Day 16

Synthetic Constructs in Java

- Reference: https://www.baeldung.com/java-synthetic

Generic Method

- We can define generic method using java.lang.Object class.

```
public class Program {
  public static void print( Object object) {
    System.out.println( object );
  }
  public static void main(String[] args) {
    Program.print( true );
    Program.print( 123 );
    Program.print( 'A' );
    Program.print( 1234567 );
    Program.print( 3.142f );
    Program.print( 123.4567d );
    Program.print( "Good Morning!!" );
    Program.print( new Date() );
  }
}
```

- Generic method using generics

```
public class Program {
  public static <T> void print( T value) {
    System.out.println( value );
  }
  public static void main(String[] args) {
    Program.print( true );
    Program.print( 123 );
    Program.print( 'A' );
    Program.print( 1234567 );
    Program.print( 3.142f );
    Program.print( 123.4567d );
    Program.print( "Good Morning!!" );
    Program.print( new Date() );
  }
}
```

- We can specify bounded type parameter for method:

```

```

```
public class Program {
  public static <T extends Number> void print( T value) {
    System.out.println( value );
  }
  public static void main(String[] args) {
    //Program.print( true );    //Not OK
    Program.print( 123 );
    //Program.print( 'A' );//Not OK
    Program.print( 1234567 );
    Program.print( 3.142f );
    Program.print( 123.4567d );
    //Program.print( "Good Morning!!" );//Not OK
    //Program.print( new Date() );//Not OK
  }
}
```

Type Erasure

- Consider generic type without upper bound.

```
class Box<T>{
  private T data;

  public Box() {
  }
  public Box(T data) {
    this.data = data;
  }
  public T getData() {
    return data;
  }
  public void setData(T data) {
    this.data = data;
  }
  @Override
  public String toString() {
    return this.data.toString();
  }
}
```

- Below is the code with type erasure.

```
class Box{
  private Object data;

  public Box() {
  }
  public Box(Object data) {
```

```
    this.data = data;
  }
  public Object getData() {
    return data;
  }
  public void setData(Object data) {
    this.data = data;
  }
  @Override
  public String toString() {
    return this.data.toString();
  }
}
```

- Consider generic type with upper bound(Number).

```
class Box<T extends Number>{
  private T data;

  public Box() {
  }
  public Box(T data) {
    this.data = data;
  }
  public T getData() {
    return data;
  }
  public void setData(T data) {
    this.data = data;
  }
  @Override
  public String toString() {
    return this.data.toString();
  }
}
```

- Below is the code with type erasure.

```
class Box{
  private Number data;

  public Box() {
  }
  public Box(Number data) {
    this.data = data;
  }
  public Number getData() {
    return data;
  }
```

```
  public void setData(Number data) {
    this.data = data;
  }
  @Override
  public String toString() {
    return this.data.toString();
  }
}
```

Bridge method

- Consider following code:

```
class Box<T>{
  private T data;

  public Box() {
  }
  public Box(T data) {
    this.data = data;
  }
  public void setData(T data) {
    this.data = data;
  }
  @Override
  public String toString() {
    return this.data.toString();
  }
}
class Sample extends Box<Integer>{

  public Sample() {
    super();
  }
  public Sample(Integer data) {
    super(data);
  }
  /*
  //Method added by compiler to achive dynamic method dispatch
  public void setData(Object data) {    //Bridge method
    super.setData((Integer)data);
  } */
  @Override
  public void setData(Integer data) {
    super.setData(data);
  }
}
```

```

```

```
public static void main3(String[] args) {
    Sample s = new Sample();
    Box b = s;  //Upcasting
    b.setData(123); //OK
    System.out.println(b.toString()); //123
}
```

```
public static void main(String[] args) {
    Sample s = new Sample();
    Box b = s;  //Upcasting
    b.setData( "Hello" );   //ClassCastException
    System.out.println(b.toString());
}
```

Restrictions on Generics

- Cannot Instantiate Generic Types with Primitive Types

```
Stack<int> s1 = new Stack<>();  //Not OK
Stack<Integer> s1 = new Stack<>();  //OK
```

- Cannot Declare Static Fields Whose Types are Type Parameters

```
class Box<T>{
  private static T data;  //Not OK
  //TODO: Getter and Setter
}
```

- Cannot Use Casts or instanceof with Parameterized Types

```
public static void printRecord( List<String> list ){
  if( list instanceof ArrayList<String> ){  //Not OK
    ArrayList<String> arrayList = ( ArrayList<String> ) list; //OK
  }
}
public static void main( String[] args ){
  ArrayList<String> arrayList = new ArrayList<>( );
  list.add( "DAC" );
  ist.add( "DMC" );
  list.add( "DESD" );
  Program.print( arrayList );

  LinkedList<String> linkedList = new LinkedList<>( );
  list.add( "RED" );
```

```
    ist.add( "GREEN" );
    list.add( "BLUE" );
    Program.print( linkedList );
  }
```

- Cannot Create Arrays of Parameterized Types

```
Box<Integer> box = new Box<>(); //OK
Box<Integer>[ ] arr = new Box<>[ 5 ]; //Not OK
```

- Cannot Create, Catch, or Throw Objects of Parameterized Types

```
class QueueFullException<T> extends Throwable // compile-time error
{ /* ... */ }
```

```
class MathException<T> extends Exception // compile-time error
{ /* ... */ }
```

```
  public static <T extends Exception, J> void execute(List<J> jobs) {
    try {
        for (J job : jobs)
            // ...
    } catch (T e) {   // compile-time error
        // ...
  }
```

- A class cannot have two overloaded methods that will have the same signature after type erasure.

```
public class Example {
    public void print(Set<String> strSet) { }
    public void print(Set<Integer> intSet) { }
}
```

Why main method is static?

- JVM is responsible for calling main method.
- Consider following scenarios if main method is non static:
  - If class is abstract.

```
```

```
abstract class Program{
  public void main( String[] args ){
    //TODO
  }
}
```

- If class is concrete but constructor is private:

```
class Program{
  private Program( ){
    //TODO
  }
  public void main( String[] args ){
    //TODO
  }
}
```

  - If class is concrete and class contains only public parameterized constructor

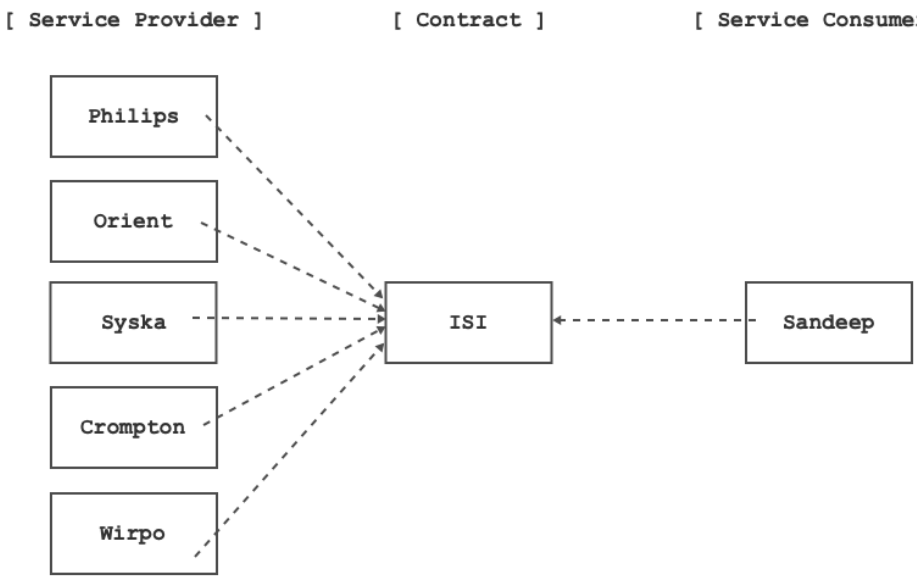
```
class Program{
  public Program( String s1, int i1, float f1, double d1 ){
    //TODO
  }
  public void main( String[] args ){
    //TODO
  }
}
```

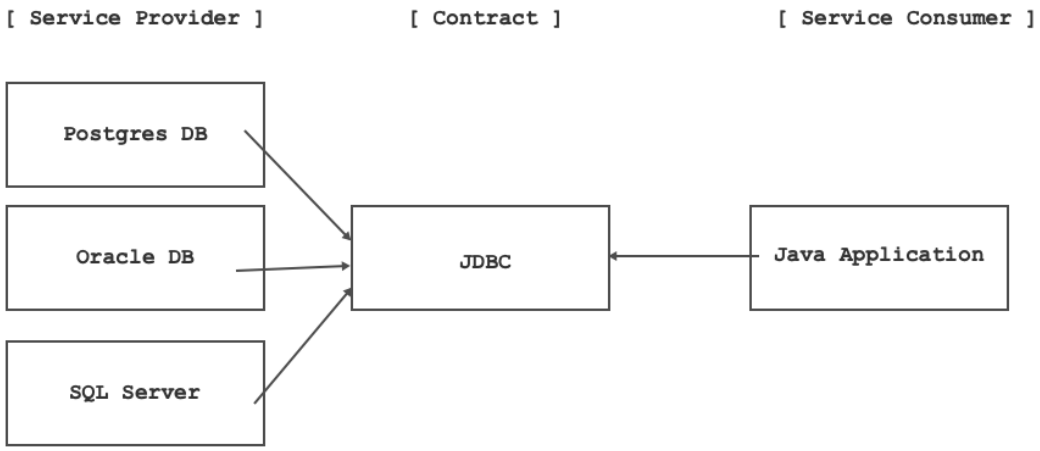
- To overcome above problems, main method is declared as static.

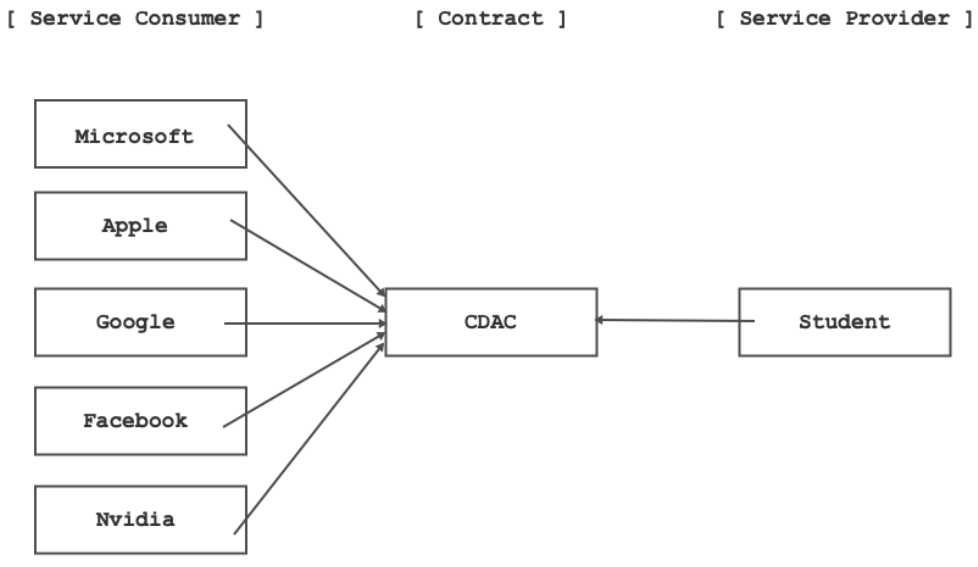
Fragile Base class problem

- If we make changes in the body super class then we must recompile super class as well as all its sub classes. This problem is called as fragile base class problem.
- We can solve fragile base class problem by defining super type as interface.

Abstraction using interface


[Service Provider]
[Contract]
[Service Consumer]
Philips
Orient
Syska
Crompton
Wirpo
ISI
Sandeep


[Service Provider]
[Contract]
[Service Consumer]
Postgres DB
Oracle DB
SQL Server
JDBC
Java Application


[Service Consumer]
[Contract]
[Service Provider]
Microsoft
Apple
Google
Facebook
Nvidia
CDAC
Student

- Set of rules are called as specification/standard.
- If we want to define specifications for sub classes in java then we should define interface.
- Interface is contract between service provider and service consumer
- Advantage of interface:
    - To build trust between service provider and service consumer.
    - It helps to achive abstraction
    - It help to minimize service provider dependency
- Non primitive types in Java:
    - interface
    - class
    - enum
    - array
- interface is a keyword in java
- In Java, interface can contain:
    - Nested type
    - Field
    - Abstract method
    - Default method
    - Static interface method
- interface fields are implicitly considered as public static and final

```
interface Printable{
  //int value;  //Error: The blank final field value may not have been
initialized
  int value = 123;
  //public static final int value = 123;
}
```

- interface methods are implicitly considered as public and abstract.

```
interface Printable{
  //void print( ) { }   //Error: Abstract methods do not specify a
body
  void print( );
  //public abstract void print( );
}
```

- We can not instantiate interface but we can create reference of interface.

```
public static void main(String[] args) {
    Printable p = null; //OK
    p = new Printable();    //Not OK
}
```

- We can not define constructor inside interface
- Consider folllowing code:

```
interface Printable{    //Contract
  int value = 123;
  //public static final int value = 123;

  void print( );
  //public abstract void print( );
}
class Test implements Printable{    //Service Provider
  @Override
  public void print() {
    System.out.println("Value   :   "+Printable.value);
  }
}
public class Program {  //Service Consumer
  public static void main(String[] args) {
    Printable p = null; //OK
    p = new Test( );    //Upcasting
    p.print();//Dynamic method dispatch
  }
}
```

Syntax to use interface

- Interfaces: I1, I2, I3
- Classes: C1, C2, C3
  - I2 implements C1 //Not OK
    - Super type of interface must be interface
  - I2 implements I1 //Not OK
    - Interface can extend another interface
  - I2 extends I1 //OK
  - I3 extends I1, I2 //OK
    - Interface can extend more than one interface. It is callled multiple interface inheritance
  - C1 extends I1;
    - Class can implement interface
  - C1 implements I1 //OK
  - C1 implements I1, I2 //OK
    - Class can implement more than one interfaces. It is called as multiple interface implementation inheritance.

- C2 implements C1 //Not OK
  - Class can extend another class
- C2 extends C1 //OK
- C3 extends C1, C2 //Not OK
  - Class can not extend more than once class. In short class do not support multiple implementation inheritance.
- C2 implements I1 extends C1 //Not OK
  - Class should first extend class and then implement interface
- C2 extends C1 implements I1 //OK
- C2 extends C1 implements I1, I2 //OK

Interface fields syntax

```
interface A{
    int num1 = 10;
    int num4 = 40;
    int num5 = 70;
}
interface B{
    int num2 = 20;
    int num4 = 50;
    int num5 = 80;
}
interface C extends A, B{   //Multiple interface inheritance
    int num3 = 30;
    int num4 = 60;
}
public class Program {
    public static void main(String[] args) {
        System.out.println("A.num5  :   "+A.num5);  //70
        System.out.println("B.num5  :   "+B.num5);  //80
        //System.out.println("C.num5    :   "+C.num5);  //Error: The field
C.num5 is ambiguous
    }
    public static void main2(String[] args) {
        System.out.println("A.Num4  :   "+A.num4);  //40
        System.out.println("B.Num4  :   "+B.num4);  //50
        System.out.println("C.Num4  :   "+C.num4);  //60
    }
    public static void main1(String[] args) {
        System.out.println("Num1    :   "+A.num1);
        System.out.println("Num1    :   "+C.num1);

        System.out.println("Num2    :   "+B.num2);
        System.out.println("Num2    :   "+C.num2);

        System.out.println("Num3    :   "+C.num3);
    }
}
```

Interface method syntax:

```
interface A{
    void f1();
}
interface B{
    void f2();
}
interface C extends A, B{
    void f3();
}
class D implements C{
    @Override
    public void f1() {
        System.out.println("D.f1");
    }
    @Override
    public void f2() {
        System.out.println("D.f2");
    }
    @Override
    public void f3() {
        System.out.println("D.f3");
    }
}
public class Program {
    public static void main(String[] args) {
        D d = new D();
        d.f1();//OK

        A a = new D();
        a.f1();//OK


        B b = new D();
        b.f2();//OK

        C c = new D();
        c.f1(); //OK
        c.f2(); //OK
        c.f3(); //Ok
    }
}
```

```
interface A{
    void f1();
    void f3();
}
interface B{
    void f2();
```

```
    void f3();
}
class C implements A, B{
    @Override
    public void f1() {
        System.out.println("C.f1");
    }
    @Override
    public void f2() {
        System.out.println("C.f2");
    }
    @Override
    public void f3() {
        System.out.println("C.f3");
    }
}
public class Program {
    public static void main(String[] args) {
        A a = new C();
        a.f1();
        a.f3();

        B b = new C();
        b.f2();
        b.f3();
    }
}
```

How to override some of the methods of interface

```
interface Printable{
    void f1();
    void f2();
    void f3();
    void f4();
}
abstract class AbstractPrintable implements Printable{
    @Override public void f1() {    }
    @Override public void f2() {    }
    @Override public void f3() {    }
}
class A extends AbstractPrintable{
    @Override
    public void f1() {
        System.out.println("A.f1");
    }
}
class B extends AbstractPrintable{
    @Override
    public void f2() {
        System.out.println("B.f2");
```

```
    }
}
class C extends AbstractPrintable{
    @Override
    public void f3() {
        System.out.println("C.f3");
    }
}
public class Program {
    public static void main(String[] args) {
        Printable p = null;

        p = new A();
        p.f1(); //A.f1

        p = new B();
        p.f2(); //B.f2

        p = new C();
        p.f3(); //C.f3
    }
}
```

Types of inheritance

- Interface inheritance
  - Single inheritance(Allowed in java)
  - Multiple inheritance(Allowed in java)
  - Hierarchical inheritance(Allowed in java)
  - Multilevel inheritance(Allowed in java)
- implementation inheritace
  - Single inheritance(Allowed in java)
  - Multiple inheritance(Not Allowed in java)
  - Hierarchical inheritance(Allowed in java)
  - Multilevel inheritance(Allowed in java)

Default interface method

- If we want to make changes in the interface at runtime then we should use default method.
- We can not provide body to the abstact method but it is mandatory to provide body to the default method.
- It is mandatory to override abstract method but it is optional to override default method.

```
interface A{
  void f1( );
  default void f2( ){
    //TODO
  }
```

```
}
class B implements A{
  @override
  public void f1( ){
    //TODO
  }
}
```

```
interface A{
  void f1( );
  default void f2( ){
    //TODO
  }
}
interface B{
  void f1( );
  default void f3( ){
    //TODO
  }
}
class C implements A, B{
  @override
  public void f1( ){
    //TODO
  }
}
```

```
interface A{
  void f1( );
  default void f2( ){
    //TODO
  }
}
interface B{
  void f1( );
  default void f2( ){
    //TODO
  }
}
class C implements A, B{
  @override
  public void f1( ){
    //TODO
  }
  @Override
  public void f2( ){  //mandatory to override
    //TODO
  }
}
```

- Consider following code:

```
interface Collection {
  void acceptRecord();

  int[] toArray();

  void printRecord();

  static void swap( int[] arr ) {
    int temp = arr[ 0 ];
    arr[ 0 ] = arr[ 1 ];
    arr[ 1 ] = temp;
  }
  default void sort() {
    int[] arr = this.toArray();
    for (int i = 0; i < arr.length - 1; i++) {
      for (int j = 0; j < arr.length - i - 1; j++) {
        if (arr[j] > arr[j + 1]) {

          int[] temp = new int[ ] { arr[j], arr[ j + 1 ] };
          Collection.swap(temp);
          arr[j] = temp[ 0 ];
          arr[ j + 1 ] = temp[ 1 ];
        }
      }
    }
  }
}

class Array implements Collection {
  private int[] arr;

  public Array() {
    this(5);
  }

  public Array(int size) {
    this.arr = new int[size];
  }

  @Override
  public void acceptRecord() {
    try (Scanner sc = new Scanner(System.in)) {
      for (int index = 0; index < this.arr.length; ++index) {
        System.out.print("Enter element :   ");
        this.arr[index] = sc.nextInt();
      }
    }
  }
```

```
  @Override
  public int[] toArray() {
    return this.arr;
  }

  @Override
  public void sort() {
    for( int i = 0; i < this.arr.length - 1; ++ i ) {
      for( int j = i + 1; j < this.arr.length; ++ j ) {
        if( this.arr[ i ] > this.arr[ j ] ) {
          int[] temp = new int[ ] { arr[ i ], arr[ j ]};
          Collection.swap(temp);
          arr[ i ] = temp[ 0 ];
          arr[ j ] = temp[ 1 ];
        }
      }
    }
  }

  @Override
  public void printRecord() {
    System.out.println(Arrays.toString(this.arr));
  }
}

public class Program {
  public static void main(String[] args) {
    Collection c = new Array();
    c.acceptRecord();
    c.sort();
    c.printRecord();
  }
}
```

- Static interface methods are helper methods that we can use inside default method as well as inside sub class. But we can not override it inside sub class.

Functional interface

- An interface which can contain Single Abstract Method (SAM) is called as Functional interface / SAM interface.
- Example:
  - java.lang.Runnable
  - java.util.Comparator
  - java.util.function.Predicate
  - java.util.function.Consumer
  - java.util.function.Supplier
  - java.util.function.Function
- Consider following code:

```
@FunctionalInterface
interface A{
  void f1();
}
```

```
@FunctionalInterface
interface A{
  void f1();
  default void f2( ){
  }
}
```

```
@FunctionalInterface
interface A{
  void f1();
  default void f2( ){
  }
  static void f3( ){
  }
}
```

```
@FunctionalInterface
interface A{
  void f1();
  default void f2( ){
  }
  default void f3( ){
  }
  static void f4( ){
  }
  static void f5( ){
  }
}
```

Shallow copy, Deep copy

- Process of copying contents from variable into another variable as it is, is called shallow copy.

```
int num1 = 10;
int num2 = num1;  //Shallow Copy
```

```
```

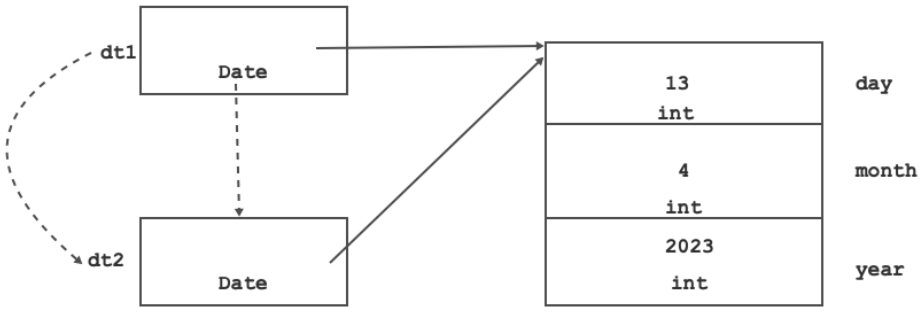
```
Date dt1 = new Date( 13,4,2023);
Date dt2 = dt1; //Shallow Copy of references
```

- If we want to create new instance from existing instance then we should use clone method.
- clone is non final and native method of java.lang.Object class:
- Syntax:
    - protected native Object clone( )throws CloneNotSupportedException
- Inside clone method, if we want to create shallow copy of instance then we should use super.clone();
- Without implementing Cloneable interface, if we try to create clone() of the instane then clone method throws CloneNotSupportedException.
- Marker interface:
    - An interface which do not contain any member is called as marker / tagging interface.
    - Marker interface are used to generate metadata. It helps JVM to perform some operations e.g to do clone, serializing state of java instance etc.
    - Example:
        - java.lang.Cloneable
        - java.util.EventListener
        - java.util.RandomAccess
        - java.rmi.Remote

```
class Date implements Cloneable{
  private int day;
  private int month;
  private int year;

  public Date(int day, int month, int year) {
    this.day = day;
    this.month = month;
    this.year = year;
  }

  public void setDay(int day) {
    this.day = day;
  }

  public void setMonth(int month) {
    this.month = month;
  }

  public void setYear(int year) {
    this.year = year;
  }
```

```
  @Override
  public Date clone( )  {
    try {
      Date other = (Date) super.clone();
      return other;
    } catch (CloneNotSupportedException e) {
      throw new InternalError(e);
    }
  }

  @Override
  public String toString() {
    return this.day+" / "+this.month+" / "+this.year;
  }
}
```

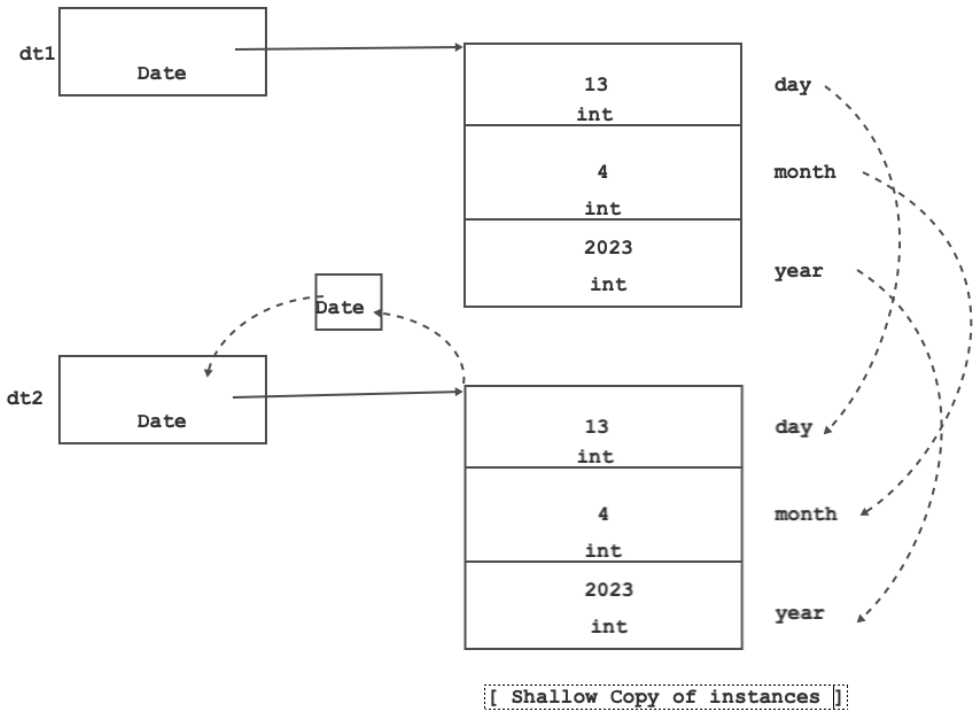
```
public class Program {
  public static void main1(String[] args) {
    Date dt1 = new Date(13, 4,2023);
    Date dt2 = dt1; //Shallow copy of references
    //System.out.println( dt1 == dt2 ); //true
  }
}
```



Shallow copy of reference

```
  public static void main(String[] args) {
    try {
      Date dt1 = new Date(13, 4,2023);
      Date dt2 = dt1.clone();
      dt2.setDay(23);
      dt2.setMonth(7);
      dt2.setYear(1983);

      System.out.println(dt1);
      System.out.println(dt2);
```

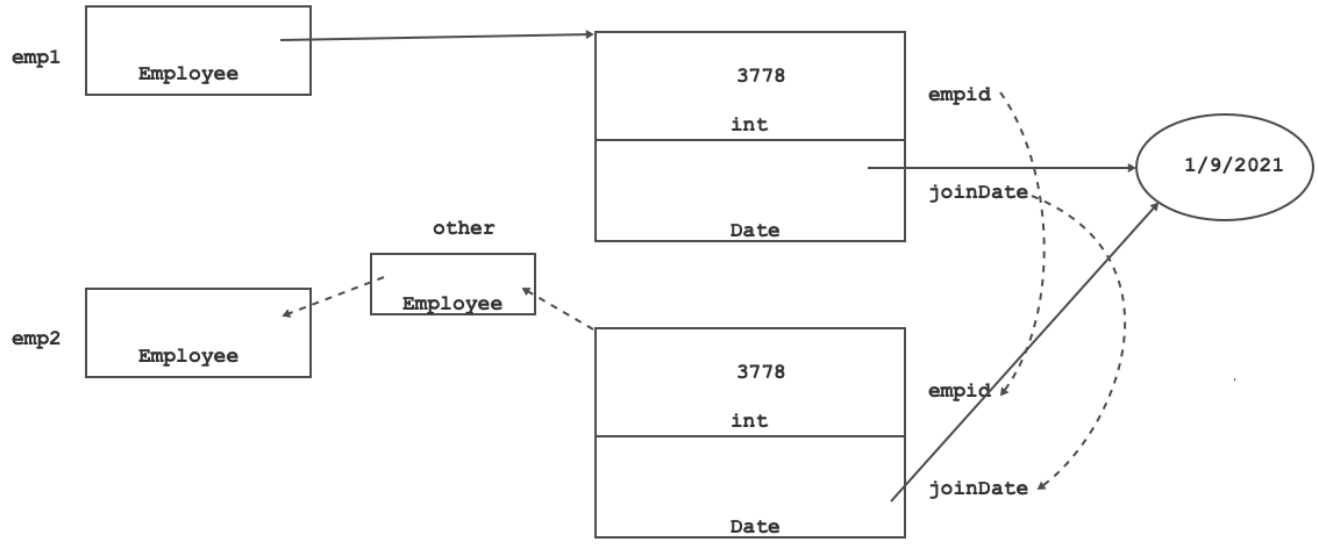
```
    } catch (CloneNotSupportedException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }
```



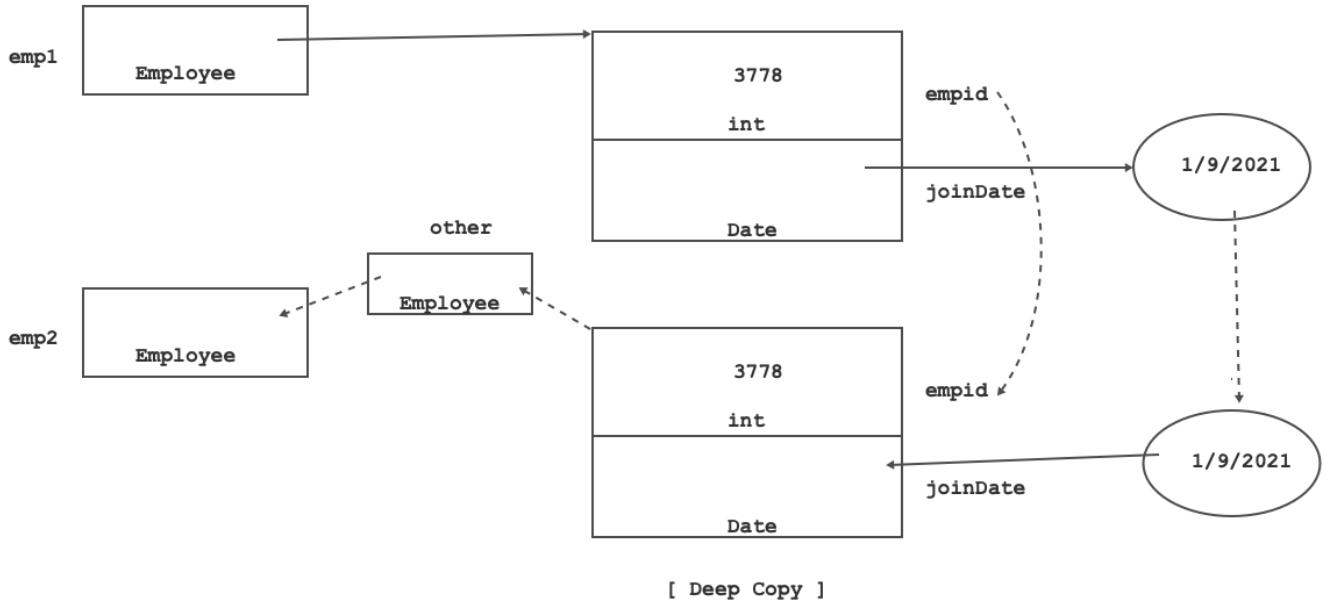
- Consider example of ArrayList:
- public class ArrayList extends AbstractList implements List, RandomAccess, Cloneable, Serializable

```
public static void main(String[] args) {
    ArrayList<Integer> list1 = new ArrayList<>();
    list1.add(10);
    list1.add(20);
    list1.add(30);


    ArrayList<Integer> list2 = (ArrayList<Integer>)list1.clone();
    list1.clear();
    System.out.println(list1);
    System.out.println(list2);
}
```

Shallow Copy



Deep Copy



Write a program to generate Linear singly linked list in java

- Operations:
  - public boolean empty()
  - public void addLast(int element)
  - public void removeFirst( )
  - public void printList( )