Object Oriented Programming with Java

Sandeep Kulange

Nested Class

- In Java, we can define class inside scope of another class. It is called nested class.
- Nested class represents encapsulation.

```
//Top-Level class
class Outer{     //Outer.class
    //Nested class
    class Inner{     //Outer$Inner.class
       //TODO
    }
}
```

- Access modifier of top level class can be either package level private or public only.
- We can use any access modifier on nested class.
- Types of nested class:
    1. Non static nested class / Inner class
    2. Static nested class

Non Static Nested Class

- Non static nested class is also called as inner class.
- If implementation of nested class depends on implementation of top level class then nested class should be non static.
- **Implementation Hint :** For the simplicity, consider non static nested class as non static method of class.

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

```
Instantiation of top level class:
Outer out = new Outer( );
```

```
* Instantiation of top level class:
- method 1
Outer out = new Outer( );
Outer.Inner in = out.new Inner( );
```

```
- method 2
Outer.Inner in = new Outer( ).new Inner( );
```

Non Static Nested Class

- Top level class can contain static as well as non static members.
- Inside non static nested class we can not declare static members.
- If we want to declare any field static then it must be final.
- Using instance, we can access members of non static nested class inside method of top level class.
- Without instance, we can use all the members of top level class inside method of non static nested class.
- But if we want refer member of top level class, inside method of non static nested class then we should use "TopLevelClassName.this" syntax.

Non Static Nested Class

```
class Outer{
    private int num1 = 10;

    public 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 static then it is simply called as static nested class.
- We can declare nested class static but we can not declare top level class static.
- If implementation of nested class do not depends on implementation of top level class then we should declare nested class static.
- **Implementation hint:** For simplicity consider static nested class as a static method of a class.

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

```
* Instantiation of top level class:
Outer out = new Outer( );

* Instantiation of static nested class:
Outer.Inner in = new Outer.Inner();
```

Static Nested Class

- Static nested class can contain static members.
- Using instance, we can access all the members of static nested class inside method of top level class.
- If we want to use non static members of top level class inside method of static nested class then it is mandatory to create instance of top level class.

Nested Class

```
class LinkedList implements Iterable<Integer>{
    static class Node{
        //TODO
    }
    //TODO
    class LinkedListIterator implements Iterator<Integer>{
        //TODO
    }
}
```

Local Class

- In Java, we can define class inside scope of another method. It is called local class / method local class.

- Types of local class:
  1. Method local inner class
  2. Method local anonymous inner class.

Method Local Inner Class

- In Java, we can not declare local variable /class static hence local class is also called as local inner class.
- We can not use reference/instance of method local class outside method.

```
public class Program {  //Program.class
    public static void main(String[] args) {
        class Complex{  //Program$1Complex.class
            private int real = 10;
            private int imag = 20;
            public void print( ) {
                System.out.println("Real Number :    "+this.real);
                System.out.println("Imag Number :    "+this.imag);
            }
        }
        Complex c1 = new Complex();
        c1.print();
    }
}
```

Method local anonymous inner class.

- In java, we can create instance without reference. It is called anonymous instance.
- Example:
- ```
  new Object( );
  ```
- We can define a class without name. It is called anonymous class.
- If we want to define anonymous class then we should use new operator.
- We can create anonymous class inside method only hence it is also called as method local anonymous class.
- We can not declare local class static hence it is also called as method local anonymous inner class.
- To define anonymous class, we need to take help of existing interface / abstract class / concrete class.
  ```

Method local anonymous inner class.

- Consider anonymous inner class using concrete class.

```
public static void main(String[] args) {
    //Object obj;   //obj => reference
    //new Object( );    // new Object( ) => Anonymous instance
    //Object obj = new Object( );//Instance with reference
    Object obj = new Object( ) {    //Program$1.class
        private String message = "Hello";
        @Override
        public String toString() {
            return this.message;
        }
    };
    String str = obj.toString();
    System.out.println(str);
}
```

Method local anonymous inner class.

- Consider anonymous inner class using abstract class:

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

```
public class Program {
    public static void main(String[] args) {
        Shape sh = new Shape() {
            private double radius = 10;
            @Override
            public void calculateArea() {
                this.area = Math.PI * Math.pow(this.radius, 2);
            }
        };

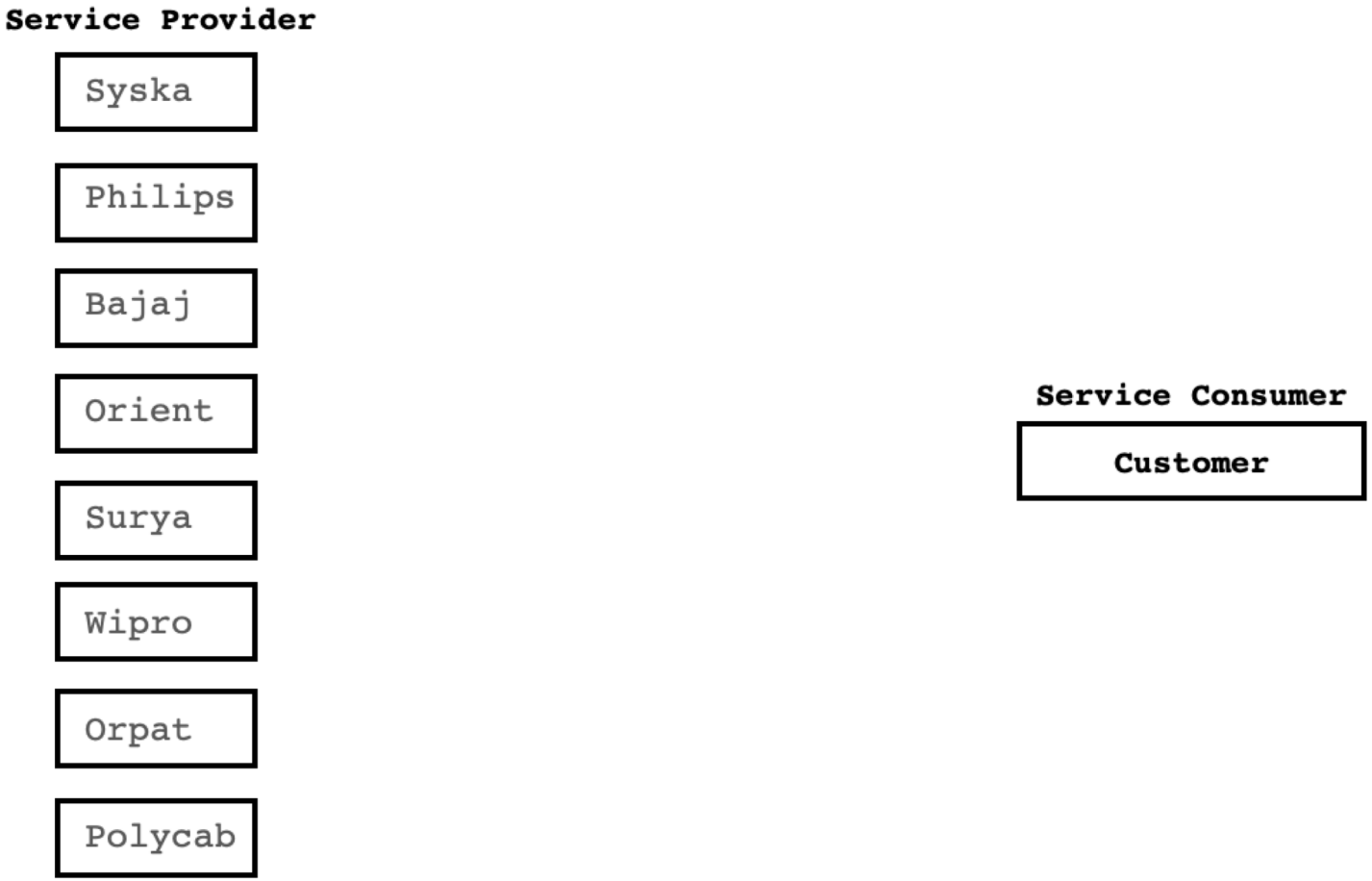
        sh.calculateArea();
        System.out.println("Area    :    "+sh.getArea());
    }
}
```

Method local anonymous inner class.

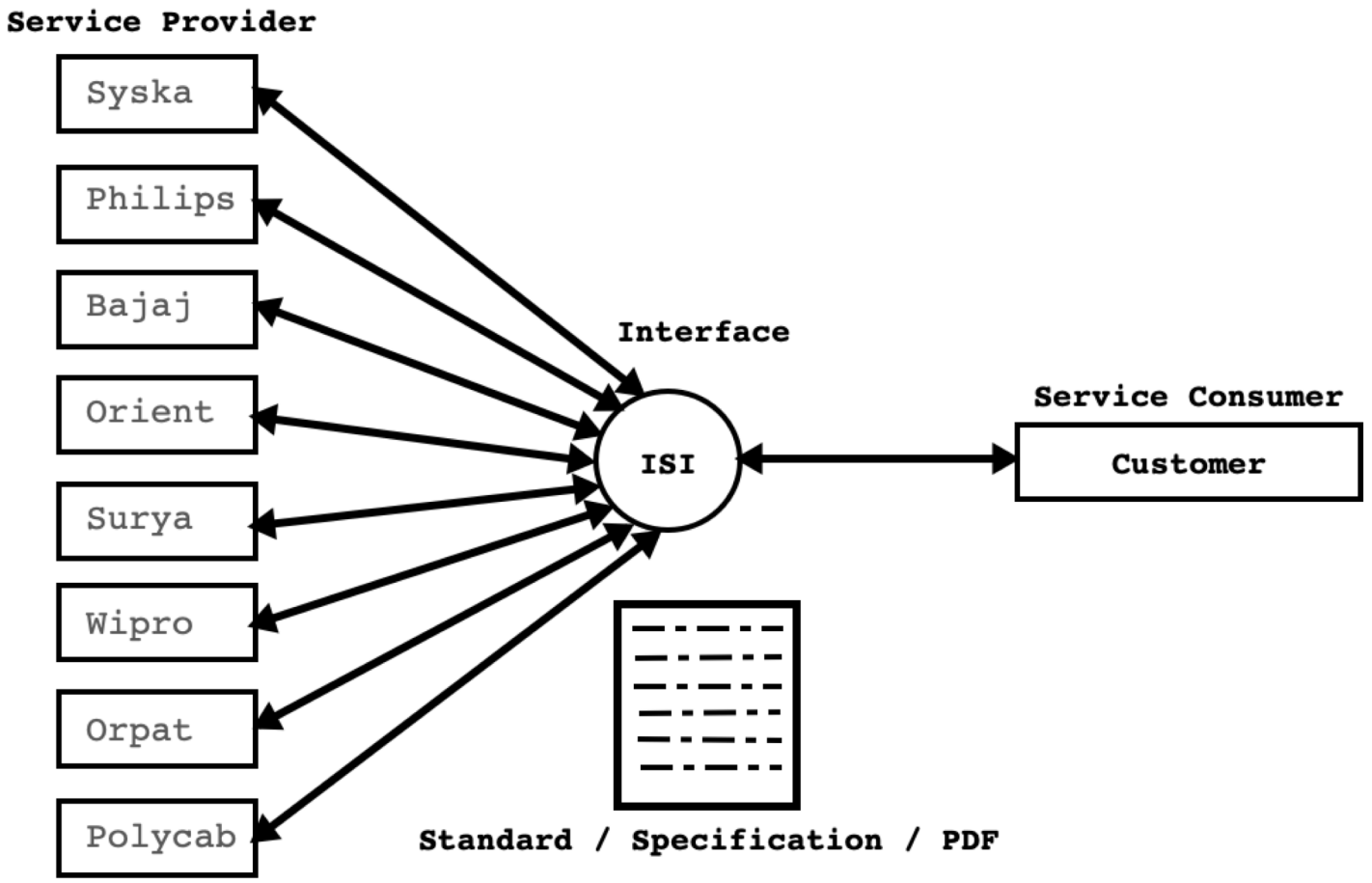
- Consider anonymous inner class using interface.

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


Service Provider
Syska
Philips
Bajaj
Orient
Surya
Wipro
Orpat
Polycab
Service Consumer
Customer

Interface


Service Provider
Syska
Philips
Bajaj
Orient
Surya
Wipro
Orpat
Polycab
Interface
ISI
Service Consumer
Customer
Standard / Specification / PDF

Interface

- Set of rules are called specification/standard.
- It is a contract between service consumer and service provider.
- If we want to define specification for the sub classes then we should define interface.
- Interface is non primitive type which helps developer:
  1. To build/develop trust between service provider and service consumer.
  2. To minimize vendor dependency.
- interface is a keyword in Java.

```
interface Printable{
    //TODO
}
```

Interface

- Interface can contain:
    1. Nested interface
    2. Field
    3. Abstract method
    4. Default method
    5. Static method
- Interfaces cannot have constructors.
- We can create reference of interface but we can not create instance of interface.
- We can declare fields inside interface. Interface fields are by default public static and final.
- We can write methods inside interface. Interface methods are by default considered as public and abstract.

```
interface Printable{
    int number = 10; //public static final int number = 10;
    void print( ) ; //public abstract void print( ) ;
}
```

Interface

- If we want to implement rules of interface then we should use implements keyword.
- It is mandatory to override, all the abstract methods of interface otherwise sub class can be considered as abstract.

```
interface Printable{
    int number = 10;
    void print( );
}
```

* Solution 1

```
abstract class Test implements Printable{
}
```

* Solution 2

```
class Test implements Printable{
    @Override
    public void print( ){
        //TODO
    }
}
```

Interface Implementation Inheritance

```
interface Printable{
    int number = 10;
    //public static final int number = 10;
    void print( ) ;
    //public abstract void print( ) ;
}
class Test implements Printable{
    @Override
    public void print() {
        System.out.println("Number   :    "+Printable.number);
    }
}
public class Program {
    public static void main(String[] args) {
        Printable p = new Test( );  //Upcasting
        p.print();  //Dynamic Method Dispatch
    }
}
```

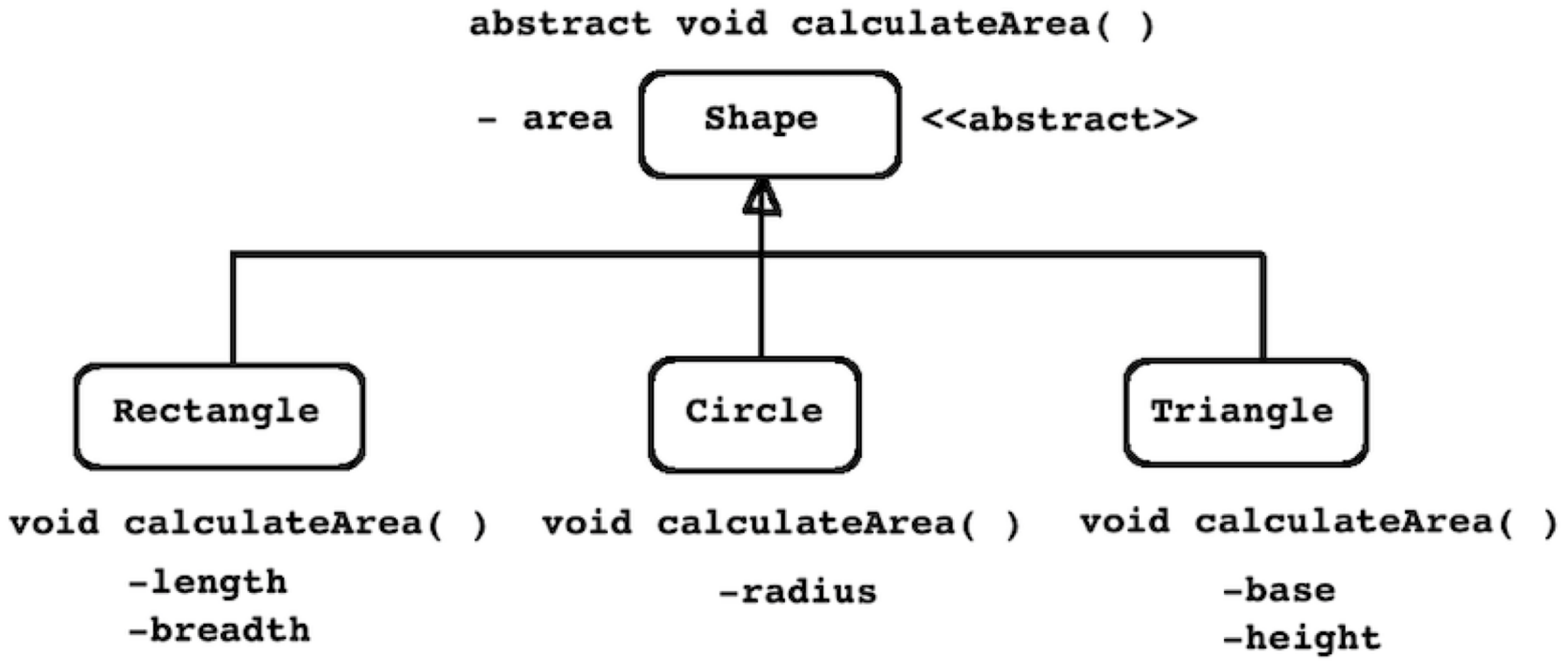
Interface Syntax

```
Interfaces     :   I1, I2, I3
Classes        :   C1, C2, C3
1. I2 implements I1                          //Not OK
2. I2 extends I1                             //OK : Interface Inheritance
3. I3 extends I1, I2                         //OK : Multiple Interface Inheritance
4. I1 extends C1                             //Not OK
5. I1 implements C1                          //Not OK
6. C1 extends I1                             //Not OK
7. C1 implements I1                          //OK : Interface implementation Inherita
8. C1 implements I1, I2                      //OK : Multiple Interface implementatio ritance
9. C2 implements C1                          //Not OK
10. C2 extends C1                            //OK : Implementation inheritance
11. C3 extends C1, C2                        //NOT OK : Multiple Implementation inhe e
12. C2 implements I1 extends C1              //NOT OK
13. C2 extends C1 implements I1              //OK
14. C2 extends C1 implements I1, I2, I3      //OK
```

Types of inheritance

- **Interface Inheritance**
  - During inheritance if super type and sub type is interface then it is called interface inheritance.
    1. Single Inheritance(Valid in Java)
    2. Multiple Inheritance(Valid in Java)
    3. Hierarchical Inheritance(Valid in Java)
    4. Multilevel Inheritance(Valid in Java)
- **Implementation Inheritance**
  - During inheritance if super type and sub type is class then it is called implementation inheritance.
    1. Single Inheritance(Valid in Java)
    2. Multiple Inheritance(Invalid in Java)
    3. Hierarchical Inheritance(Valid in Java)
    4. Multilevel Inheritance(Valid in Java)

Abstract Class



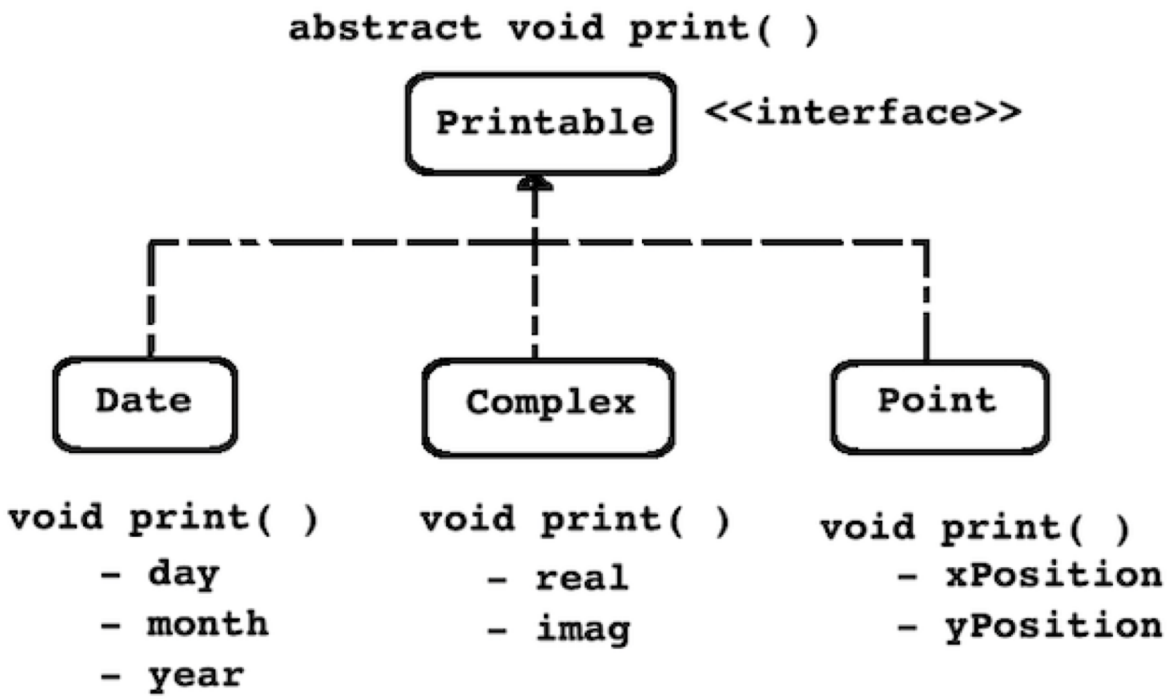

```
Shape[] arr = new Shape[ 3 ];
arr[ 0 ] = new Rectangle( ); //Upcasting
arr[ 1 ] = new Circle( ); //Upcasting
arr[ 2 ] = new Triangle( ); //Upcasting
```

Abstract Class

1. If "is-a" relationship is exist between super type and sub type and if we want same method design in all the sub types then super type must be abstract.
2. Using abstract class, we can group instances of related type together
3. Abstract class can extend only one abstract/concrete class.
4. We can define constructor inside abstract class.
5. Abstract class may or may not contain abstract method.

- **Hint :** In case of inheritance if state is involved in super type then it should be abstract.

Interface




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

Interface

1. If "is-a" relationship is not exist between super type and sub type and if we want same method design in all the sub types then super type must be interface.

2. Using interface, we can group instances of unrelated type together.

3. Interface can extend more than one interfaces.

4. We can not define constructor inside interface.

5. By default methods of interface are abstract.

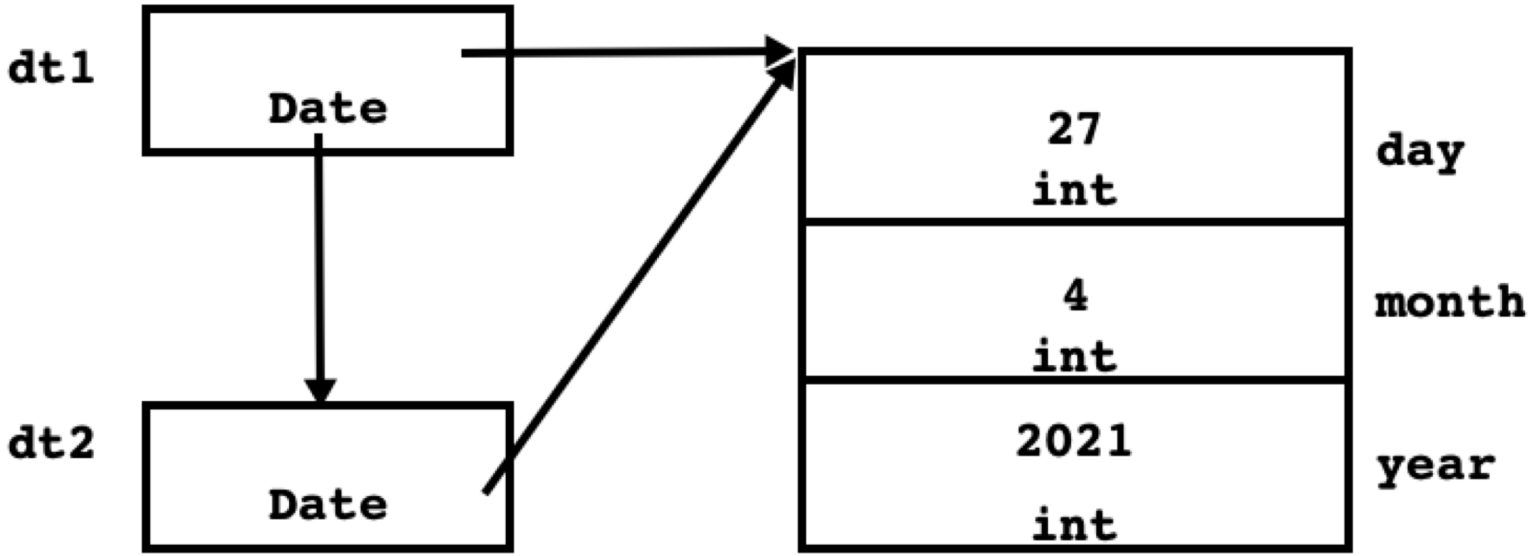
- **Hint :** In case of inheritance if state is not involved in super type then it should be interface.

Commonly Used Interfaces

1. java.lang.AutoCloseable
2. java.io.Closeable
3. java.lang.Cloneable
4. java.lang.Comparable
5. java.util.Comparator
6. java.lang.Iterable
7. java.util.Iterator
8. java.io.Serializable

Cloneable Interface Implementation

- `Date dt1 = new Date( 27, 4, 2021 );`
- `Date dt2 = dt1;  //Shallow Copy Of References`

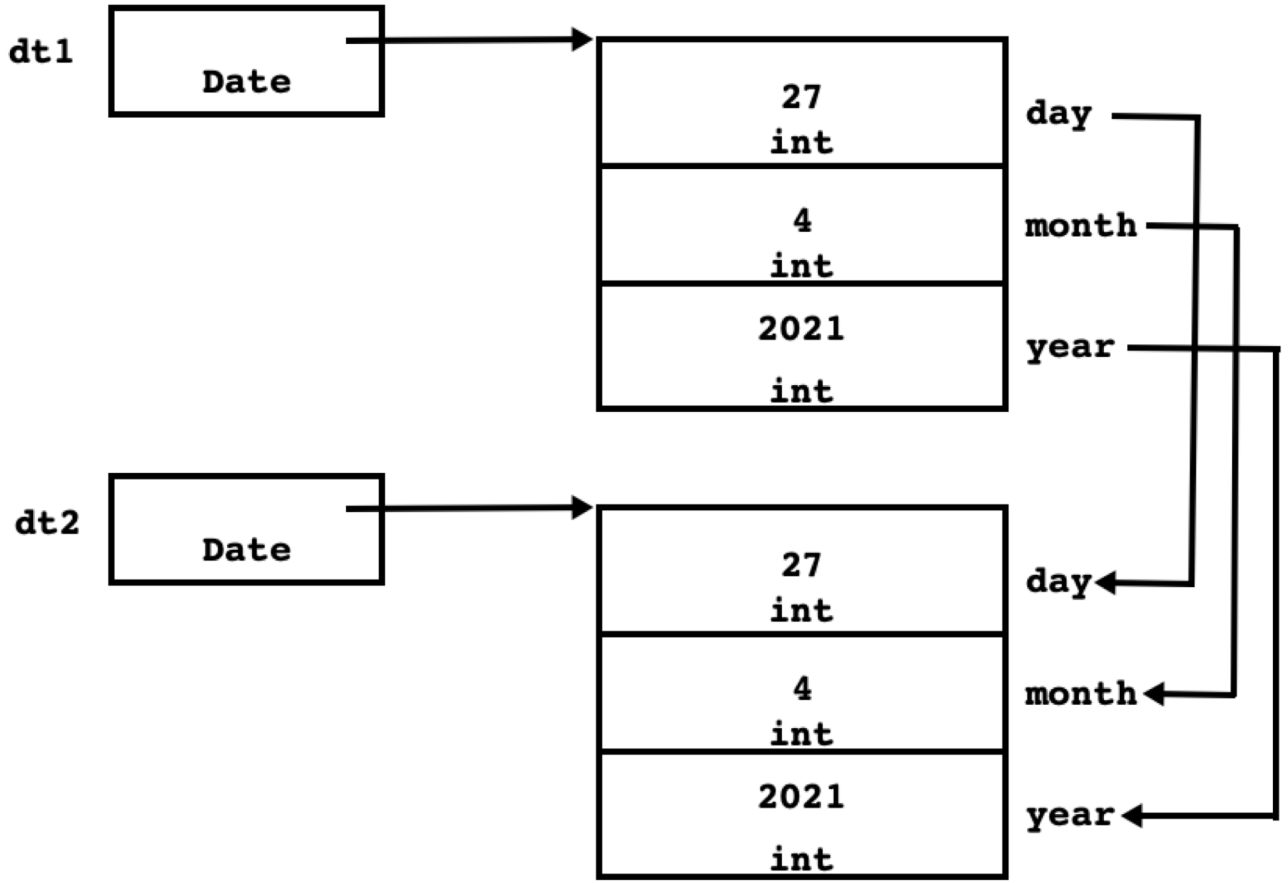


Cloneable Interface Implementation

- If we want to create new instance from existing instance then we should use clone method.
- clone( ) is non final native method of java.lang.Object class.
- Syntax:
  - **protected native Object clone( ) throws CloneNotSupportedException**
- Inside clone() method, if we want to create shallow copy instance then we should use super.clone( ) method.
- Cloneable is interface declared in java.lang package.
- Without implementing Cloneable interface, if we try to create clone of the instance then clone() method throws CloneNotSupportedException.

Cloneable Interface Implementation

- `Date dt1 = new Date( 27, 4, 2021 );`
- `Date dt2 = dt1.clone( ); //Shallow Copy Of Instance`



Marker Interface

- An interface which do not contain any member is called marker interface. In other words, empty interface is called as marker interface.
- Marker interface is also called as tagging interface.
- If we implement marker interface then Java compiler generates metadata for the JVM, which help JVM to clone/serialize or marshal state of object.
- Example:
    1. java.lang.Cloneable
    2. java.util.EventListener
    3. java.util.RandomAccess
    4. java.io.Serializable
    5. java.rmi.Remote

Comparable

- It is interface declared in java.lang package.
- "**int compareTo(T other)**" is a method of java.lang.Comparable interface.
- If state of current object is less than state of other object then compareTo() method should return negative integer(-1).
- If state of current object is greater than state of other object then compareTo() method should return positive integer(+1).
- If state of current object is equal to state of other object then compareTo() method should return zero(0).
- If we want to sort, array of non primitive type which contains all the instances of same type then we should implement Comparable interface.

Comparator

- It is interface declared in java.util package.
- "**int compare(T o1, T o2)**" is a method of java.util.Comparator interface.
- If state of current object is less than state of other object then compare() method should return negative integer(-1).
- If state of current object is greater than state of other object then compare() method should return positive integer(+1).
- If state of current object is equal to state of other object then compare() method should return zero(0).
- If we want to sort, array of instances of non primitive of different type then we should implement Comparator interface.

Iterable and Iterator Implementation

- Iterable<T> is interface declared in java.lang package.
- Implementing this interface allows an object to be the target of the "for-each loop" statement.
- It is introduced in JDK 1.5
- Methods of java.lang.Iterable interface:
    1. Iterator<T> iterator()
    2. default Spliterator<T> spliterator()
    3. default void forEach(Consumer<? super T> action)

Iterable and Iterator Implementation

- Iterator<E> is interface declared in java.util package.
- It is used to traverse collection in forward direction only.
- It is introduced in JDK 1.2
- Methods of java.util.Iterator interface:
    1. boolean hasNext()
    2. E next()
    3. default void remove()
    4. default void forEachRemaining(Consumer<? super E> action)

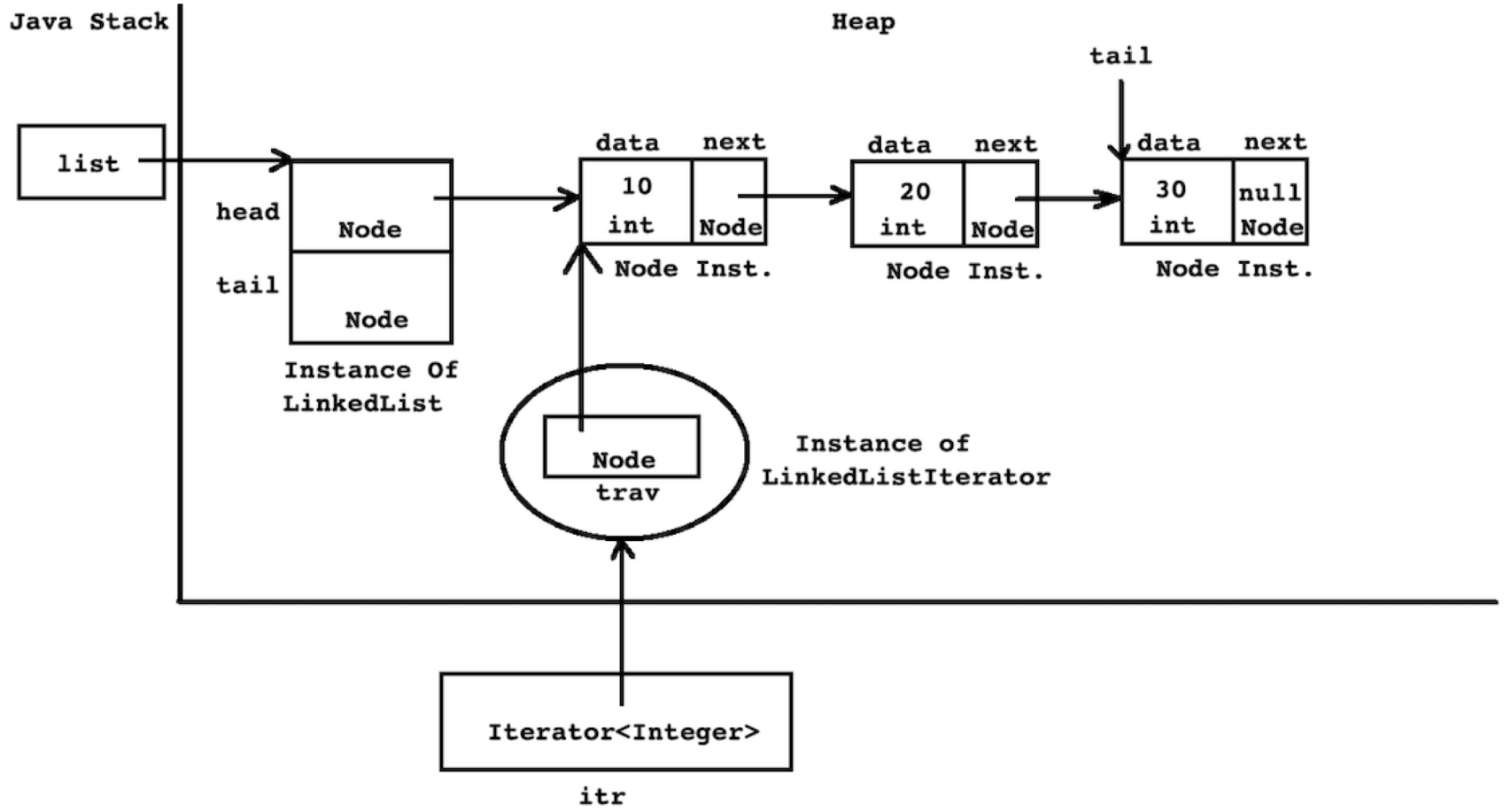
Iterable and Iterator Implementation

```
LinkedList<Integer> list = new LinkedList<>( );
    list.add(10);
    list.add(20);
    list.add(30);

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

```
foreach loop implicitly work as follows
Integer element = null;
Iterator<Integer> itr = list.iterator();
while( itr.hasNext()) {
    element = itr.next();
    System.out.println(element);
}
```

Iterable and Iterator Implementation



Thank You.

[sandeepkulange@sunbeaminfo.com]