Demo.java:

```
package com.masai;

public class Demo {

	int x = 100;

	Demo d1 = new Demo();

	//A a1 = new A();

	public static void main(String[] args) {

		Demo d1 = new Demo();



	}

}
```

--the above application will throw a runtime exception called StackOverFlowError exception.

Demo.java:

```
package com.masai;

public class Demo {

	int x = 100;

	Demo d1;

	//A a1 = new A();

	public static void main(String[] args) {

		Demo d1 = new Demo();
```

```
		System.out.println(d1.d1);//null

		d1.d1=new Demo();

		System.out.println(d1.d1.x);

	}

}
```

Accessing static members of a class:
==========

--to access the static members from the main method, we have 3 options:

1. we can access directly without any ref variable (within the same class).

2.By using class name .(dot) // it is most recomended way.

3.By creating an object of that class and through object reference.

example:

Demo.java:

```
package com.masai;

public class Demo {

	static int x = 100;

	static void fun1() {
		System.out.println("inside fun1 of Demo");
	}



	public static void main(String[] args) {

//		System.out.println(x);
//		fun1();
```

```
//		System.out.println(Demo.x);
//		Demo.fun1();


		Demo d1 = new Demo();
		d1.fun1();

	}

}
```

accessing static member by using class name :-

--because when we use a class name inside our application that class context will be created.(and static memebers are loaded inside this context area)

accessing static member by using class object:-

example:

```
package com.masai;

public class Demo {

	static int x = 100;

	static void fun1() {
		System.out.println("inside fun1 of Demo");
	}

	int y =20;


	public static void main(String[] args) {

		Demo d1 = new Demo();

		System.out.println(d1.y);//20
		System.out.println(d1.x);//100

		d1= null;
```

```
            System.out.println(d1);

            System.out.println(d1.x);// 100
            System.out.println(d1.y);// NPE


        }

}
```

Note: each ref variable of a class have 2 parts, one part points to the heap area and another part points to the context area.

--so that we can access the static members with the ref variable of a class also.

--the part which is pointing to the heap area will be under the control of developer(we can assign another object or null) and the part which is pointing to the context area, will not be under the control of developer.

example:

Demo.java:

```
---------------

package com.masai;

public class Demo {

        static int x = 100;

        int y =20;


        public static void main(String[] args) {

            Demo d1 = new Demo();

            System.out.println(d1.x);//100
            System.out.println(d1.y);//20

            d1.x=200;
            d1.y=500;
```

```
        Demo d2 = new Demo();

        System.out.println(d2.x);//200
        System.out.println(d2.y);//20



    }

}
```

Note: non-static members are specific to each object, where as static members are sharable with all the object of same class, (single copy of the static member will be created.)

System.out.println:-

```
package com.masai;

public class Demo {

    static int x =100;

    static A a1 = new A();


    public static void main(String[] args) {

        System.out.println(Demo.x);

        Demo.a1.funA();//A

        System.out.println();//PrintStream

    }

}
```

--funA() belongs to A class, similarly println() method belongs to PrintStream class .

--this PrintStream class is statically defined inside the System class with 'out' variable.

```
//example of System class..
class System {

static  PrintStream out = new PrintStream();

}
```

Same class object as static member :
==========

```
package com.masai;

public class Demo {

	int x =100;


	static Demo d1 = new Demo();

	public static void main(String[] args) {


		Demo d1 = new Demo();

		System.out.println(d1.x);

		System.out.println(d1.d1.x);
		System.out.println(Demo.d1.x);

	}

}
```

methods in java:
==========

--Java does not support nested method defination, we can only call one method from another method.

--method is having 2 part :

1.method signature
2.method body

```
public void fun1() // method signature
{ // method body
//10000
//or Empty body

}
```

--in java we have two type sof methods:

1.normal or concreate method : method with body

2.abstract method : method without body

method with parameters:
---

example:

```
package com.masai;

public class Demo {

	//method declaration
	public void fun1(int x) {


		System.out.println("inside fun1 of Demo "+x);
	}

	public static void main(String[] args) {


		Demo d1 = new Demo();

		byte x=20;
```

```
		//int x=30;
		//method call
		d1.fun1(x);


	}

}
```

Example : method with ref type parameter:

Demo.java:

```
----------

package com.masai;

public class Demo {

	//method declaration
	public void fun1(A a1) {


		System.out.println("inside fun1 of Demo "+a1);
	}

	public static void main(String[] args) {


		Demo d1 = new Demo();

		A obj = new A();
		d1.fun1(obj);  // after the method execution A obj will not be eligible for GC
		d1.fun1(new A()); //after the method execution A obj will be eligible for GC

	}

}
```

example 2:

==========

```
package com.masai;

public class Demo {

	//method declaration
	public void fun1(A a1) {
		if(a1 != null) {

		System.out.println("inside fun1 of Demo "+a1);

		a1.funA();

		}else
			System.out.println("please don't pass null value");

	}

	public static void main(String[] args) {


		Demo d1 = new Demo();

		A obj = null;

		d1.fun1(obj);

	}

}
```

Polymorphism:
==========

--defining more than one functionality(method) with the same name inside a class.

there are 2 types of polymorphism:

1.static polymorphism (compile time polymorphism):-- more than one method with the same name but with diff parameter (which method will be executed will be decided at compile time), we achive static polymorphism by using method overloading..

2.dynamic polymorphism (runtime poly...): more than one method with same name and same parameter, dynamic polymorphism we achieve through inheritance (by using method overriding) , which method will be executed will be decided at runtime.

disadv of the static polymorphism:
==========

example 1:

```
package com.masai;

public class Demo {

	void fun1() {

		System.out.println("inside fun1 of Demo");
	}

	void fun1(int x) {

		System.out.println("inside fun1(int x) of Demo");
	}

	void fun1(float f) {

		System.out.println("inside fun1(float f) of Demo");
	}



	public static void main(String[] args) {

		Demo d1 = new Demo();

		d1.fun1(10.55f);



	}

}
```

example 2:

```
package com.masai;

public class Demo {

	void fun1() {

		System.out.println("inside fun1 of Demo");
	}



	void fun1(byte b) {

		System.out.println("inside fun1(byte b) of Demo");
	}



	public static void main(String[] args) {

		Demo d1 = new Demo();

		d1.fun1((byte)10);

	}

}
```

--the main drawback of the static polymorphism is, compiler may goes to the ambigious state , and will generate compilation error.

ex1:

```
package com.masai;

public class Demo {
```

```
	void fun1(int x, float y) {

		System.out.println("inside fun1(int x,float y) of Demo");
	}


	void fun1(float x,int y) {

		System.out.println("inside fun1(float x, int y) of Demo");
	}


	public static void main(String[] args) {

		Demo d1 = new Demo();

		d1.fun1(10,10); // CE


	}

}
```

example 2:

```
package com.masai;

public class Demo {

	void fun1(A a1) {

		System.out.println("inside fun1(A a1) of Demo");
	}


	void fun1(B b1) {

		System.out.println("inside fun1(B b1) of Demo");
```

```
	}


	public static void main(String[] args) {

		Demo d1 = new Demo();

		d1.fun1(null);


	}

}
```

method with return type:
==========

example 1:

```
package com.masai;

public class Demo {


	int fun1() {

		System.out.println("inside fun1 of Demo");

		byte x=10;

		return x;
	}


	public static void main(String[] args) {

		Demo d1 = new Demo();

		long l= d1.fun1();
		int i= d1.fun1();
```

```
		byte b= (byte)d1.fun1();



		System.out.println(i);

	}

}
```

method with Object as return type:

```
package com.masai;

public class Demo {


	A fun1() {

		System.out.println("inside fun1 of Demo");


		A a1=new A();

		return a1;
		//return null;
		//return new A();

	}


	public static void main(String[] args) {

		Demo d1 = new Demo();

		A obj= d1.fun1();

		obj.funA();

	}
```

```
}
```

example 2:

Demo.java:

```
package com.masai;

public class Demo {

	A fun1(String username, String password) {

		if(username.equals("admin") && password.equals("123")) {
			System.out.println("inside fun1 of Demo");
			return new A();
		}else
			return null;


	}

	public static void main(String[] args) {

		Demo d1 = new Demo();

		A obj= d1.fun1("admin","123");

		if(obj != null)
			obj.funA();
		else
			System.out.println("invalid username or password");


	}

}
```

Note: -- in java there is a class called Object class, it is a predefined class in java library which is a super class(parent class) of any classes in java world.

--so if a method is returning an object, then we can hold that obj either in its own class ref variable or its parent class ref variable

*******to the Object class variable we can assign/store any class object.**

```
Demo d1 = new Demo();

//Object obj= d1.fun1("admin","123");
A a1= d1.fun1("admin","123");
```