

        return n *factorial(n - 1);
    }
}
```



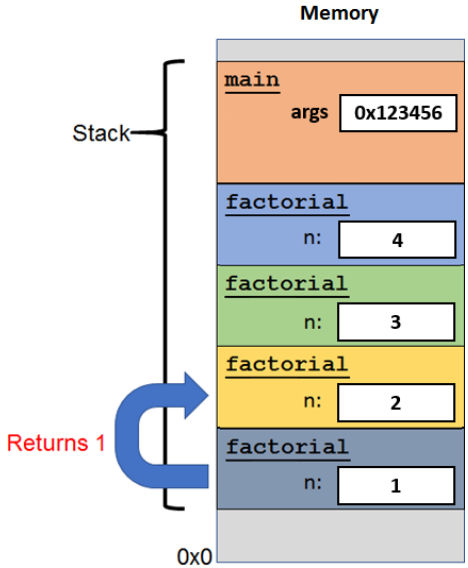
Example of Stack Memory Allocation with Recursive Calls

```java
public class FactorialDemo {
    public static void main(String[] args) {
        System.out.println(factorial(4));
    }

    public static int factorial(int n) {
        if (n == 1) {
            return 1;
        }

        return n * factorial(n - 1);
    }
}
```

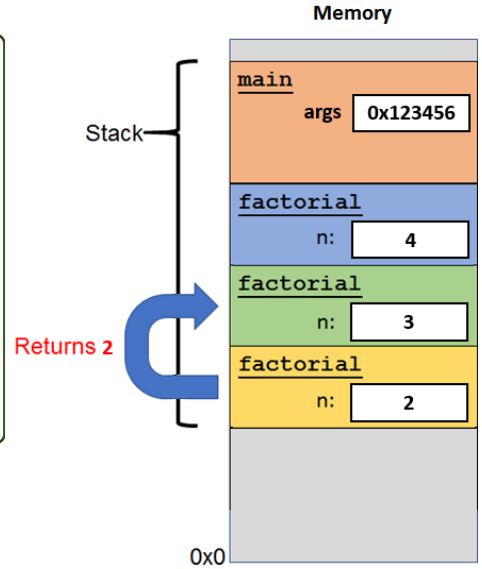


Example of Stack Memory Allocation with Recursive Calls

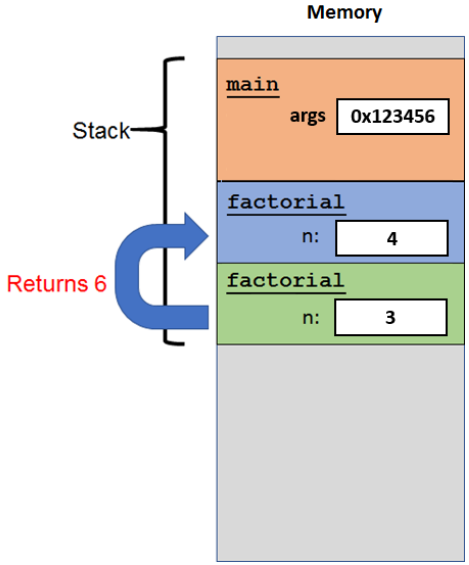
```java
public class FactorialDemo {
  public static void main(String[] args) {
    System.out.println(factorial(4));
  }

  public static int factorial(int n) {
    if (n == 1) {
      return 1;
    }

    return n * factorial(n - 1);
  }
}
```



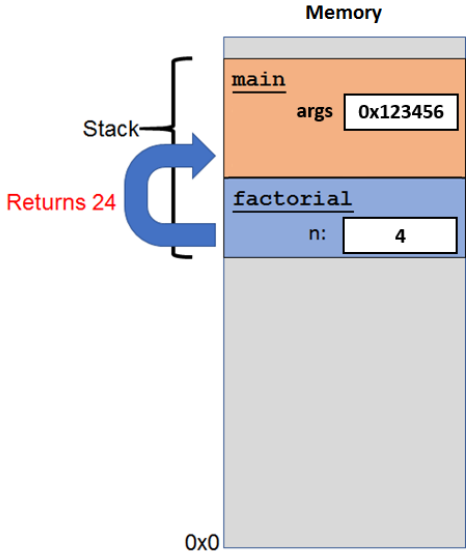
```java
public class FactorialDemo {
    public static void main(String[] args) {
        System.out.println(factorial(4));
    }

    public static int factorial(int n) {
        if (n == 1) {
            return 1;
        }

        return n *factorial(n - 1);
    }
}
```



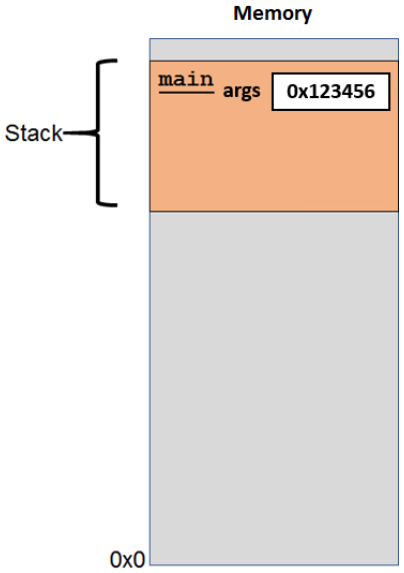
```java
public class FactorialDemo {
    public static void main(String[] args) {
        System.out.println(factorial(4));
    }

    public static int factorial(int n) {
        if (n == 1) {
            return 1;
        }

        return n *factorial(n - 1);
    }
}
```



```java
public class FactorialDemo {
    public static void main(String[] args) {
        System.out.println(factorial(4));
    }

    public static int factorial(int n) {
        if (n == 1) {
            return 1;
        }

        return n *factorial(n - 1);
    }
}
```

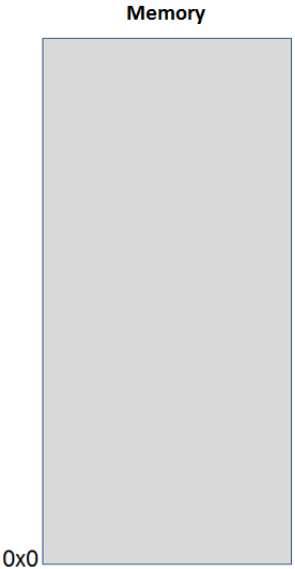


```java
public class FactorialDemo {
  public static void main(String[] args) {
    System.out.println(factorial(4));
  }

  public static int factorial(int n) {
    if (n == 1) {
      return 1;
    }

    return n * factorial(n - 1);
  }
}
```



Example of Stack Memory Allocation with Recursive Calls

```java
public class FactorialDemo {
  public static void main(String[] args) {
    System.out.println(factorial(4));
  }

  public static int factorial(int n) {
    if (n == 1) {
      return 1;
    }

    return n * factorial(n - 1);
  }
}
```



1 Basics of computer programming

2 Java Programming Introduction

3 Java Programming Basics

4 Memory Model

5 Parameter Passing Mechanism
- Passing by Value
- Passing by Reference
- Parameter Passing Mechanism in Java

6 References

- Understanding the technique used to pass information between variables and into methods can be a difficult task for a Java developer, especially those accustomed to a much more verbose programming language, such as C or C++. In these expressive languages, the developer is solely responsible for determining the technique used to pass information between different parts of the system. For example, C++ allows a developer to explicitly pass a piece of data either by value, by reference, or by pointer. The compiler simply ensures that the selected technique is properly implemented and that no invalid operation is performed.
- In the case of Java, these low-level details are abstracted, which both reduces the onus on the developer to select a proper means of passing data and increases the security of the language (by inhibiting the manipulation of pointers and directly addressing memory).
- In addition, though, this level of abstraction hides the details of the technique performed, which can obfuscate a developer's understanding of how data is passed in a program.
- We should examine the various techniques used to pass data and deep-dive into the technique that the Java Virtual Machine (JVM) and the Java Programming Language use to pass data.

- In general, there are two main techniques for passing data in a programming language:
  - Passing by value
  - Passing by reference
- Some languages consider passing by reference and passing by pointer two different techniques, in theory, one technique can be thought of as a specialization of the other, where a reference is simply an alias to an object, whose implementation is a pointer.

1 Basics of computer programming

2 Java Programming Introduction

3 Java Programming Basics

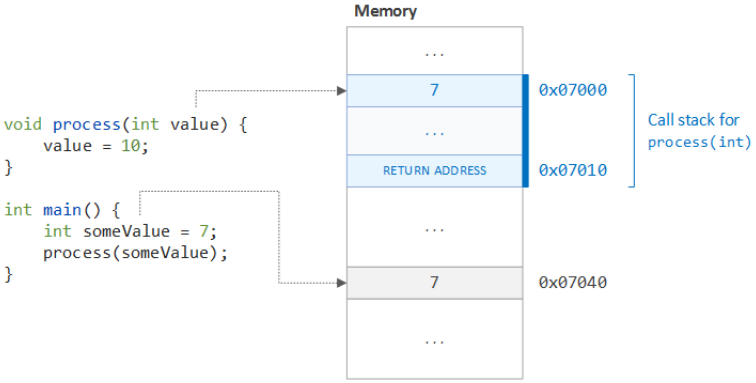
4 Memory Model

5 Parameter Passing Mechanism
- Passing by Value
- Passing by Reference
- Parameter Passing Mechanism in Java

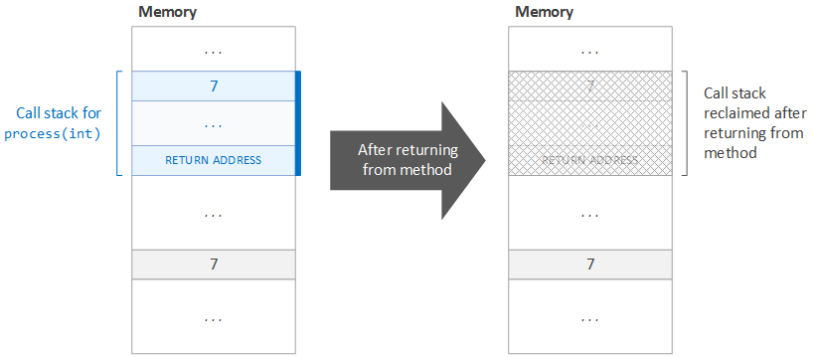
6 References

- Passing by value constitutes copying of data, where changes to the copied value are not reflected in the original value.
- Example

```cpp
#include <iostream>

using namespace std;

void process(int value) {
    cout << "Value passed into function: " << value << endl;
    value = 10;
    cout << "Value before leaving function: " << value << endl;
}

int main() {
    int someValue = 7;
    cout << "Value before function call: " << someValue << endl;  // 7
    process(someValue);
    cout << "Value after function call: " << someValue << endl;   // 7

    return 0;
}
```



- The change made to the argument passed into the process() function was not preserved after we exited the scope of the function. This loss of data was due to the fact that a copy of the value held by the someValue variable was placed on the call stack prior to the execution of the process function. Once the process function exited, this copy was popped from the call stack and the changes made to it were lost.



- The figure illustrates the action of popping the call stack at the completion of the process method.
- The value copied as the argument to the process method is lost (reclaimed) once the call stack is popped, and therefore, all changes made to that value are in turn lost during the reclamation step.

1 Basics of computer programming

2 Java Programming Introduction

3 Java Programming Basics

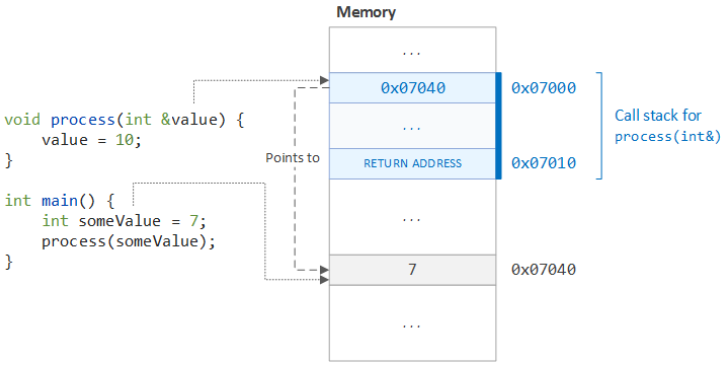
4 Memory Model

5 Parameter Passing Mechanism
- Passing by Value
- Passing by Reference
- Parameter Passing Mechanism in Java

6 References

- Passing by reference consitutes the aliasing of data, where changes to the aliased value are reflected in the original value.
- Example

```cpp
#include <iostream>

using namespace std;

void process(int &value) {
  cout << "Value passed into function: " << value << endl;
  value = 10;
  cout << "Value before leaving function: " << value << endl;
}

int main() {
  int someValue = 7;
  cout << "Value before function call: " << someValue << endl;  // 7
  process(someValue);
  cout << "Value after function call: " << someValue << endl;   // 10

  return 0;
}
```



- When exiting the function, the assignment we made to our argument that was passed by reference was preserved outside of the scope of the function.
- In the case of C++, we can see that under-the-hood, the compiler has passed a pointer into the function that points to the someValue variable. Thus, when this pointer is dereferenced (as happens during reassignment), we are making a change to the exact location in memory that stores the someValue variable.

1 Basics of computer programming

2 Java Programming Introduction

3 Java Programming Basics

4 Memory Model

5 Parameter Passing Mechanism
- Passing by Value
- Passing by Reference
- Parameter Passing Mechanism in Java

6 References

- Unlike in C++, Java does not have a means of explicitly differentiating between pass by reference and pass by value. Instead, the Java Language Specification declares that the passing of all data, both object and primitive data, is defined by the rule: All data is passed by value.

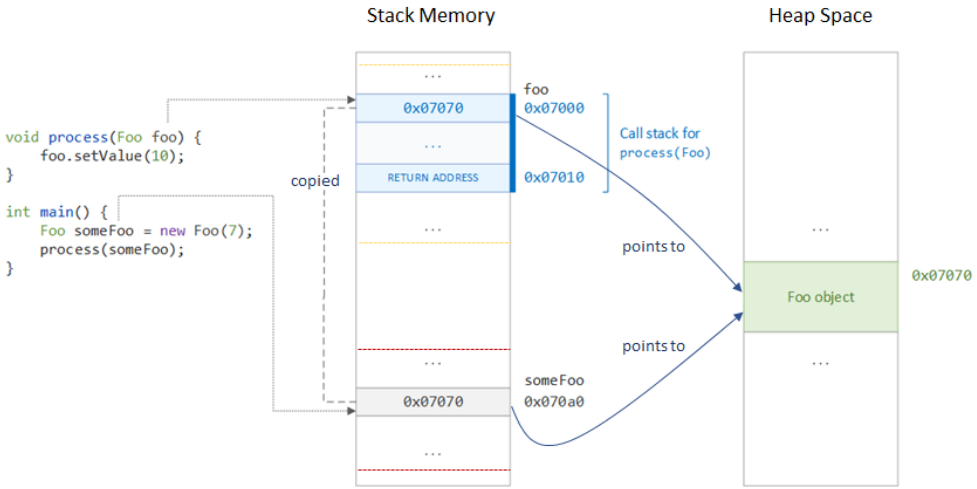
  Primitives are Pass-By-Value

  - They are always stored inside the Stack Memory. Different methods have different Stack spaces. So changes to the values in another Stack Space are not present in the former location.

  Objects are also Pass-By-Value

  - The value associated with an object is actually a pointer, called a reference, to the object in memory.
  - Objects are stored inside the Heap. The reference/pointer is stored on the Stack. So changes to the content of the object by another method are present to all references which identify the same object. The object's reference is passed by value, so the Heap address of the object is copied and passed as a parameter to the called method.

- Arguments in Java are always passed-by-value. There is NO pass-by-reference semantic in Java.



Passing object to the method process

- We can change the fields of the object that its reference passed into a method and invoke its methods, but we cannot change the object that the reference points to.
- Since the pointer is passed into the method by value, the original pointer is copied to the call stack when the method is invoked. When the method scope is exited, the copied pointer is lost, thus losing the change to the pointer value.
- Although the pointer is lost, the changes to the fields are preserved because we are dereferencing the pointer to access the pointed-to object: The pointer passed into the method and the pointer copied to the call stack are identical (although independent) and thus point to the same object.
- Thus, when the pointer is dereferenced, the same object at the same location in memory is accessed. Therefore, when we make a change to the dereferenced object, we are changing a shared object.
- Passing by reference in Java is also known as pass-by-sharing.

```java
int someValue = 10;
int anotherValue = someValue;
someValue = 17;
System.out.println("Some value = " + someValue);
System.out.println("Another value = " + anotherValue);
```

Command window

```
Some value = 17
Another value = 10
```

Example of Passing Primitive to Method

```
public void process(int value) {
  System.out.println("Entered method (value = " + value + ")");
  value = 50;
  System.out.println("Changed value within method (value = " + value + ")");
  System.out.println("Leaving method (value = " + value + ")");
}

PrimitiveProcessor processor = new PrimitiveProcessor();
int someValue = 7;
System.out.println("Before calling method (value = " + someValue + ")");
processor.process(someValue);
System.out.println("After calling method (value = " + someValue + ")");
```

Command window

```
Before calling method (value = 7)
Entered method (value = 7)
Changed value within method (value = 50)
Leaving method (value = 50)
After calling method (value = 7)
```

```java
public class Ball {}

Ball someBall = new Ball();
System.out.println("Some ball before creating another ball = " + someBall);
Ball anotherBall = someBall;
someBall = new Ball();
System.out.println("Some ball = " + someBall);
System.out.println("Another ball = " + anotherBall);
```

Command window

```
Some ball before creating another ball = Ball@6073f712
Some ball = Ball@2ff5659e
Another ball = Ball@6073f712
```

```java
public class Vehicle {
  private String name;

  public Vehicle(String name) {
    this.name = name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  @Override
  public String toString() {
    return "Vehicle[name = " + name + "]";
  }
}
```

```
public class VehicleProcessor {
  public void process(Vehicle vehicle) {
    System.out.println("Entered method (vehicle = " + vehicle + ")");
    vehicle.setName("A changed name");
    System.out.println("Changed vehicle within method (vehicle = " + vehicle + ")");
    System.out.println("Leaving method (vehicle = " + vehicle + ")");
  }

  public void processWithReferenceChange(Vehicle vehicle) {
    System.out.println("Entered method (vehicle = " + vehicle + ")");
    vehicle = new Vehicle("A new name");
    System.out.println("New vehicle within method (vehicle = " + vehicle + ")");
    System.out.println("Leaving method (vehicle = " + vehicle + ")");
  }
}

VehicleProcessor processor = new VehicleProcessor();
Vehicle vehicle = new Vehicle("Some name");
System.out.println("Before calling method (vehicle = " + vehicle + ")");
processor.process(vehicle);
System.out.println("After calling method (vehicle = " + vehicle + ")");
processor.processWithReferenceChange(vehicle);
System.out.println("After calling reference-change method (vehicle = " + vehicle + ")");
```

Command window

```
Before calling method (vehicle = Vehicle[name = Some name])
Entered method (vehicle = Vehicle[name = Some name])
Changed vehicle within method (vehicle = Vehicle[name = A changed name])
Leaving method (vehicle = Vehicle[name = A changed name])
After calling method (vehicle = Vehicle[name = A changed name])
Entered method (vehicle = Vehicle[name = A changed name])
New vehicle within method (vehicle = Vehicle[name = A new name])
Leaving method (vehicle = Vehicle[name = A new name])
After calling reference-change method (vehicle = Vehicle[name = A changed name])
```

1 Basics of computer programming

2 Java Programming Introduction

3 Java Programming Basics

4 Memory Model

5 Parameter Passing Mechanism

6 References

References

- Allen B. Downey, Chris Mayfield, ***Think Java***, (2016).
- Graham Mitchell, ***Learn Java the Hard Way***, 2nd Edition, (2016).
- Cay S. Horstmann, ***Big Java - Early Objects***, 7e-Wiley, (2019).
- James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, ***The Java Language Specification - Java SE 8 Edition***, (2015).
- Martin Fowler, ***UML Distilled - A Brief Guide To The Standard Object Modeling Language***, (2004).
- Richard Warburton, ***Object-Oriented vs. Functional Programming - Bridging the Divide Between Opposing Paradigms***, (2016).
- Naftalin, Maurice Wadler, Philip ***Java Generics and Collections***, O'Reilly Media, (2009).
- Eric Freeman, Elisabeth Robson ***Head First Design Patterns - Building Extensible and Maintainable Object***, Oriented Software-O'Reilly Media, (2020).
- Alexander Shvets ***Dive Into Design Patterns***, (2019).

Thank You!