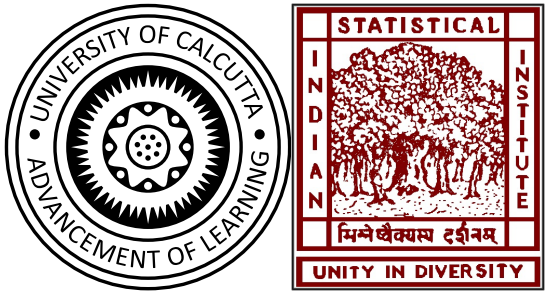
Introduction to JAVA Programming

Rathindra Nath Dutta

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

June 15, 2018



Outline

1. Java Programming: Class Fundamentals
   - Class & Object
   - Method & Constructor
   - `this` Keyword
   - Object Deletion
   - Access Controls
   - Nested Class
   - About `String` Class
   - Command Line Argument
   - Arrays Revisited

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

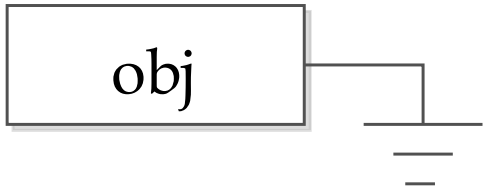
- All object variables are just references, initially points to `null`
- The `new` operator dynamically allocates memory for an object
- It translate a logical construct (class) into a physical reality (object)

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;
```

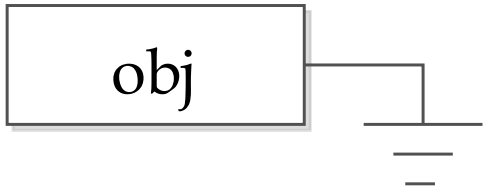


A closer Look at Object Creation

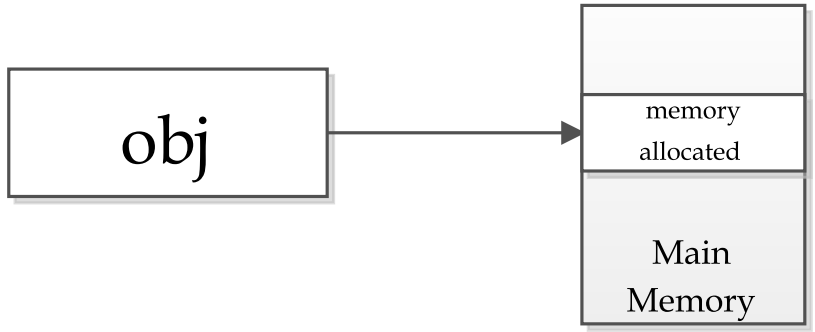
new

- 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;
```

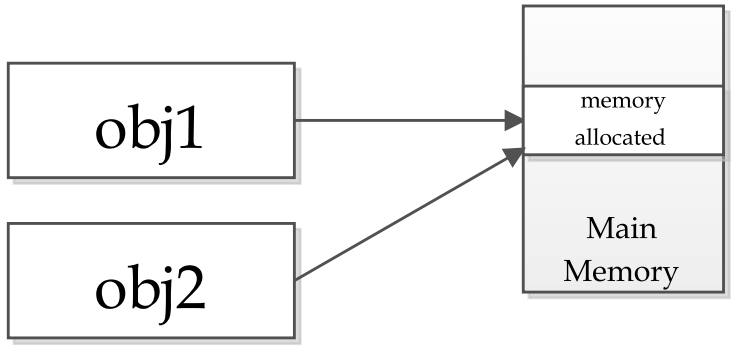



```
obj = new ABC();
```



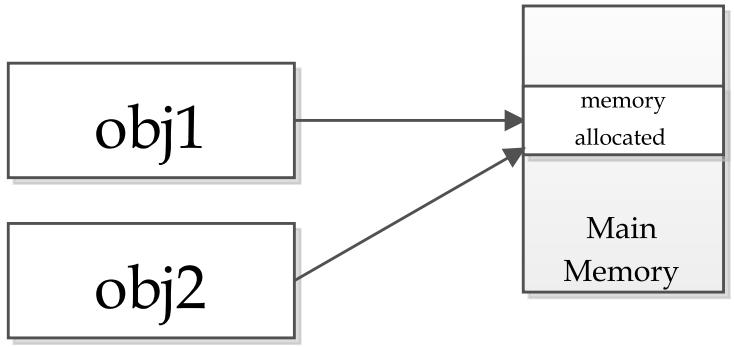
Understanding Object References

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

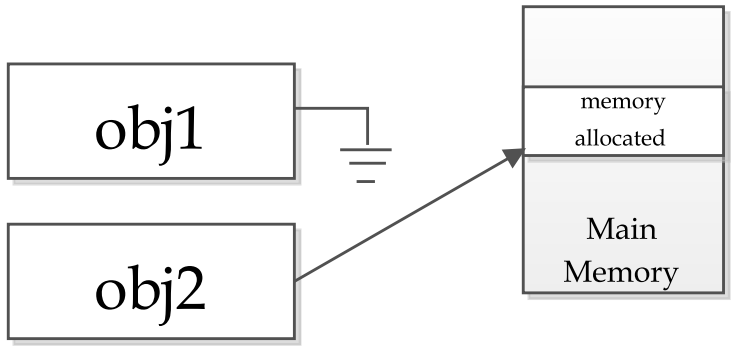


Understanding Object References

```
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)

What is `this`?

- `this` keyword 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 the same class, (reduces redundant code fragments)
- 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() {...}
```

- 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

- A `static` member can be accessed without creating any object of that class; most common example is `main()`

Access Specifier: Static

- A `static` member can be accessed without creating any object of that class; most common example is `main()`
- For a static variable all object instances of the class share the same variable; no individual copy is made - like global variable within a class

Access Specifier: Static

- A `static` member can be accessed without creating any object of that class; most common example is `main()`
- For a static variable all object instances of the class share the same variable; no individual copy is made - like global variable within a class
- Methods declared static have several restrictions:
  - can only directly call other static methods & 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

Access Specifier: Static

- A `static` member can be accessed without creating any object of that class; most common example is `main()`
- For a static variable all object instances of the class share the same variable; no individual copy is made - like global variable within a class
- Methods declared static have several restrictions:
  - can only directly call other static methods & 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 static methods

Access Specifier: Static

- A `static` member can be accessed without creating any object of that class; most common example is `main()`
- For a static variable all object instances of the class share the same variable; no individual copy is made - like global variable within a class
- Methods declared static have several restrictions:
  - can only directly call other static methods & 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 static methods
- A static block gets executed exactly once when the class is loaded; generally used for initialization of static fields

Access Specifier: Static

- A `static` member can be accessed without creating any object of that class; most common example is `main()`
- For a static variable all object instances of the class share the same variable; no individual copy is made - like global variable within a class
- Methods declared static have several restrictions:
  - can only directly call other static methods & 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 static methods
- A static block 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 field can be declared as `final` to prevent its content from being modified; essentially makes it a constant
- Making a method final prevents it from being overridden in some derived class
- Such final methods can enhance performance: compiler is free to make *inline* calls to them, *early binding* is possible
- Declaring a class 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`
- Refer to example 3(Stack)

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
- Can directly access members (even private ones) of its enclosing class
- Outer class cannot directly access member of the nested class
- A nested class can be static (declared with `static` specifier); accessing non-static members of its enclosing class must be done 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

Why Bother?

Followings are compelling reasons for using a nested classes.

A way of logically grouping classes that are only used in one place If a class is useful to only one other class, then it is logical to embed it in that class and keep them together. Nesting such "helper classes" makes their package more streamlined.

Increases encapsulation Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can still access them. In addition, B itself can be hidden from the outside world.

More readable & maintainable code Nesting small classes within top-level classes places the code closer to where it is used

Refer to example 4(LinkedList)

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)
```

Command Line Argument

- The `main()` accepts an array of `String` objects passing *command-line arguments*

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

- The array `args` is populated by the information passed directly after the program name on the command line when executed
- These information are passed as strings (splitted by whitespace)
- The first command-line argument is stored at `args[0]`, the second at `args[1]`, and so on

A Brain Teaser

Write a program to determine whether an integer (passed from command-line) is even or odd without using any conditional statements and the using modulo operator

A Brain Teaser

Write a program to determine whether an integer (passed from command-line) is even or odd without using any conditional statements and the using modulo operator

```
public static void main(String[] args) {
    if(args.length<1)
        System.out.println("Syntax: java Test <integer>");
    else {
        int n = Integer.parseInt(args[0]);



    }
}
```

A Brain Teaser

Write a program to determine whether an integer (passed from command-line) is even or odd without using any conditional statements and the using modulo operator

```
public static void main(String[] args) {
    if(args.length<1)
        System.out.println("Syntax: java Test <integer>");
    else {
        int n = Integer.parseInt(args[0]);

        String[] str = {"Even", "Odd"};
        System.out.println(str[n&1]);

    }
}
```

A Closer Look at Arrays

- Multidimensional arrays can be allocated simply writing as follows:

```
int[][] arr= new int[dim1_size][dim2_size];
```

A Closer Look at Arrays

- Multidimensional arrays can be allocated simply writing as follows:

```
int[][] arr= new int[dim1_size][dim2_size];
```

- Alternatively we can write the following:

```
arr = new int[rows][];
for (int i = 0; i < dim1_size; i++) {
    arr[i] = new int[dim2_size];
}
```

A Closer Look at Arrays

- Multidimensional arrays can be allocated simply writing as follows:

```
int[][] arr= new int[dim1_size][dim2_size];
```

- Alternatively we can write the following:

```
arr = new int[rows][];
for (int i = 0; i < dim1_size; i++) {
    arr[i] = new int[dim2_size];
}
```

- There is no advantage to individually allocating the second dimension arrays here
- When we allocate dimensions individually, we need not to allocate the same size(number of elements) for each dimension
- A multidimensional array is actually *array of arrays*, the length of each array may vary

A Closer Look at Arrays

- Multidimensional arrays can be allocated simply writing as follows:

```
int[][] arr= new int[dim1_size][dim2_size];
```

- Alternatively we can write the following:

```
arr = new int[rows][];
for (int i = 0; i < dim1_size; i++) {
    arr[i] = new int[dim2_size];
}
```

- There is no advantage to individually allocating the second dimension arrays here
- When we allocate dimensions individually, we need not to allocate the same size(number of elements) for each dimension
- A multidimensional array is actually *array of arrays*, the length of each array may vary
- Refer to example 5(matrix)