Chapter 2 . Everything is a object

- A reference of a object is separate than the object. A reference can exist even if there is no object associated with it.

- Where storage live :

1. Registers : It is the fastest storage because registers are present in processor itself. But number of registers are very few and programmer don't have direct control of them.
2. Stack : This is the second fastest after register as processor have direct support for them through *stack pointers.* They are a part of RAM and Java System must know while creating the program, the exact lifetime of all the items that lives on the stack. This put constraint on flexibility of the program. So java storage exist on the stack particularly references but java objects themselves are not stored on it.
3. Heap: It is general purpose storage also part of the RAM where objects lives. For heap, java system don't need to know how long the storage must stay on the heap. This gives a good amount of flexibility. Whenever new keyword is used to create a object, the storage is allocated on the heap. But heap take more time to allocate and cleanup storage compare to stack.
4. Constant storage: Constant values are always placed in programmes but sometime they are placed in ROM in embedded systems.
5. Non-RAM storage: Sometime we need objects to stay outside the programme, even when program is not running, outside the control of programme. Like *streamed objects* which are turn into stream of bytes to send from one machine to other machine. Also *persistence objects* which are stored on the disk so that they can hold their state even when program is terminated.

- Static keyword :

- When one want to use a single storage for a field no matter how many objects holds that field they still share the same value. Also when one want a method which is not associated with any object then one should do that using static keyword.

- Example

```
class StaticTest {
static int i = 47;
```

When one will create two objects as shown below

```
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
```

Both st1.i and st2.i will have the same value 47. If the value get changed through one object then the same value will reflect for the other.

- One can even use static fields directly using class name. Example for above we can use field 'i' directly as StaticTest.i.
- Static methods too can be called in the same way with class name. Lets say there is a static method increment() in class StaticTest then one can call it directly without creating it object as StaticTest.increment().
- But it is also true that one can call static method using an object too.
- (KnS) Static methods can't be overridden as they belongs to class.
- (KnS) Only methods, variables and top-level nested classes can be marked as static.
- (KnS) Constructors, classes, Interfaces, Inner classes, Inner class methods and instance variables, Local variables all these can't be marked as static.
- Non-static methods can't be called through static methods even they are a part of same class. Example

```
package ch2EverythingIsAnObject;

class StaticAccess

{
	int a;
	static int b;

	static void method1()
	{
	  //a=10;//cannot access non-static variables
	  b=20;
	  //method2(); //cannot call a non-static method
	   System.out.println("inside static method");
	   System.out.println("value of b :"+b);

	}

	void method2()
	{
	  a=30;
	  b=40;//non-static method can access static variable
	   System.out.println("inside non-static method");
	   System.out.println("value of a:"+a+" "+"value of a:"+b);
	   method1();//non-static method can call static method
	}
}

public class StaticAccessingStatic
{
	public static void main(String args[])
	{
		StaticAccess sa = new StaticAccess();
		sa.method1();
		sa.method2();
```

```
            }
        }
        Output:
        inside static method
        value of b :20
        inside non-static method
        value of a:30 value of a:40
        inside static method
        value of b :20
```

Above one can see method2() cant be called from a static method but reverse is possible. (Note: see also chapter 5 for why keyword 'this' cant be used in static method)

```
package ch2EverythingIsAnObject;

class Static2
{
	static int i1, i2;
	int i3;
	static void increment()
	{
		i1++;
		i2++;

	}
	static void decrement()
	{
		i1--;
		i2--;
	}
	static void printValue()
	{
		System.out.println(i1+" "+i2);
	}
	void sum()
	{
		int sum = 0;
		sum = i1+i2;
		printValue();//a non-static member can call a static
		System.out.println("sum : "+sum);

	}
	static void AccessNonStatic()
	{
		//a static method cannot call a non-static members
	    // i3 = 0;
		//System.out.println(sum());
	}
}
public class StaticMembers {
	public static void main(String[] args) {
		// calling members by classname
		System.out.println("calling with classname");
		System.out.println(Static2.i1+" "+Static2.i2);
		Static2.increment();
		Static2.printValue();
		Static2.decrement();
		Static2.printValue();
```

```
            //Static2.sum();//cant call non-static member using classname

            System.out.println("calling with object");
            Static2 stc = new Static2();
            System.out.println(stc.i1+" "+stc.i2);
            stc.increment();
            stc.printValue();
            stc.decrement();
            stc.printValue();

            System.out.println("calling non-static members");
            stc.increment();
            stc.sum();
            }
}
Output:
calling with classname
0 0
1 1
0 0
calling with object
0 0
1 1
0 0
calling non-static members
1 1
sum : 2
```

- **Special case : Primitive types :**

Instance of creating variables by using new, which place them on the heap, java uses a alternative where variables are automatically created without any references. This new types are called primitive types. Primitive types holds the values directly and they are placed on the stack hence they are much more efficient.

- Wrapper class allows one to make non-primitive objects who represents that primitive types.

Ex. `char c = 'x';`

```
Character ch = new Character(c);
```

Or one could also use:

```
Character ch = new Character('x');
```

Java SE5 *autoboxing* will automatically convert from a primitive to a wrapper type:

```
Character ch = 'x';
```

and back:

```
char c = ch;
```

- **High Precision numbers :**

Java provides two classes to perform high-precision arithmetic: BigInteger and BigDecimal. Although they fits as the same category of wrapper classes but either don't have primitive analogue. One can do everything that one does with int and char but one must use method calls instead of operators. Here the operation is bit slow because big sizes are involved.

- **Scoping :**

1. Scope is java is determined by curly braces {}.

Ex.

```
{
int x = 12;
// Only x available
{
int q = 96;
// Both x & q available
}
// Only x available
// q is "out of scope" :
}
```

2. Scope of Objects:

Lifetime of objects is not same as that of primitives.

```
{
String s = new String("a string");
} // End of scope
```

Even as above scope of reference 's' is not longer exist but objects still lives on the heap which is removed by garbage collection when its find that no reference is referencing to it.

- The argument list :

```
class Return1
{
	int a = 10, b = 10;
	int sum = 0;
	int retSum()
	{
		sum = a+b;
		return sum;
		//System.out.println("after return");//this wont execute
	}
	void retNothing()
	{
		System.out.println(sum);
		return;//its not compulsory but still code runs fine
```

```
        }
}
```

'return' keyword return the data from a method. If we write any code after the return method then there will be compiler error as shown above in method **retSum()**. Also when you don't want to return anything one can mention **void** as return type. Also as seen above in method **retNothing()** you can still use return keyword (itss not mandatory)and it will work fine.

The static keyword can be used in **3** scenarios

1. static variables
2. static methods
3. static blocks of code.

Static Variables :

- It is a variable which **belongs to the class** and **not** to **object**(instance)
- Static variables are **initialized only once** , at the start of the execution . These variables will be initialized first, before the initialization of any instance variables
- A **single copy** to be shared by all instances of the class
- A static variable can be **accessed directly** by the **class name** and doesn't need any object
- Syntax : <***class-name>.<variable-name>***

Static Method :

- It is a method which **belongs to the class** and **not** to the **object**(instance)
- A static method **can access only static data**. It can not access non-static data (instance variables)
- A static method **can call only** other **static methods** and can not call a non-static method from it.
- A static method can be **accessed directly** by the **class name** and doesn't need any object
- Syntax : <***class-name>.<method-name>***
- A static method cannot refer to "this" or "super" keywords in anyway

Side Note:

- main method is static , since it must be be accessible for an application to run , before any instantiation takes place.

Static Block :

The static block, is a block of statement inside a Java class that will be executed when a class is first loaded in to the JVM

```
1  class Test{
2   static {
3   //Code goes here
4   }
5  }
```

A **static block helps to initialize the static data members**, just like constructors help to initialize instance members.