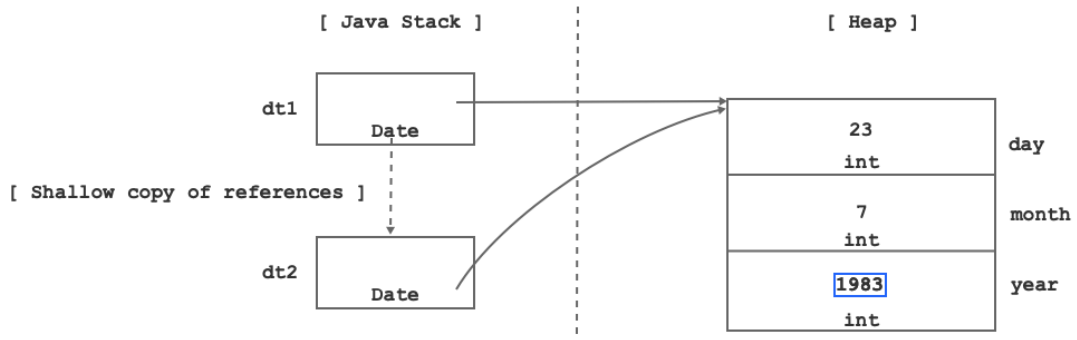
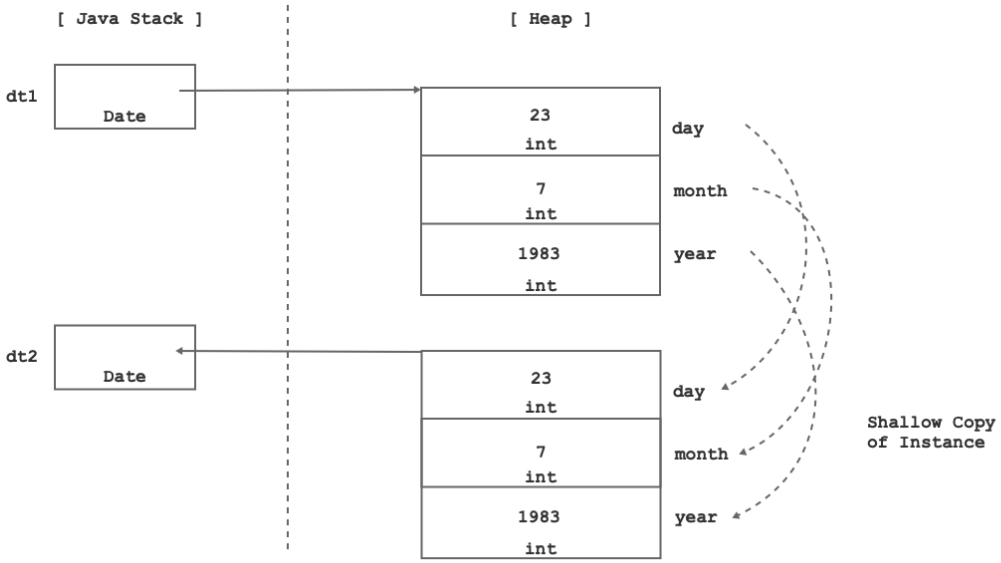
Day 17

- Shallow Copy of Date reference:



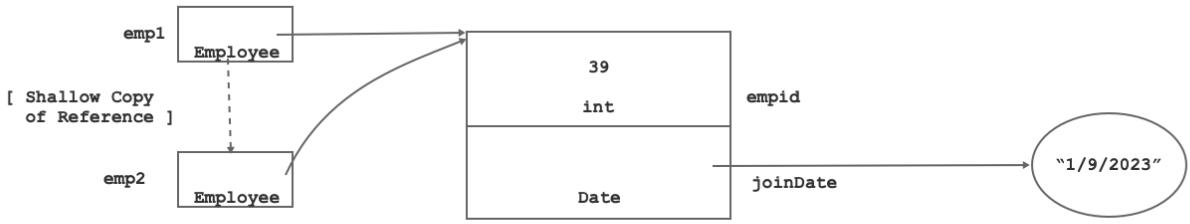

```
Date dt1 = new Date( 23, 7, 1983 );
Date dt2 = dt1; //Shallow Copy of references
```

- Shallow copy of Date instance:



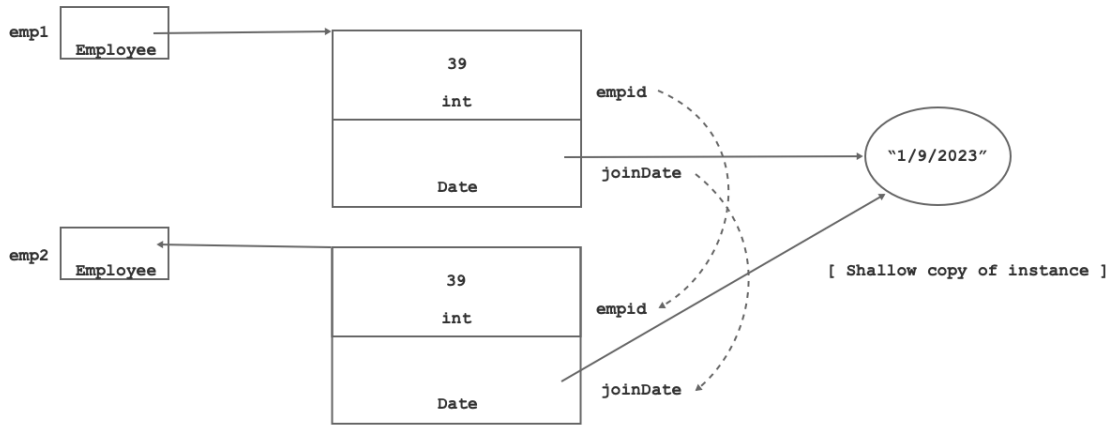

```
Date dt1 = new Date( 23, 7, 1983 );
Date dt2 = dt1.clone( );
```

- Shallow Copy of Employee reference:



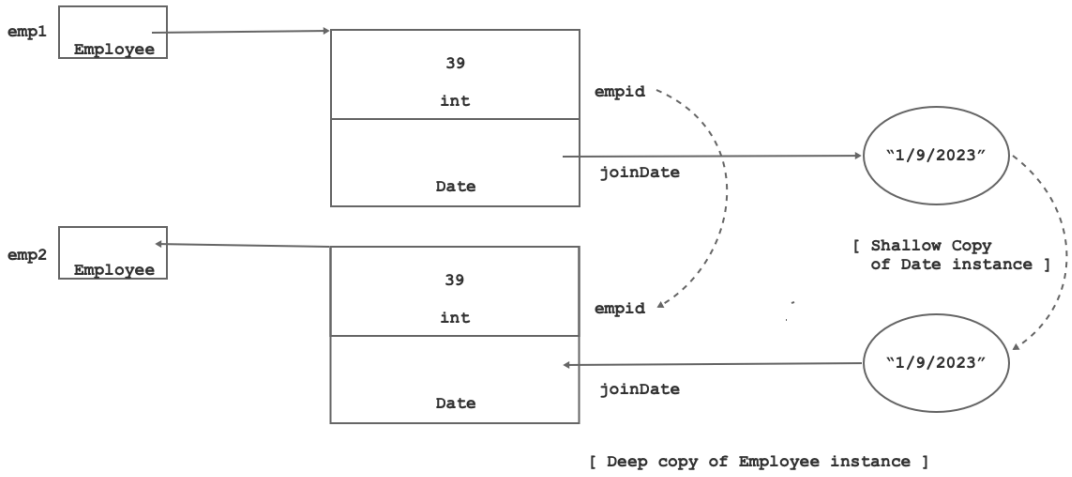

```
Employee emp1 = new Employee( 39, 1,9,2021);

Employee emp2 = emp1;//Shallow copy of references
```

- Shallow Copy of Employee instance:




```
Employee emp1 = new Employee( 39, 1,9, 2021 );
Employee emp2 = emp1.clone( );
```

- Deep Copy of Employee instance:




```
Employee emp1 = new Employee( 39, 1,9, 2021 );
Employee emp2 = emp1.clone( );
```

Fundamental interfaces of Core Java:

- java.lang.AutoCloseable
- java.io.Closeable
- java.lang.Cloneable
- java.lang.Comparable
- java.util.Comparator
- java.lang.Iterable
- java.util.Iterator

Comparable interface implementation

- Consider data for sorting

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	1980-12-17	800.00	-	20
7499	ALLEN	SALESMAN	7698	1981-02-20	1600.00	300.00	30
7521	WARD	SALESMAN	7698	1981-02-22	1250.00	500.00	30
7566	JONES	MANAGER	7839	1981-04-02	2975.00	-	20
7654	MARTIN	SALESMAN	7698	1981-09-28	1250.00	1400.00	30
7698	BLAKE	MANAGER	7839	1981-05-01	2850.00	-	30
7782	CLARK	MANAGER	7839	1981-06-09	2450.00	-	10
7788	SCOTT	ANALYST	7566	1982-12-09	3000.00	-	20
7839	KING	PRESIDENT	-	1981-11-17	5000.00	-	10
7844	TURNER	SALESMAN	7698	1981-09-08	1500.00	0.00	30
7876	ADAMS	CLERK	7788	1983-01-12	1100.00	-	20
7900	JAMES	CLERK	7698	1981-12-03	950.00	-	30
7902	FORD	ANALYST	7566	1981-12-03	3000.00	-	20
7934	MILLER	CLERK	7782	1982-01-23	1300.00	-	10

- If we want to sort array/collection of non primitive type using Arrays.sort() method then non primitive type must implement Comparable interface.
- Comparable is interface declared in java.lang package.
    - T -> the type of objects that this object may be compared to
- Method:
    - int compareTo(T other)
        - Returns a negative integer(-1) : current/calling object is less than the specified object.
        - Returns zero(0): current/calling object is equal to the specified object.
        - Returns a positive integer(1) : current/calling object is greater than the specified object.
- Comparable interface provides natural ordering(default ordering) of elements inside same class.
- This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method.

```
@NoArgsConstructor
@AllArgsConstructor
@Getter @Setter
//@ToString
```

```
public class Employee implements Comparable<Employee>{
  private int empNumber;
  private String empName;
  private String job;
  private int manager;
  private LocalDate hireDate;
  private float salary;
  private float commision;
  private int deptNumber;

  @Override
  public int compareTo(Employee other) {
    return this.empNumber - other.empNumber;
  }
  @Override
  public String toString() {
    return String.format("%-5d%-10s%-10s%-5d%-15s%-10.2f%-10.2f%-5d",
this.empNumber, this.empName, this.job, this.manager, this.hireDate,
this.salary, this.commision, this.deptNumber);
  }
}
```

Comparator interface implementation

- If we wan to sort array/collection of non primitive type using different criteria then we should implement Comparator interface.
- Comparator is functional interface declared in java.util package.
    - T -> the type of objects that may be compared by this comparator
- Method:
    - int compare(T o1, T o2)
        - Returns a negative integer(-1) : If first argument is less than the second argument.
        - Returns zero(0): If first argument is equal to the second argument.
        - Returns a positive integer(1) : If first argument is greater than the second argument.
- Comparator interface provides custom ordering of elements in different classes.

```
  import java.util.Comparator;

public class DeptNumberComparator implements Comparator<Employee> {
  @Override
  public int compare(Employee e1, Employee e2) {
    return e1.getDeptNumber() - e2.getDeptNumber();
  }
}
```

```
```

```
import java.util.Comparator;

public class HireDateComparator implements Comparator<Employee> {
  @Override
  public int compare(Employee e1, Employee e2) {
    return e1.getHireDate().compareTo(e2.getHireDate());
  }
}
```

```
import java.util.Comparator;
public class IdComparator implements Comparator<Person>{
  @Override
  public int compare(Person p1, Person p2) {
    if( p1 instanceof Student && p2 instanceof Student ) {
      Student s1 = (Student) p1;
      Student s2 = (Student) p2;
      return s1.getRollNumber() - s2.getRollNumber();
    }else if( p1 instanceof Employee && p2 instanceof Employee ) {
      Employee e1 = (Employee) p1;
      Employee e2 = (Employee) p2;
      return e1.getEmpid() - e2.getEmpid();
    }else if( p1 instanceof Student && p2 instanceof Employee ) {
      Student s1 = (Student) p1;
      Employee e2 = (Employee) p2;
      return s1.getRollNumber() - e2.getEmpid();
    }else {
      Employee e1 = (Employee) p1;
      Student s2 = (Student) p2;
      return e1.getEmpid() - s2.getRollNumber();
    }
  }
}
```

Iterable & Iterator

- Traversing using iterator

```
import java.util.Iterator;
import java.util.LinkedList;

public class Program {
  public static void main(String[] args) {
    LinkedList<Integer> list = new LinkedList<>();
    list.addLast(10);
    list.addLast(20);
    list.addLast(30);

    Integer element = null;
```

```
    Iterator<Integer> itr = list.iterator();
    while( itr.hasNext()) {
      element = itr.next();
      System.out.print(element+"    ");
    }
    System.out.println();
  }
}
```

- Traversing using foreach loop

```
    public static void main(String[] args) {
    LinkedList<Integer> list = new LinkedList<>();
    list.addLast(10);
    list.addLast(20);
    list.addLast(30);

    for( Integer element : list )
      System.out.println( element );
  }
```

- We can use foreach loop on Array and any instance which implements java.lang.Iterable interface.
- Iterable is interface declared in java.lang package.
    - T -> the type of elements returned by the iterator
- Implementing this interface allows an object to be the target of the "for-each loop" statement.
- It is introduced in JDK 1.5
- Methods:
    - java.util.Iterator iterator()
    - default Spliterator spliterator()
    - default void forEach(Consumer<? super T> action)
- Iterator is interface declared in java.util package.
    - E -> the type of elements returned by this iterator
- It is introduced in JDK 1.2
- Methods:
    - boolean hasNext()
    - E next()
    - default void remove()
    - default void forEachRemaining(Consumer<? super E> action)

```

```

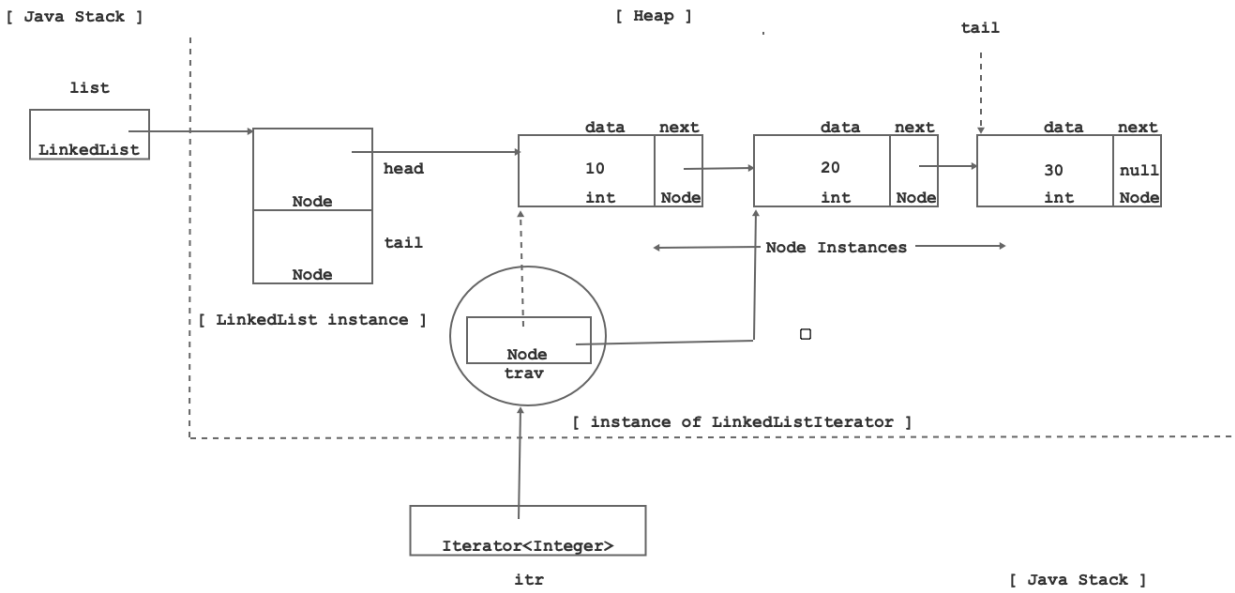
```
import java.util.Iterator;
class Node{
  int data;
  Node next = null;
  public Node( int data ) {
    this.data = data;
  }
}
```

```
class LinkedList implements Iterable<Integer>{
  private Node head = null;
  private Node tail = null;

  public boolean empty( ) {
    return this.head == null;
  }
  public void addLast( int element ) {
    Node newNode = new Node( element );
    if( this.empty() )
      this.head = newNode;
    else
      this.tail.next = newNode;
    this.tail = newNode;
  }
  @Override
  public Iterator<Integer> iterator() {
    Iterator<Integer> itr = new LinkedListIterator( this.head );
//Upcasting
    return itr;
  }

}
```

```
class LinkedListIterator implements Iterator<Integer>{
  private Node trav;

  public LinkedListIterator(Node head) {
    this.trav = head;
  }
  @Override
  public boolean hasNext() {
    return this.trav != null;
  }
  @Override
  public Integer next() {
    int data = trav.data;
    trav = trav.next;
    return data;
```

```
    }
  }
```

```
public class Program {
  public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.addLast( 10 );
    list.addLast( 20 );
    list.addLast( 30 );

    for( int element : list )
      System.out.println( element );
  }
}
```



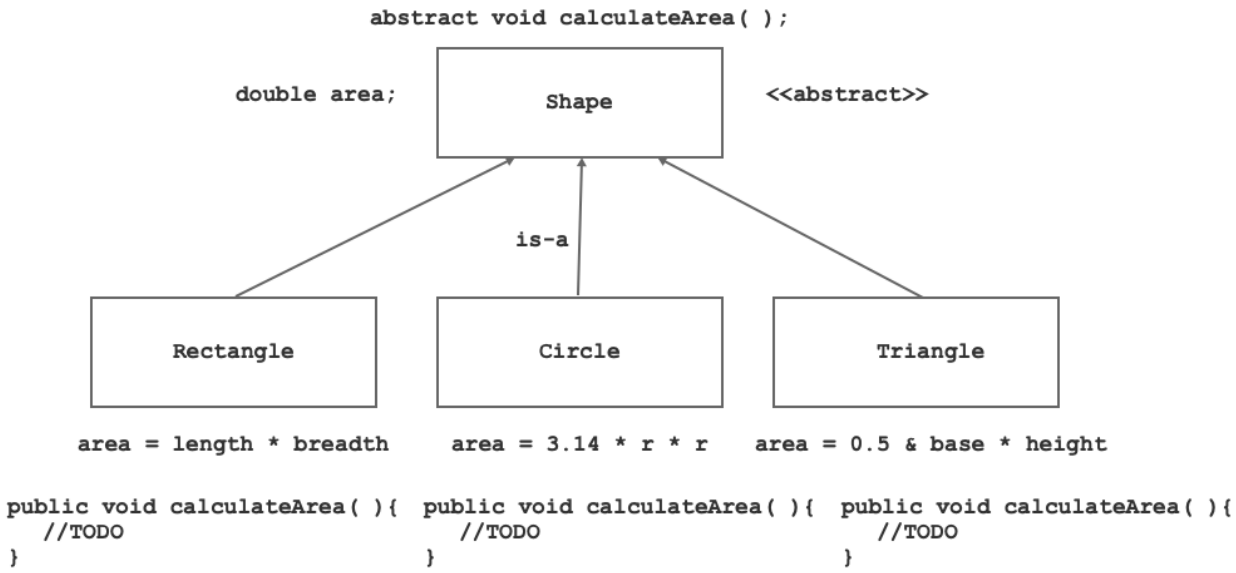
- Process of visiting elements in a collection is calleed as travsering.
- In C++ if we use any object as a pointer then such object is called as smart pointer.
- Iterator is a smart pointer which is used to traverse collection.
- Any class which implements Iterable interface is considered as travsersible using foreach loop.

Abstract class versus Interface

Abstract class

- If "is-a" relationship is exist between super type & sub type and if we want to use same method design/signature in all the sub classes then super type should be abstract class.

- Consider following diagram



- Using abstract class we can group elements/instances of realated types together.

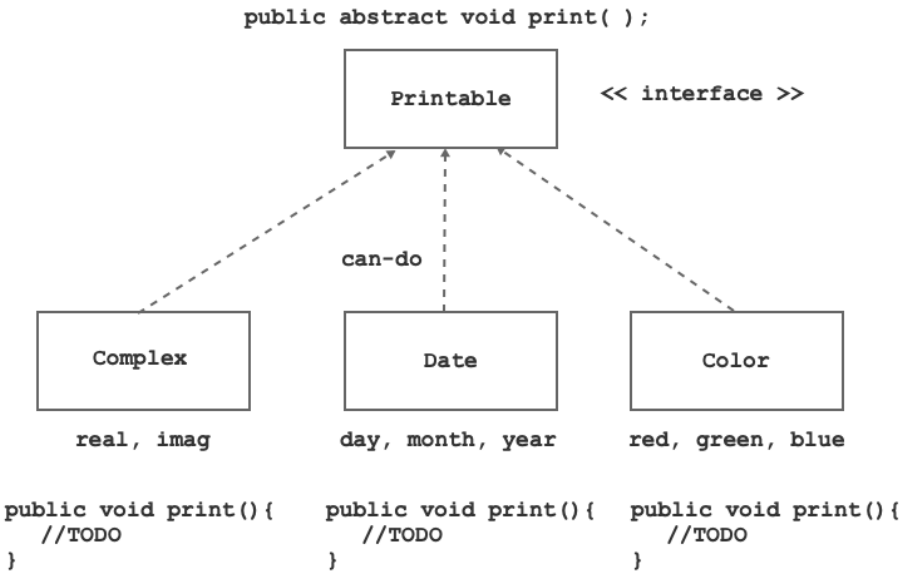
```
Shape[] arr = new Shape[ 3 ];
arr[ 0 ] = new Rectangle(); //Upcasting
arr[ 1 ] = new Circle();  //Upcasting
arr[ 2 ] = new Triangle();  //Upcasting
```

- Super class of abstract class can be either concrete class / abstract class. Abstract class can extend only one abstract class / concrete class.
- We can define constructor inside abstract class.
- Abstract may/may not contain abstract method.
- When state is involved in super type then it should be abstract class

Interface

- If "is-a" relationship is not exist between super type & sub type("can-do" relationship is exist) and if we want to use same method design in all the sub classes then super type should be interface.

- Consider following diagram



- Using interface, we can group element/instances of unrelated type together.

```
Printable[] arr = new Printable[ 3 ];
arr[ 0 ] = new Complex( );  //Upcasting
arr[ 1 ] = new Date( );  //Upcasting
arr[ 2 ] = new Color( );  //Upcasting
```

- Super type of interface must be interface. Interface can extend multiple interfaces.
- We can not define constructor inside interface.
- Interface methods are by default public and abstract.
- When state is not involved in super type then it should be interface.

Nested class

- We can define class inside scope of another class. It is called as nested class.
- Consider following code:

```
//Top level class
class Outer{  //Outer.class
  //Nested class
  class Inner{  //Outer$Inner.class
      //TODO: Implementation part
  }
}
```

- Access modifier of top level class can be either package level private or public only but we can use any access modifier on nested class.
- Types of nested class
  - Non static nested class / Inner class

- Static nested class

Inner class

- In Java, non static nested class is called as inner class.
- If implementation of nested class depends on top level class then we should declare nested class non static.

```
class LinkedList implements Iterable<Integer>{

    private Node head = null;
    private Node tail = null;

//TODO: other implementation

    @Override
    public Iterator<Integer> iterator() {
        Iterator<Integer> itr = new LinkedListIterator( this.head );
//Upcasting
        return itr;
    }

    class LinkedListIterator implements Iterator<Integer>{
        private Node trav;
        public LinkedListIterator(Node head) {
            this.trav = head;
        }
  //TODO: Other implementation
    }
}
```

- Note: For the simplicity, consider non static nested class a non static method of the class.
- Consider following code:

```
class Outer{
  class Inner{
    //TODO
  }
}
```

  - Instantiation of top level class:

```
Outer out = new Outer();
```

  - Instantiation of inner class:

```
Outer out = new Outer();
Outer.Inner in = out.new Inner();
```

```
Outer.Inner in = new Outer().new Inner();
```

- Inside non static nested class i.e inner class, we can not define static members(fields & methods). But if we want to declare any static field then it must be final.
- Using instance, we can access members of non static nested class inside method of top level class.
- Consider following example:

```
class Outer{
  private int num1 = 10;    //OK
  private static int num2 = 20; //OK

  class Inner{
    private int num3 = 30;  //OK
    //private static int num4 = 40; //Not OK
    private final static int num4 = 40; //OK
  }

  public void print( ) {
    System.out.println("Num1    :   "+this.num1);
    System.out.println("Num2    :   "+Outer.num2);
    Inner in = new Inner();
    System.out.println("Num3    :   "+in.num3);
    System.out.println("Num4    :   "+Inner.num4);
  }
}
public class Program {
  public static void main(String[] args) {
    Outer out = new Outer();
    out.print();
  }
}
```

- Without instance, we can access all the members of top level class inside method of non static nested class i.e inner class.
- Consider following code:

```
class Outer{
  private int num1 = 10;
  private static int num2 = 20;
```

```
  class Inner{
    private int num3 = 30;
    private final static int num4 = 40;
    public void print( ) {
      System.out.println("Num1  :   "+num1);    //OK
      System.out.println("Num2  :   "+num2);    //OK
      System.out.println("Num3  :   "+this.num3);
      System.out.println("Num4  :   "+Inner.num4);
    }
  }
}
public class Program {
  public static void main(String[] args) {
    Outer.Inner in = new Outer().new Inner();
    in.print();
  }
}
```

- Consider following code snippet

```
  class Outer{
  private int num1 = 10;
  class Inner{
    private int num1 = 20;
    public void print( ) {
      int num1 = 30;
      System.out.println("Num1  :   "+Outer.this.num1); //10
      System.out.println("Num1  :   "+this.num1);   //20
      System.out.println("Num1  :   "+num1);    //30
    }
  }
}
public class Program {
  public static void main(String[] args) {
    Outer.Inner in = new Outer().new Inner();
    in.print();
  }
}
```

Static Nested Class

- If we declare nested class as a static then it is called as static nested class.
- If implementation of nested class do not depend on top level class then we should declare nested class static.
- Consider following code:

```
```

```
  class LinkedList implements Iterable<Integer>{
    static class Node{
      int data;
      Node next = null;
      public Node( int data ) {
        this.data = data;
      }
    }

    private Node head = null;
    private Node tail = null;
  }
```

- In Java, we can not declare top level class static but we can declare nested class static.

```
static class Outer{ //Not OK
  //TODO
}
```

```
//Top Level class
class Outer{  //Outer.class
    //Static Nested class
    static class Inner{ //OK: Outer$Inner.class
      //TODO
    }
}
```

- Note: For simplicity, consider static nested class as a static method of the class.
- Instantiation of Top level class:

```
Outer out = new Outer( );
```

- Instantiation of static nested class:

```
Outer.Inner in = new Outer.Inner();
```

- We can declare/define static members inside static nested class.
- Using instance, we can access all the members of top level class inside method of top level class.

```
```

```
class Outer{
  private int num1 = 10;
  private static int num2 = 20;

  static class Inner{
    private int num3 = 30;  //OK
    private static int num4 = 40;   //OK
  }
  public void print( ) {
    System.out.println("Num1    :   "+this.num1);
    System.out.println("Num2    :   "+Outer.num2);
    Inner in = new Inner();
    System.out.println("Num3    :   "+in.num3);
    System.out.println("Num4    :   "+Inner.num4);
  }
}
public class Program {
  public static void main(String[] args) {
    Outer out = new Outer();
    out.print();
  }
}
```

- We can access static members of the top level class inside method of static nested class directly. But to access non static member of the class we must use instance of the class.

```
class Outer{
  private int num1 = 10;
  private static int num2 = 20;

  static class Inner{
    private int num3 = 30;  //OK
    private static int num4 = 40;   //OK

    public void print( ) {
      //System.out.println("Num1    :   "+num1);    //Not OK
      Outer out = new Outer();
      System.out.println("Num1  :   "+out.num1);    //OK
      System.out.println("Num2  :   "+num2);    //OK
      System.out.println("Num3  :   "+this.num3);
      System.out.println("Num4  :   "+Inner.num4);
    }
  }
}
public class Program {
  public static void main(String[] args) {
    Outer.Inner in = new Outer.Inner();
    in.print();
  }
}
```

Local Class

- We can define class inside scope of another method. It is called as local class.
- Local class is also called as method local class.
- we can not use reference / instance of local class outside method.
- Types of local class:
  - Method local inner class.
  - Method local anonymous inner class.

Method local inner class.

- In Java, we can not declare local variable as well as local class static.
- Since method local class is non static, it is also called as method local inner class.

```
public class Program {
  public static void main(String[] args) {
    //Method local inner class
    class Complex{  //Program$1Complex.class
      private int real = 10;
      private int imag = 20;
      public Complex() {
        this(0,0);
      }
      public Complex(int real, int imag) {
        this.real = real;
        this.imag = imag;
      }
      @Override
      public String toString() {
        return this.real+"  "+this.imag;
      }
    }
    Complex c1 = new Complex();
    System.out.println(c1.toString());
  }
}
```

Method local anonymous inner class

- In Java, we can define class without name. It is called as anonymous class.
- We can define anonymous class inside method only. Hence it is called as anonymous inner class.
- If we want to define anonymous inner class then we must use new operator.

```
public class Program {
  public static void main(String[] args) {
    //Object obj;       //obj is refernce of java.lang.Object class
```

```
    //new Object( );    //Anonymous instance of java.lang.Object class
    //Object obj = new Object();    //Instance with object reference

    Object obj = new Object( ) {    //Program$1.class
      private String message = "Hello World";
      @Override
      public String toString() {
        return this.message;
      }
    };

    String str = obj.toString();
    System.out.println( str );
  }
}
```

```
abstract class Shape{
  protected double area;
  public double getArea() {
    return area;
  }
  public abstract void calculateArea();
}

public class Program {
  public static void main(String[] args) {
    Shape sh = new Shape() {
      private double radius = 10.5;
      @Override
      public void calculateArea() {
        this.area = Math.PI * Math.pow(radius, 2);
      }
    };
    sh.calculateArea();
    System.out.println("Area    :   "+sh.getArea());
  }
}
```

```
interface Printable{
  void print( );
}
public class Program {
  public static void main(String[] args) {
    Printable p = new Printable() {
      @Override
      public void print() {
        System.out.println("Hello World!!");
      }
    };
```

```
      p.print();
    }
  }
```

Reflection

Annotation