Introduction to JAVA Programming

Rathindra Nath Dutta

Junior Research Fellow
Advanced Computing & Microelectronics Unit
Indian Statistical Institute, Kolkata

June 15, 2018

Outline

The Object-Oriented Paradigm

Procedural Approach

- Data and the functions/methods is kept separately
- As the code size increases it becomes unmanageable!
- Here emphasize on, how data is to be processed, i.e. methods
- It is *top-down*

The Object-Oriented Paradigm

Procedural Approach

- Data and the functions/methods is kept separately
- As the code size increases it becomes unmanageable!
- Here emphasize on, how data is to be processed, i.e. methods
- It is *top-down*

OOP

- Problem/program is divided into a set of entities called **objects**
- Object = Data + Methods: methods are tightly coupled with data
- Here we emphasize on the data rather than methods
- Data is often hidden and are accessed/processed via the methods
- Objects communicate via message passing through methods
- It is *bottom-up*

The OOP Philosophy I

Abstraction

- Idea is to specify only some relevant details & leave the rest hidden
- A common way to manage abstraction via <u>hierarchical definition</u> (easy to manage)
    - Define the outer functionalities and keep the internal as a black-box
    - Next define the internal in similar way
- This encourages *data hiding*
- The next three principles work together and achieve data abstraction

The OOP Philosophy II

Encapsulation

- It binds together the data and methods which operate on that data
- It is like a wrapper
- The **class** construct is generally used for encapsulating data and the method into an **object**

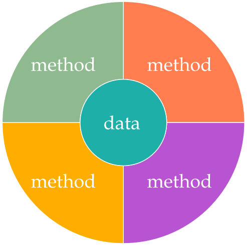


Figure: Visualization of an object

The OOP Philosophy III

Inheritance

- Process of acquiring the properties (both data and methods) from another object
- Enables the hierarchical definition of the classes, and hence objects
- First we define the common attributes, then specific attributes are defined as required:
  - The shapes have area, perimeter, fill color etc.
  - The circle has radius, but a square has side, whereas a triangle have three sides!

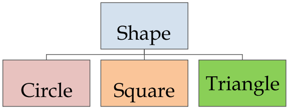


Figure: Inheritance Hierarchy

The OOP Philosophy IV

Polymorphism

- A Greek word, which means “many forms”
- This feature allow us to define a common interface for some general class of actions
    - One of the specific action is chosen depending upon the situation
- There are many ways to apply this feature!
    - Function overloading
    - Dynamic method dispatch
    - Generics/Template class etc.

The Java Programming Language

- Java is realated to C++, which is direct descendent of C
- The syntax and many features were taken from C++
- Java was developed as response to the shortcomings of C++ and other languages
  - We needed coding for the internet which required **security** and **platform-independency**
- Now Java can be vaguely thought as “Internet version of C++”
- Java influenced the C# language developed by Microsoft

The History of Java I

- The Sun Microsystems developed a C++ like language named Oak (1991) headed by James Gosling
- Later renamed it as Java (1995)
- In 1996 JDK 1.0 was released
- 1997 JDK 1.1 (many libraries were added)
- 1998 JDK 1.2, this is called Java 2 (to denote 2nd generation) (Thread support was added)
- 1999 Sun Released J2SE and J2EE (still JDK 1.2)
- 2003 J2SE 1.3 (with JDK 1.3)
- 2002 J2SE 1.4 (with JDK 1.4)
- 2004 J2SE 5 (to denote a significant change) (with JDK 1.5)
    - It introduced Generics, Annotations, Autoboxing & Auto-unboxing, Enumeration, enhanced for-each loop, varargs, static import, formatted I/O, concurrency utilities

The History of Java II

- Next one is called: Java SE 6 (with JDK 1.6)
- In 2010 Oracle acquired Sun Microsystems, then released Java SE 7 (with JDK 1.7) (this was another significant improvement)
- In 2014 Java SE 8 came with more improvements and features
- Current version (as of today) is Java SE 9

Java's Magic: Bytecode

- The key that incorporates both security and portability
- The output of the java compiler is not some executable code, rather it is **bytecode**
- Bytecode is highly optimized set of instructions designed to be executed by the Java runtime system, called *Java Virtual Machine* (**JVM**)

$$\text{source code} \xrightarrow{\text{javac}} \text{bytecode} \xrightarrow{\text{java}} \text{execution}$$

- The $\underline{\text{bytecode is fully machine independent}}$ code, only machine specific JVM must be installed – **portability**
- The code is contained(while running) within the JVM, thus can not affect other parts of the system – **security**
- HotSpot technology was introduced, which provides Just-In-Time (**JIT**) compilers for bytecodes.

Java Buzzwords

A list of buzzwords which describes the full potential of Java

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

A Hello World Program I

The Code

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

A Hello World Program II

What Else?

- A text editor like notepad/gedit/vim/nano
- JDK installation for compiling source code
- JRE installation for execution(often installs with JDK) `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
- An IDE like eclipse (optional) `http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/oxygen3a`

A Hello World Program III

Executing The Code

- Save it as **Test.java** (file name should be same, and extension denotes it's a java source file)
- Compile: `javac Test.java`
- It will create a single file named Test.class, which contains the bytecode (since we had only one class called Test)
- Run: `java Test`
- Be careful that we write only the class name containing the main() method (not the filename with .class extension)

Anatomy the Hello World program I

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

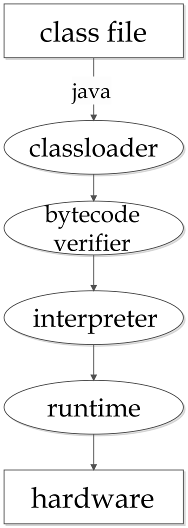
- `class` keyword is used to define a java class, Test is the class name
- `public` is an *access modifier* and defines the visibility of a variable/function/class, public makes things globally accessible
- `static` is an *access specifier* which allows a class member to be accessed from outside of the class without having to instantiate an object of that class, we need it since JVM calls the main() method
- The braces {. . . } defines a block of code; the outer braces for the class definition and the inner braces are for body of the main() method

Anatomy the Hello World program II

- The `void` keyword tells the compiler that the main() method returns nothing
- `String[]` args is a parameter for the main() method, it an array of String type objects(i.e. strings) named args (array is nothing but collection of similar objects)
- `System` is a predefined class in `java.lang` package
- `out` is the output stream object(also a member of the System class) which is connected to the console
- `println()` is a built-in method which takes a `String` object as its argument and prints it into the console
- "hello world" is a String literal passed into the println() method
- A statement ends with a semicolon

Execution Flow of a Java Program

classloader	the subsystem of JVM that is used to load class files
bytecode verifier	checks the code fragments for illegal code that can violate access right to objects
interpreter	reads bytecode stream then executes the instructions



Java Runtime System I

- **JVM** (Java Virtual Machine) is an abstract machine a specification that provides runtime environment in which java bytecode can be executed
- JVMs are available for various hardware and software platforms (i.e. JVM is platform dependent)
- Its implementation has been provided by Sun(Oracle) and other companies; OpenJDK is popular in various linux distributions
- The implementation is known as **JRE** (Java Runtime Environment).
- Whenever we run a java program, an instance of JVM is created

Java Runtime System II

The JVM performs following operation:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

Java Runtime System III

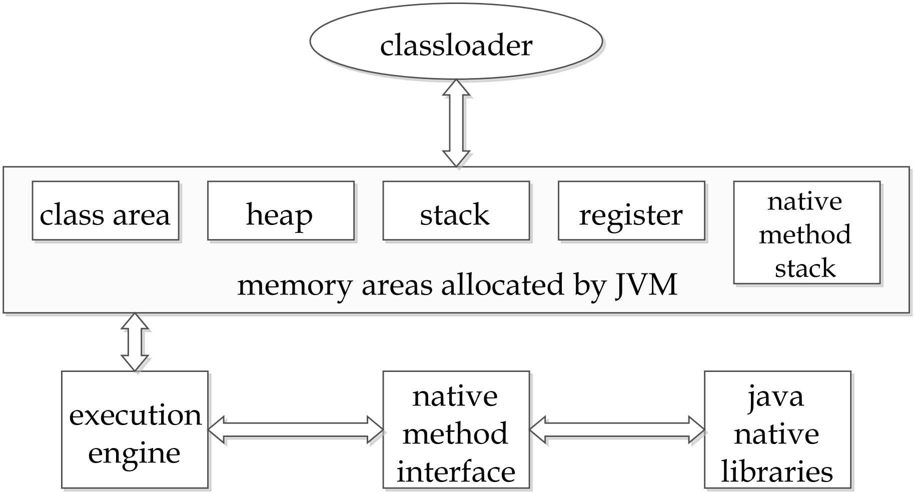


Figure: Java Runtime System

Java Runtime System IV

- **Classloader** is a subsystem of JVM that is used to load class files
- **Class(Method) Area** stores per-class structures such as the runtime constant pool, field and method data, the code for methods
- **Heap** is the runtime data area in which objects are allocated
- **Stack** stores frames. It holds local variables and partial results, and plays a part in method invocation and return
  - Each thread has a private JVM stack, created at its birth
  - A new frame is created each time a method is invoked and destroyed when its method invocation completes
- **Program Counter Register** contains the address of the Java virtual machine instruction currently being executed
- **Native Method Stack** contains all the native methods used in the application

Java Runtime System V

Execution Engine contains:

- A **virtual processor**
- **Interpreter**: Reads bytecode stream then execute the instructions
- **Just-In-Time(JIT) compiler**: used for improving performance; JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation

Basic Syntax

- **Case Sensitivity** – Java is case sensitive, which means identifier `Hello` and `hello` would have different meaning in Java
- **Class Names** – the first letter should be in uppercase; if several words are used to form a name of the class, each inner word's first letter should be in uppercase (**camel case**)
- **Method Names** – method names should start with a lowercase letter; camel case is used for longer names
- **Source File Name** – name of the program file should exactly match with the class name
- **public static void main(String args[])** – Java program processing starts from the main() method which is a mandatory part of every Java program.

Identifiers

- Java components require names; names used for classes, variables, and methods are called **identifiers**
  - All identifiers should begin with a letter (A-Z or a-z), currency character ($) or an underscore (_)
  - After the first character, identifiers can have any combination of characters
  - A keyword cannot be used as an identifier
  - Most importantly, identifiers are case sensitive
- Examples of legal identifiers: `age`, `$salary`, `_value`, `__1_value`.
- Examples of illegal identifiers: `123abc`, `-salary`.

Keywords

List of reserved words in Java

`abstract`	`continue`	`for`	`new`	`switch`
`assert`	`default`	`goto`[1]	`package`	`synchronized`
`boolean`	`do`	`if`	`private`	`this`
`break`	`double`	`implements`	`protected`	`throw`
`byte`	`else`	`import`	`public`	`throws`
`case`	`enum`	`instanceof`	`return`	`transient`
`catch`	`extends`	`int`	`short`	`try`
`char`	`final`	`interface`	`static`	`void`
`class`	`finally`	`long`	`strictfp`	`volatile`
`const`[1]	`float`	`native`	`super`	`while`

Moreover `false`, `null`, `true` are reserved word for literal values

[1]not used

Comments

- Both single-line and multi-line comments are supported
- All characters inside any comment are ignored by Java compiler

```
public class MyFirstJavaProgram {
    /* This is my first java program
    *  This will print 'Hello World' as the output
    *  This is an example of multi-line comments
    */
    public static void main(String []args) {
        // This is an example of single line comment
        /* This is also an single line comment */
        System.out.println("Hello World");
    }
}
```

- Documentation comments `/** ... */` facilitates `javadoc`

Variables & Datatypes

- A variable is name of reserved area allocated in memory
- Each variable is associated with a type when declared

DataType	**Default Value**	**Size**
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

- Java uses Unicode system rather than ASCII code system
- Java requires forward declaration of variables
- Every non-primitive-type(arrays, String, objects etc.) variables must be initialized before it is used

Scope & Lifetime

- A variable is declared within some block enclosed by curly braces
- That block defines the **scope** of the variable, i.e. visibility of that variable to the other parts of the program
- We can not access it outside of its scope
- A block also defines the **lifetime** of a variable, i.e. how long its value is retained
- At the end of its lifetime a variable becomes eligible to be cleared by the *Garbage Collector* (GC daemon)
- The scope of a class (member) variable is confined to the corresponding object and its lifetime is same as its container object

Type Conversion

- Java performs automatic type promotion for an assignment when destination type is lager in size

  `int x; short y; double z; x = y; z = x;`
- Exception: `boolean` type can not be stored to other types
- However down-casting must be done explicitly

  `int x; double z; x = (int)z;`
- Explicit casting has other applications like fractional division

  `(float)3/2`
- Type promotion in expressions:
  - All `byte`, `short`, and `char` types are promoted to `int`
  - If one of the operands is `long` then the whole expression is promoted to `long`
  - If one of the operands is `float` then the entire expression is promoted to `float`
  - If one of the operands is `double` then result is `double`

Operators

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

Control statements

- `if-else` and `switch` works just like in C/C++
- `while`, `do-while`, and `for` loops are also same as in C/C++
- Jump: `break` and `continue` is there

Arrays

- Array is a group of similar typed elements
- Randomly accessible via index
- Declaring an array: `type varName[];` or `type[] varName;`
- Allocating space for the array: `varName = new type[size];`
  `int[] dataArray = new int[10];`
- we are allowed to write both `int dataArray[]` and
  `int[] dataArray`
- The one with square brackets after datatype is more readable and preffered; when declaring multiple arrays it is compact
- Initializing an array: `double[] arr = {1.9, 3.4, 3.05};`

An Example

```
//creating an array
int[] arr = new int[10];
//storing values
for (int i = 0; i < arr.length; i++) {
    arr[i] = i;
}
//updating values
for (int i = 0; i < arr.length; i++) {
    arr[i]++;
}
//retrieving values
for (int i = 0; i < arr.length; i++) {
    System.out.println(arr[i]);
}
```

foreach loop

Used to iterate over a collections of object, like an array

```
int[] arr = new int[10];
.
.
.
for (int val : arr) {
    System.out.println(val);
}
```

The loop takes the value of each elements in the set(arr array in this case) one at each iteration into a (iteration) variable (here val)

Class & Object

- An *object* is an instance of a class.
- Objects have states and behaviours. Example: A dog has states - colour, name, breed as well as behaviours – wagging the tail, barking, eating
- A *class* can be defined as a template/blueprint that describes the behaviour/state that the object of its type support
- Both variables and methods declared inside a class definition are members of the class

An Example

```
class ABC {
    //states or variables
    int x;
    //behaviours or methods
    void foo() {
        // function body
    }
    int bar() {
        // function body
    }

    public static void main(String []args) {
        ABC obj; //creating an object
        //a variable of type ABC
        obj = new ABC(); //instantiating the object
    }
}
```

A closer Look at Object Creation

`new`

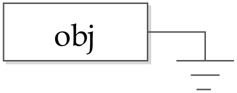
- All object variables are just references, initially points to `null`
- The `new` operator dynamically allocates memory for an object
- It translate a logical construct (class) into a physical reality (object)

A closer Look at Object Creation

new

- All object variables are just references, initially points to `null`
- The `new` operator dynamically allocates memory for an object
- It translate a logical construct (class) into a physical reality (object)

```
ABC obj;
```

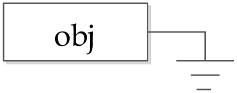


A closer Look at Object Creation

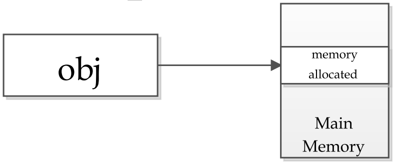
new

- All object variables are just references, initially points to `null`
- The `new` operator dynamically allocates memory for an object
- It translate a logical construct (class) into a physical reality (object)

```
ABC obj;
```

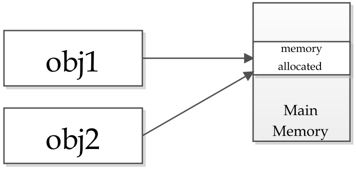


```
obj = new ABC();
```



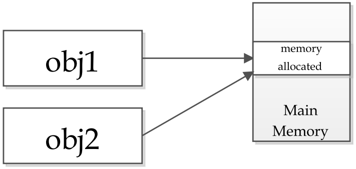
Understanding Object References

```
ABC obj1 = new ABC();
ABC obj2 = obj1;
```

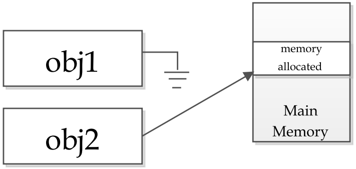


Understanding Object References

```
ABC obj1 = new ABC();
ABC obj2 = obj1;
```



```
ABC obj1 = new ABC();
ABC obj2 = obj1;
obj1 = null;
```



Methods

- Declaration and usage of method is similar to C/C++
- Methods can have parameters passed into it when called
- Methods can return some data; we can write `return` in methods with return type `void`
- In Java everything is *call-by-copy*, for primitives values gets copied while for objects reference gets copied.

Method overloading

- Java allows two or more methods within same class having same name
- They must differ by parameter declaration (type and/or count)
- While invoking the parameters determines which version of the method to load
- Its enables *compile-time polymorphism*

Constructor

- A *constructor* initializes an object upon its creation
- It has same name as its class
- Its syntactically similar to a method, except it does not have any return type, not even void
- It gets immediately called when an object is being instantiated by the `new` operator
- It implicitly returns the fully instantiated object of the class
- Even if we don't write one explicitly compiler provides a default (dummy) one
- Just like any other method constructors can have parameters and can be overloaded
- Refer to example 2(complex)

This Keyword

- `this` can be used inside any (non-static)method to refer to the current object
- `this` is always a reference to the object on which the method is invoked
- `this` is also used to invoke other constructors of te same class, this reduces code
- Refer to example 2.1(complex)

Garbage Collection

- Objects are dynamically allocated by the `new` operator, but there is no **delete** operator like C++, java handles deallocation automatically
- When no reference to an object exists, the allocated momory space is eligible for garbage collection
- The GC daemon runs sporadically (if at all); its implementation may vary for different vendor
- For the most part, one should not have to think about it while writing typical programs
- One can request JVM to perform garbage collection by executing `System.gc()`; but JVM may or may not decide to do a GC at that point
- You may read the discussion at `https://stackoverflow.com/questions/66540/when-does-system-gc-do-anything`

Finalization

- Often objects need to perform some action when it is destroyed
- Objects may hold non-Java resources such as file handles, which must be released before the object is destroyed
- Similar to destructor in C++, Java provides a `finalize()` method which have the following form:

```
protected void finalize() { //your code }
```

- Notice that the method has `protected` *access modifier* to prevent accessing it from outside its class
- The method is invoked when GC triggers, not when the object goes out of its scope/lifetime

Access Control

- As sated earlier classes are used to create *data abstraction*
- Hiding(restricted access) members, both data and methods, is an important aspect of data abstraction
- Java provides four access modifiers:

`public`	members can be accessed from everywhere
default	members can be accessed from anywhere within same package
`protected`	members can be accessed from anywhere in the same package & within subclasses in other packages
`public`	members can be accessed only within same class

- Refer to example 3(stack)

Access Specifier: Static

- When a member is declared `static`, it can be accessed without creating any object of that class; most common example is `main()`
- For a static <u>variable</u> all object instances of the class share the same variable; no individual copy is made - its like global variable within a class
- <u>Methods</u> declared static have several restrictions:
  - can only directly call other static methods and can only directly access static data
  - non-static data & methods must be accessed though some object (what we do inside `main()`)
  - cannot refer to `this` or `super` in any way
- `java.lang.Math` class provides a large collection useful methods declared as static
- A static <u>block</u> gets executed exactly once when the class is loaded; generally used for initialization of static fields

Practice exercise: modify the Complex class to implement a counter which increments whenever a new object is created

Access Specifier: Final

- A <u>field</u> can be declared as `final` to prevent its content from being modified; essentially makes it a constant
- Making a <u>method</u> final prevents it from being overridden in some derived class
- final methods can enhance performance: compiler is free to make *inline* calls to them, *early binding* is possible
- Declaring a <u>class</u> as final prevents it from being inherited; it implicitly declares all of its methods as final too
- It is illegal to declare a class as both abstract and final

Nested & Inner Classes

- A *nested class* is a class defined within another class
- It is possible to declare a class within any block scope
- Scope is bounded by the enclosing class/block
- It can directly access members (even private ones) of its enclosing class
- A nested class can be static (declared with static specifier); all non-static members of its enclosing class must be accessed through an object
- An *inner class* is a non-static nested class; it can directly access all members of the outer class
- An *anonymous inner class* is a inner class having no name; such classes are widely used for event handling

The String Class

- String literals are also objects of class `String`
- A string object is immutable (constant); whenever we modify we actually create a new object
- Java also provides `StringBuffer` for string manipulation
- + works as string concatenation operator if either of its two operands is a string (other one is converted to string using `toString()` method is required)
- Some useful methods of `String` class:

```
boolean equals(str2)

int length()

char charAt(index)
```

Inheritance