Java Language Features

Naming Conventions

- **Pascal case** is a naming convention used in programming where words are written without spaces, and each word's initial letter is capitalized.
- Consider below examples:
    - String
    - StringBuilder
    - StringIndexOutOfBoundsException
- In Java, pascal case naming convention is used to give name to the file name and type name(Interface,Class and Enum).
- **Camel case** is a naming convention used in programming where words are written without spaces, and each word's initial letter except first word is capitalized.
- Consider below examples
    - main
    - parseInt
    - showInputDialog
    - waitForPendingFinalizer
- In Java, camel case convention is used to give name to the:
    - Method parameter
    - Method local variable
    - Non final field
    - Method
- **Snake case** naming convention in Java refers to the process of naming variables, methods, and other identifiers by using lowercase letters and underscores (_) to separate words. In case of constants it will be in uppercase.
- Consider below examples:
    - MIN_PRIORITY
    - NORM_PRIORITY
    - MAX_PRIORITY
    - DAY_OF_MONTH
- In Java, snake case convention is used to give name to the final fields and enum constants.
- Package names are generally written in lowercase. Consider below examples:
    - java.lang
    - org.example
    - javax.servlet.http
- Reference: https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html

Getter and Setter Demonstration

- **Remember** class members in Java are by default considered as package level private.

- To achieve data security, we need to first hide the data. To do so, we declare fields in class private.

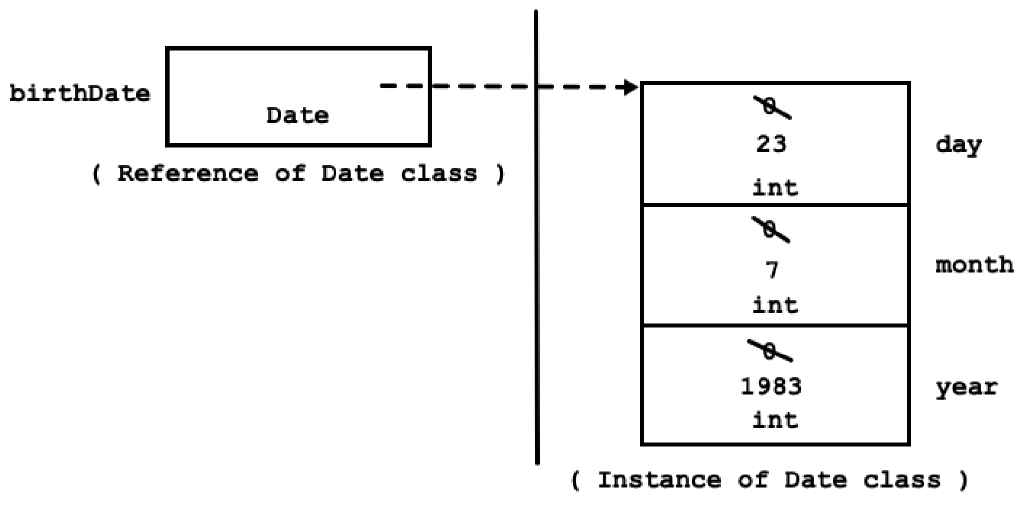
- But it fields are private then we can not access it outide the class directly. Consider below code:

```
class Date{
  private int day;
  private int month;
  private int year;
}

public class Program {
  public static void main(String[] args) {
    Date birthDate = new Date();
    birthDate.day = 23;         //Compiler Error: The field Date.day is not visible
    birthDate.month = 7;    //Compiler Error: The field Date.month is not visible
    birthDate.year = 1983;    //Compiler Error: The field Date.year is not visible
  }
}
```

- We can not access private members directly but we can access(get/set) it using method. Previously we have written acceptRecord/printRecord methods to access it. But its too much specific to console.

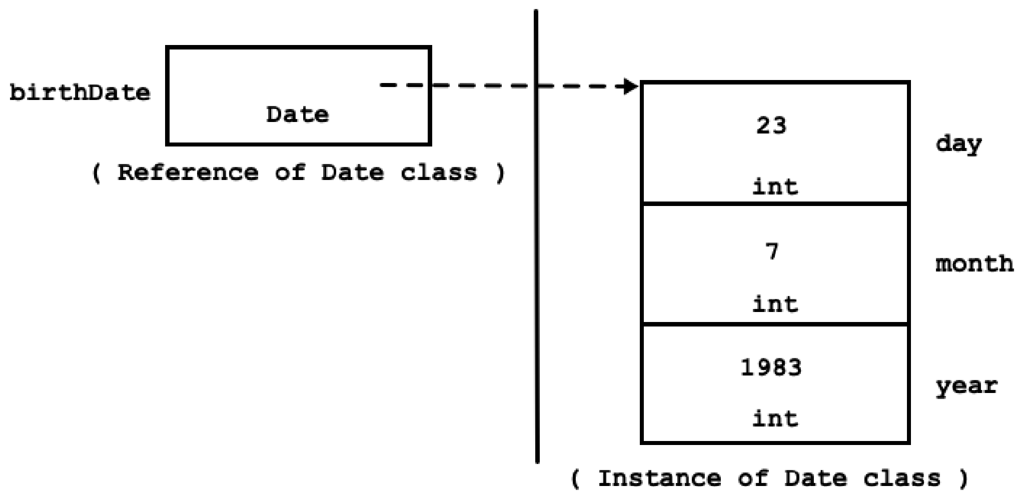
- Lets see, how to define methods to modify state of the birthDate instance.

```java
class Date{
    private int day;
    private int month;
    private int year;

    //Date this = birthDate
    public void setDay(int day) {
        this.day = day;
    }
    //Date this = birthDate
    public void setMonth(int month) {
        this.month = month;
    }
    //Date this = birthDate
    public void setYear(int year) {
        this.year = year;
    }
}

public class Program {
    public static void main(String[] args) {
        Date birthDate = new Date();
        birthDate.setDay(23);      //OK: setting day value for birthDate instance
        birthDate.setMonth(7);     //OK: setting month value for birthDate instance
        birthDate.setYear( 1983 ); //OK: setting Year   value for birthDate instance
    }
}
```



- Lets see, how to define methods to read state of the birthDate instance. Due to space issue in image, I have not considered constructor / setter method. But you can define it.

```java
class Date{
    private int day = 23;
    private int month = 7;
    private int year = 1983;

    //Date this = birthDate;
    public int getDay() {
        return this.day;
    }
    //Date this = birthDate;
    public int getMonth() {
        return this.month;
    }
    //Date this = birthDate;
    public int getYear() {
        return this.year;
    }
}

public class Program {
    public static void main(String[] args) {
        Date birthDate = new Date();
        int day = birthDate.getDay();     //OK: getting day value from birthDate instance
        int month = birthDate.getMonth();//OK: getting month value from birthDate instance
        int year = birthDate.getYear();      //OK: getting Year   value from birthDate instance
    }
}
```



How to get System Date using Java API?

- Using java.util.Date class

```java
import java.util.Date;
public class Program {
    public static void main(String[] args) {
        Date currentDate = new Date();

        int day = currentDate.getDate();
        int month = currentDate.getMonth() + 1; // Note: Month is 0-based
        int year = currentDate.getYear() + 1900; // Note: Year is represented as year - 1900

        System.out.println(day+" / "+ month + " / "+ year);
    }
}
```

- Using java.util.Calendar class

```java
import java.util.Calendar;
public class Program {
    public static void main(String[] args) {
        Calendar currentDate = Calendar.getInstance();

        int day = currentDate.get(Calendar.DAY_OF_MONTH);
        int month = currentDate.get(Calendar.MONTH) + 1; // Note: Month is 0-based
        int year = currentDate.get(Calendar.YEAR);

        System.out.println(day + " / " + month + " / " + year);
    }
}
```

- Using java.util.LocalDate

```
import java.time.LocalDate;
public class Program {
    public static void main(String[] args) {
        LocalDate currentDate = LocalDate.now();

        int day = currentDate.getDayOfMonth();
        int month = currentDate.getMonthValue();
        int year = currentDate.getYear();

        System.out.println(day + " / " + month + " / " + year);
    }
}
```

- Assignment: Now try to write code to get System Time using above API.
- Let us see, how to create Calendar instance using given day, month & year and how to add number of days in given Date.

```
import java.util.Calendar;
public class Program {
  public static void main(String[] args) {
    int day = 21;
    int month = 9; // Note: Month is 0-based (0 for January, 1 for February, ..., 11 for December)
    int year = 2021;

    Calendar calendar = Calendar.getInstance(); // Create a Calendar instance

    calendar.set(Calendar.YEAR, year);
    calendar.set(Calendar.MONTH, month - 1); // Adjust for 0-based month
    calendar.set(Calendar.DAY_OF_MONTH, day);
    System.out.println("Old date: " + day + " / " + month + " / " + year);//Old date: 21 / 9 / 2021

    calendar.add(Calendar.DAY_OF_MONTH, 15); // Add 15 days to the date

    // Get the updated date
    day = calendar.get(Calendar.DAY_OF_MONTH);
    month = calendar.get(Calendar.MONTH) + 1; // Adjust for 0-based month
    year = calendar.get(Calendar.YEAR);
    System.out.println("New date: " + day + " / " + month + " / " + year);//New date: 6 / 10 / 2021
  }
}
```

SimpleDateFormat

- SimpleDateFormat is a class declared in java.text package. It is used to format and parse date.

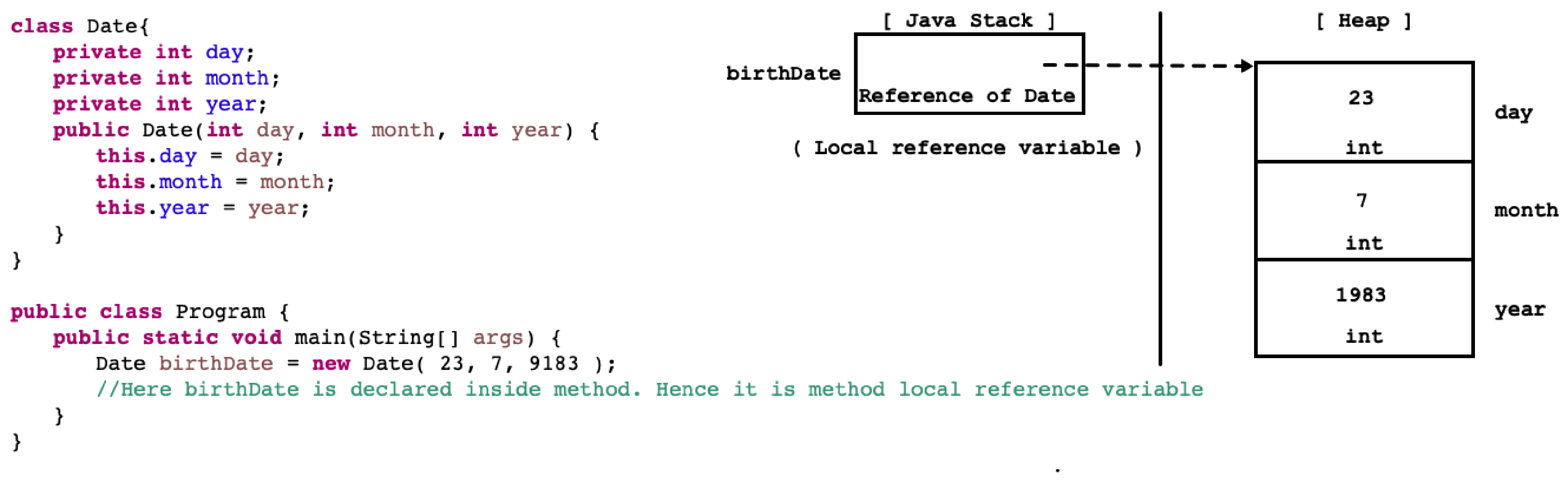
```
import java.text.SimpleDateFormat;
import java.util.Date;
public class DateFormatDemo {
    public static void main(String[] args) {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        Date currentDate = new Date();
        String formattedDate = sdf.format(currentDate);
        System.out.println("Formatted date: " + formattedDate)
    }
}
```

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class DateFormatDemo {
    public static void main(String[] args) {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String dateStr = "2024-03-05 14:30:00";
        try {
            Date parsedDate = sdf.parse(dateStr);
            System.out.println("Parsed date: " + parsedDate);
        } catch (ParseException e) {
            System.out.println("Error parsing date: " + e.getMessage());
        }
    }
}
```

- Below are common format symbols used in the SimpleDateFormat class:
    - **yyyy**: Represents the year in a four-digit format. Example: 2024.
    - **MM**: Represents the month in a two-digit format. January is 01 and December is 12.
    - **dd**: Represents the day of the month in a two-digit format.
    - **HH**: Represents the hour of the day (24-hour clock) in a two-digit format.
    - **hh**: Represents the hour of the day (12-hour clock) in a two-digit format.
    - **mm**: Represents the minute in a two-digit format.
    - **ss**: Represents the second in a two-digit format.
    - **SSS**: Represents the millisecond in a three-digit format.
    - **E**: Represents the day of the week in a short textual format (e.g., Mon, Tue).
    - **EEE**: Represents the day of the week in a short textual format (e.g., Mon, Tue).
    - **EEEE**: Represents the day of the week in a full textual format (e.g., Monday, Tuesday).
    - **a**: Represents the AM/PM marker in lowercase (am or pm).
    - **Z**: Represents the time zone in RFC 822 time zone format (+HHMM or -HHMM). Example: +0530.
    - **zzzz**: Represents the time zone in full time zone format. Example: Pacific Standard Time.
    - **zzz**: Represents the time zone in abbreviated time zone format. Example: PST.

Memory allocation for reference variable

- We can declare reference variable inside method. It is called as method local reference variable.
- Method local reference variable get space on Java Stack. Consider below diagram.



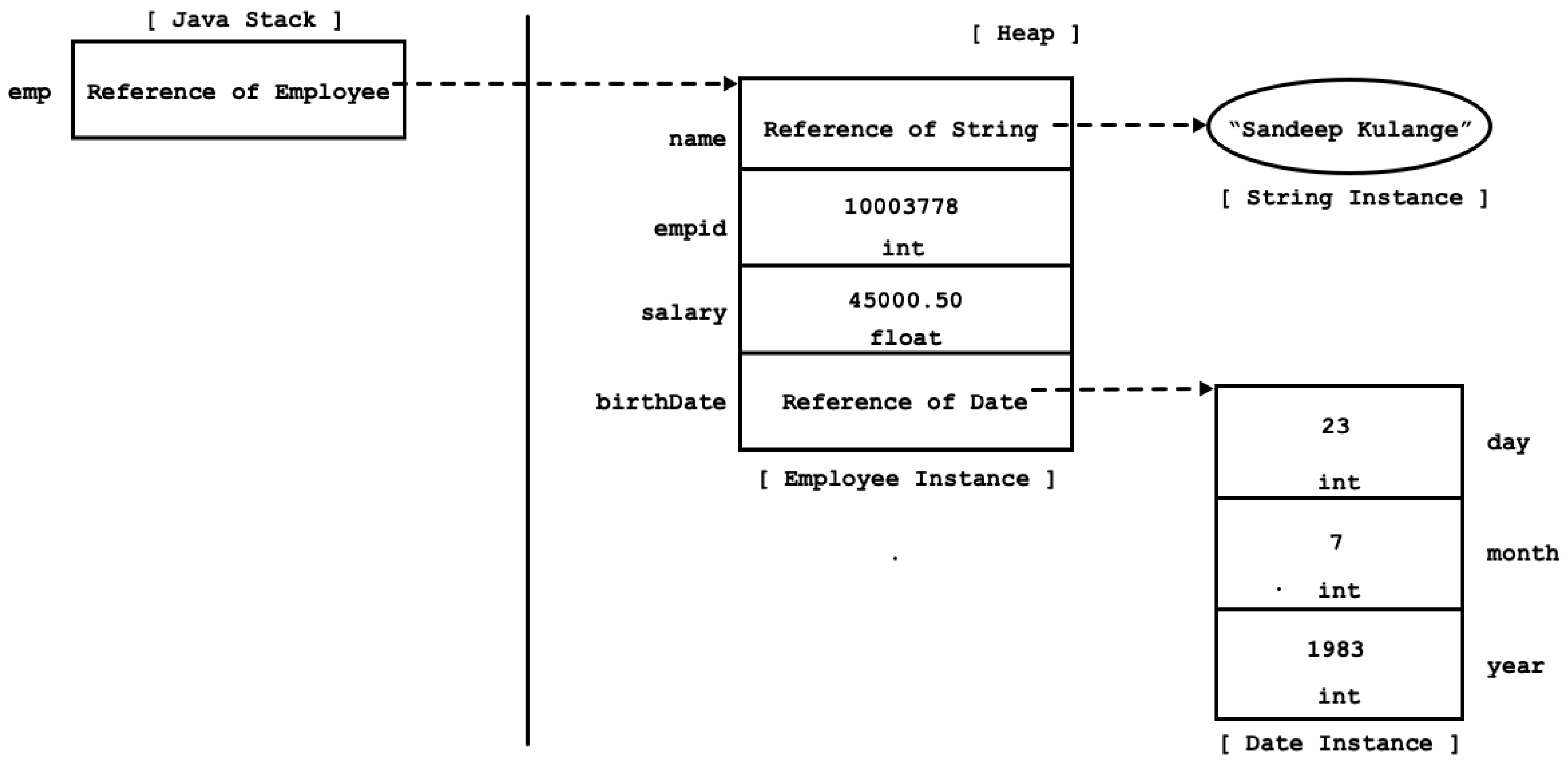
```
class Date{
    private int day;
    private int month;
    private int year;
    public Date(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }
}

public class Program {
    public static void main(String[] args) {
        Date birthDate = new Date( 23, 7, 9183 );
        //Here birthDate is declared inside method. Hence it is method local reference variable
    }
}
```

- We can declare reference variable as a field(static as well as non static) of the class. As we know, non static field get space inside instance, non static reference variable also get space inside instance on heap.
- Consider below code:

```
class Date{
  private int day;
  private int month;
  private int year;
  //TODO: define constructor, getter and setter methods
}
```

```
  class Employee{
    private String name;
    private int empid;
    private float salary;
    private Date birthDate;
    //TODO: define constructor, getter and setter methods
  }
```

```
class Program{
  public static void main( String[] args ){
    Employee emp = new Employee( );
    emp.setName( "Sandeep Kulange" );
    emp.setEmpid( 10003778 );
    emp.setSalary( 45000.50f );
    BirthDate birthDate = new Date( 23, 7, 1983 );
    emp.setBirthDate( birthDate );
    //or emp.setBirthDate( new Date( 23, 7, 1983 ) );
  }
}
```



- As shown in above image, emp is local variable hence it will get space on Java stack. name and birthDate are non static fields declared inside class. Hence it will get space in instance on Heap.
- If we declare , refeference variable as a static field inside class then it will get space on method area. We will discuss it in static section.

Final Variable

- Consider below code and recall concept reagrding garbage value:

```
  public static void main(String[] args) {
    int number;
    System.out.println( number );   //Error: The local variable number may not have been initialized
  }
```

```
  public static void main(String[] args) {
    int number = 100;   //OK
    number = number + 50;   //OK
    System.out.println( number );   //150
  }
```

- If we want to make any variable readonly then we should use final modifier in Java. Consider below code:

```
public class Program {
  private static Scanner sc = new Scanner(System.in);
  public static void main(String[] args) {
    final int num1 = 10;  //OK
    //num1 = num1 + 5;    //Not OK

    final int num2;   //OK
    num2 = 10;    //OK
    //num2 = num2 + 5;    //Not OK

    System.out.print("Enter number    :   ");
    final int num3 = sc.nextInt();    //OK
    //num3 = num3 + 5;    //Not OK

    final int num4;
    System.out.print("Enter number    :   ");
    num4 = sc.nextInt();  //OK
    //num4 = num4 + 5;    //Not OK
  }
}
```

- Observations from above demo are:
  - We can provide value to final local variable either during declaration or after declaration. But once we store value inside it we can not modify it. Hence it is readonly variable.
  - We can provide value to final variable either at compile time or runtime using scanner etc.
  - final means last value providing to the variable. Here after we can not modify it.
- If we dont want to modify value of the field inside any method of the class then we should declare such field final.
- Let us first understand syntax:

```
class Test{
  private final int number;//Error: The blank final field number may not have been initialized
}
```

```
class Test{
  private final int number = 10;//OK
}
```

```
class Test{
  private final int number = 10;//OK
  public Test( ) {    //Constructor
    this.number = 20; //Error: The final field Test.number cannot be assigned
  }
}
```

```
class Test{
  private final int number;
  public Test( ) {    //Constructor
    this.number = 20; //OK
  }
}
```

- **Conclusion** if field is final then we should intialize it using either field initializer or constructor but not both.
- Let us see, how to declare and use final field. Consider below code:

```
class Test{
  private final int number = 10;//OK
  public int getNumber() {
    return this.number;
  }
}
public class Program {
  public static void main(String[] args) {
        Test t = new Test( );
        System.out.println("Number    :   "+t.getNumber());
      }
    }
```

- **Note** we can declare reference final but we can not declare instance final.
- Consider below code:

```
final Comple c1 = new Complex( 10, 20 );  //OK

System.out.println("Real Number : "+c1.getReal( ) );  //10
System.out.println("Imag Number : "+c1.getImag( ) );  //20

//Modifing state of instance
c1.setReal( 50 ); //OK
c1.setImag( 60 ); //OK

System.out.println("Real Number : "+c1.getReal( ) );  //50
System.out.println("Imag Number : "+c1.getImag( ) );  //60
```

```
final Complex c1 = new Complex( 10, 20 ); //OK
c1 = new Complex( 50, 60 ); //Not OK
//Since c1 is final, it can't store reference of second instance.
```

- We can declare method and class too final but we will discuss it in inheritance.

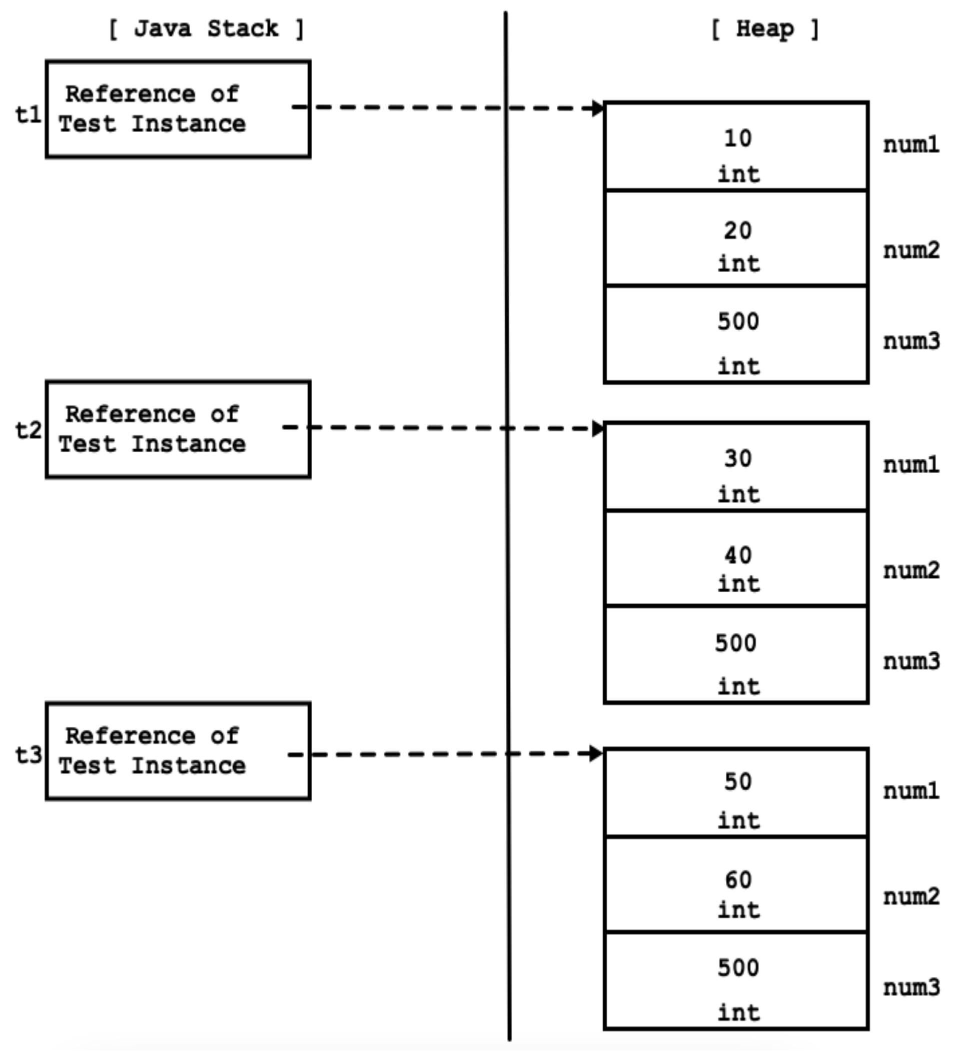
Static Fields

```
public static void main( ){
  static int num1 = 10; //Not OK
  int num2 = 20;  //OK: Non static method local variable
}
```

- In Java, we can not declare local variable static but we can declare field static.s
- Non static field declared inside class is called as instance variable. As we have discussed earlier, instance variable get space once per instance.
- Since instance variable get space once per instance:
  - We should initialize it inside constructor.
  - We should use this reference inside method of same class and object reference inside method of different class to access it.
  - We should define instance method to process its state.
- Below code demonstrate instance variable declaration and its initialization

```
class Test{
  private int num1;       //Instance variable
  private int num2;       //Instance variable
  private final int num3; //Instance variable

  public Test( ){ //Parameterless constructor
    this.num3 = 500;
  }

  public( int num1, int num2 ){ //Parameterized constructor
    this.num1 = num1;
    this.num2 = num2;
    this.num3 = 500;
  }
}
```

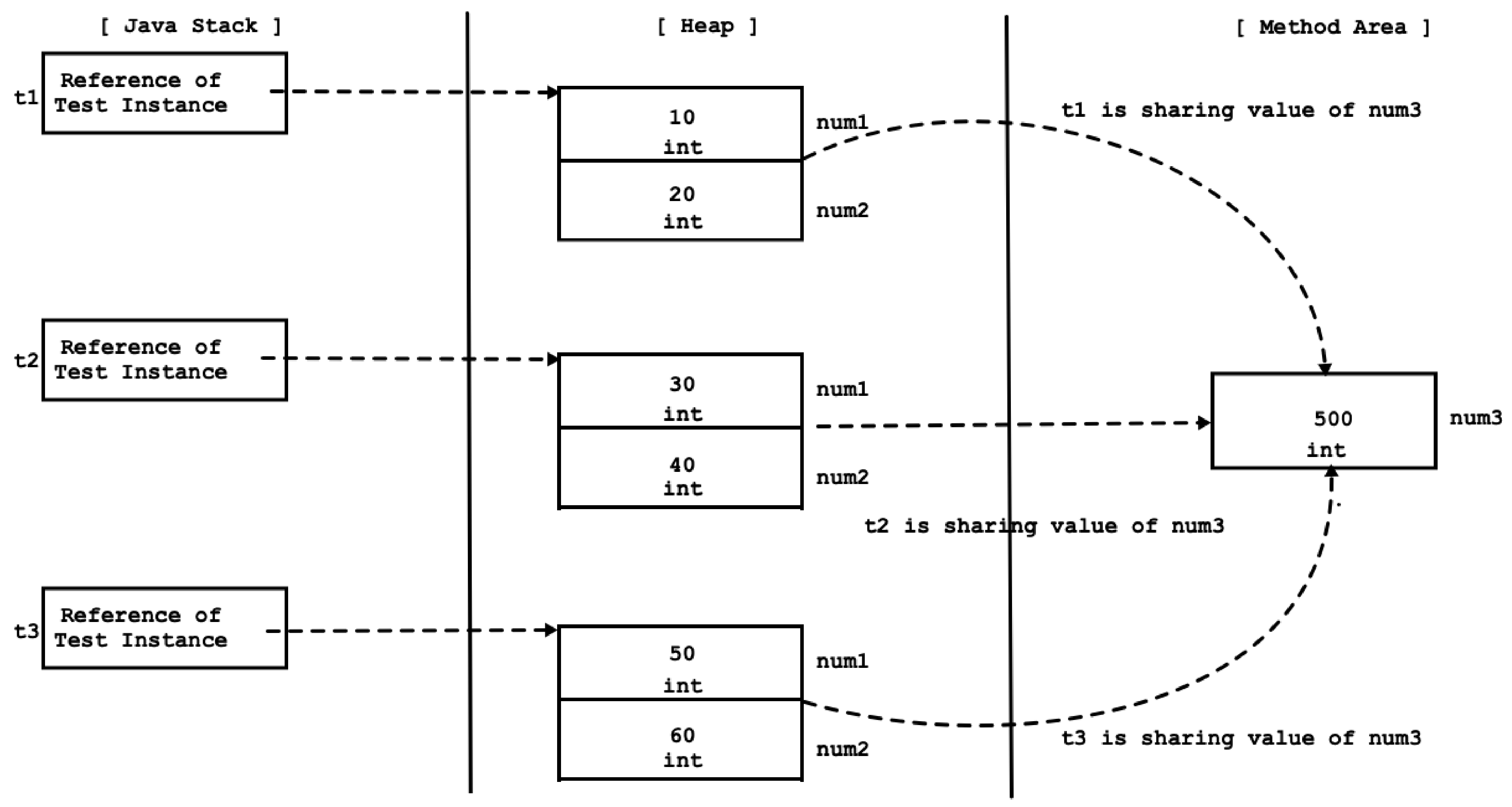
```
class Program{
  public static void main( String[] args ){
    Test t1 = new Test(10, 20);
    Test t2 = new Test(30, 40);
    Test t3 = new Test(50, 60);
    }
  }
```



- If we want to share value of the field inside all the instances of same class then we should declare such field static.
- static is modifier in Java(In C/C++ it is storage class).
- Since static fields are designed to share among all the instances of same class, it doesn't get space inside instance. Hence size of instance depends on size of all the non static fields declared inside class.
- If we declare field static then such field is called as class level variable. In Java, class level variables are designed to access using class name and dot operator.
- Since static field i.e. class level variable do not get space inside instance, we should not initialite it inside constructor body.
    - For static initialization of class level variable, we should use static field initializer.
    - For dynamic initialization of class level variable, we should use static initializer block.
- Consider below code:

```
   class Test{
     private int num1;       //Instance variable
     private int num2;       //Instance variable
     private static final int num3 = 500; //Instance variable

     public Test( ){ //Parameterless constructor
       //TODO
     }

     public( int num1, int num2 ){ //Parameterized constructor
       this.num1 = num1;
       this.num2 = num2;
     }
   }
```

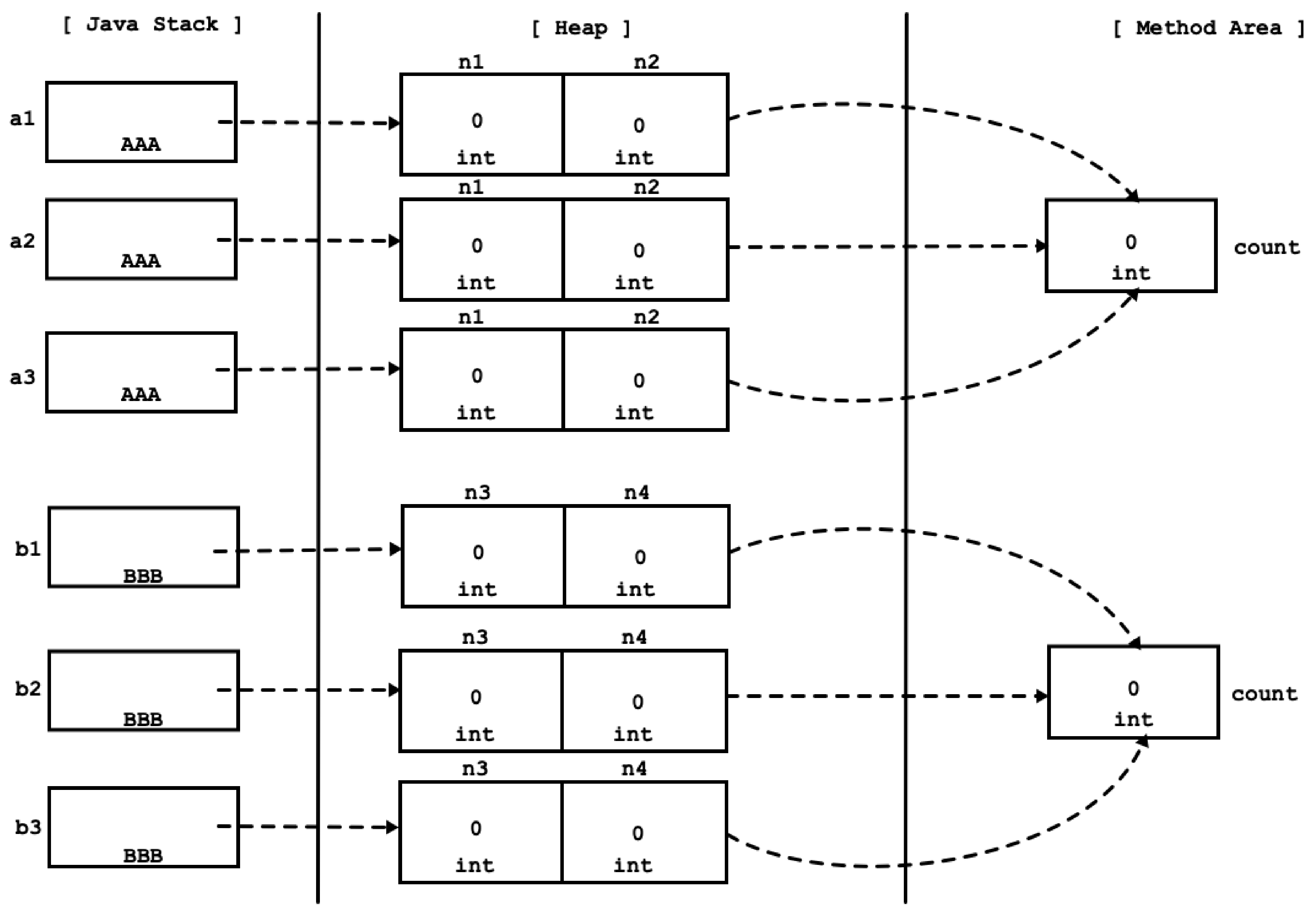
```
  class Program{
    public static void main( String[] args ){
      Test t1 = new Test(10, 20);
      Test t2 = new Test(30, 40);
      Test t3 = new Test(50, 60);
    }
  }
```


[Java Stack]
[Heap]
[Method Area]
t1
Reference of Test Instance
t2
Reference of Test Instance
t3
Reference of Test Instance
10 int
num1
20 int
num2
30 int
num1
40 int
num2
50 int
num1
60 int
num2
500 int
num3
t1 is sharing value of num3
t2 is sharing value of num3
t3 is sharing value of num3

- Static field / class level variable get space once per class on method area. Consider below code:

```java
class AAA{
  private int n1;
  private int n2;
  private static int count;
}
class TestAAA{
  public static void main( String[] args ){
    AAA a1 = new AAA( );
    AAA a2 = new AAA( );
    AAA a3 = new AAA( );
  }
}
```

```java
class BBB{
  private int n3;
  private int n4;
  private static int count;
}
class TestBBB{
  public static void main( String[] args ){
    BBB b1 = new BBB( );
    BBB b2 = new BBB( );
    BBB b3 = new BBB( );
  }
}
```



- To initialize non static fields, we should use constructor whereas to initialize static fields we should use static initializer / initialization block. Consider below code:
- Static initialization block can initialize only static fields of the class.

```
class Test{
  private int num1;
  private int num2;
  private static int num3;

  static{ //Static initialization block
    Test.num3 = 30;
  }

  public Test( ){ //Constructor
    this.num1 = 10;
    this.num2 = 20;
  }
}
```

- JVM executes, static initialization block during class loading.
- We can initialize all the static fields inside single static initializer block or we can define separate initialization block for each field. Consider below code:
    - Example 1:

```
class Test{
  private static int num1;
  private static int num2;
  private static int num3;
  static{
    Test.num1 = 10;
    Test.num2 = 20;
    Test.num3 = 30;
  }
}
```

- Example 2:

```
class Test{
  private static int num1;
  private static int num2;
  private static int num3;
  static{
    Test.num1 = 10;
  }
  static{
    Test.num2 = 20;
  }
  static{
    Test.num3 = 30;
  }
}
```

- When we define multiple static initialization block inside class then JVM executes it sequentially.
- Let us revise syntax:

```
class Test{
  private int num1 = 10;        //Instance field initializer
  private static int num2 = 20; //Static field initializer

  private int num3;
  private static int num4;

  {//Instance Initializer block
    this.num3 = 30;
  }
  static{//Static initialization block
    Test.num4 = 40;
  }
  int num5;
  public Test( ){ //Constructor
    this.num5 = 50;
  }
}
```

- **Remember** inside method, if name of local variable and name of non static field is same then to avoid conflict, we should use this before non static field. Similarly, if name of local variable and name of static field is same then to avoid conflict, we should use class name before static field.
- Reference: https://docs.oracle.com/javase/tutorial/java/javaOO/initial.html

Static Methods

- Non static method defined inside class is also called as instance method.
- Instance variable always get space inside instance and to process state of instance variable, we should call instance method on instance.
- We can not call instance method on class. Consider below code:

```
class Test{
  private int number; //Instance Variable
  public int getNumber( ){  //Instance method
    return this.number;
  }
  public void setNumber( int number ){//Instance method
    this.number = number;
  }
}
```

```
class Program{
  public static void main( String[] args ){
    //Test.setNumber( 100 );  //Not OK: Can not call instance method on class name
    //int number = Test.getNumber(  );  //Not OK: Can not call instance method on class name

    Test t = new Test( ); //OK: Instance
    t.setNumber( 100 ); //OK: Instance method is called on instance
    System.out.println("Number  : "+t.getNumber()); //OK: Instance method is called on instance
    }
  }
```

- Since instance, method is designed to call on instance, it gets this reference.
- In general, to process state of non static fields, we should define non static method inside class and to process state of static fields, we should define static method inside class.
- Static method defined inside class is called as class level method. It is designed to call on class name.
- Consider below code:

```
class Test{
  private static int number; //Class level Variable
  public static int getNumber( ){  //class level method
    return Test.number;
  }
  public static void setNumber( int number ){//Class level method
    Test.number = number;
  }
}
```

```
class Program{
  public static void main( String[] args ){
    Test.setNumber( 100 );  //OK: We can call class level method on class name
    System.out.println("Number  : "+Test.getNumber());  //OK: We can call class level method on class name

    Test t = new Test( ); //OK: Instance
    t.setNumber( 100 ); //OK: Not Recommended
    System.out.println("Number  : "+t.getNumber()); //OK: Not Recommended
  }
}
```

> **Remember** even though static methods are designed to call on class name, we can call it on instance too. But it is not recommended.

- **Why static method do not get this reference?**
  - If we call, non static method on instance then non static method get this reference.
  - Static method is designed to call on class name.
  - Since static method is not designed to call on instance, it doesn't get this reference.

  > If we try to call static method on class name or instance then it doesn't get this reference.

- Since static method do not get this reference, we can not access non static members inside static method directly. In this case, we must use instance of the class. In other words, static method can access only static members of the class directly.
- Consider below example:

```
class Test{
  private int num1;
  private int num2;
  private static int num3;

  public void showRecord( ) {
    System.out.println("Num1  :   "+this.num1);
    System.out.println("Num2  :   "+this.num2);
    System.out.println("Num3  :   "+Test.num3);
  }
  public static void displayRecord( ) {
    Test t = new Test();
    System.out.println("Num1  :   "+t.num1);
    System.out.println("Num2  :   "+t.num2);
    System.out.println("Num3  :   "+Test.num3);
  }
}
public class Program {
    public static void main(String[] args) {
        Test t = new Test();
        t.showRecord();

        Test.displayRecord();
        }
    }
```

- **Remember** inside method, if we require to use this reference to access the members only then we should declare method non static otherwise declare method static.
- **Note :**
    - We cannot access any instance member (variable or method) using class name. We must use object reference.
    - We can use object reference to access class level member(variable or method) but it doesn't belongs to instance. Hence to access class level member inside method of same class / different class, we should use class name.
- Reference: https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html

Nested class and Local class

- First, let us discuss **Nested Class**.
    - In Java, we can define class inside another class / scope of another class. It is called as nested class.
    - Consider below code:

    ```
    public class LinkedList{  //Top level class
      //TODO
      private class LinkedListNode{  //Nested class
        //TODO
      }
    }
    ```

    - Java compiler generates .class file for nested class as well as top level class:
        - .class file for the top level class will be `LinkedList.class`.
        - .class file for the nested class will be `LinkedList$LinkedListNode.class`.
    - Since nested class is considered as member of a class, we can use any(private/package private/protected) access modifier on it. But access modifier of top level class can be either package private or public only.
    - Nested classes offer several advantages:
        - **Logical Grouping:**
            - If a class is only used within another class, nesting it inside that class logically organizes related functionality in one place, simplifying the package structure.
        - **Increased encapsulation:**
            - When one class (let's call it A) needs to provide access to its private members to another class (let's call it B), instead of making those members public or creating complex accessors, nesting class B within class A allows B to access A's private members while keeping them hidden from the outside world.
        - **Improved readability and maintainability:**
            - Placing small, related classes within the top-level class where they are used makes the code more readable and easier to maintain.
- **Types of Nested class:**
    - Non static nested class (also called as inner class)
    - Static nested class

    ```
    public class LinkedList{  //Top level class

      private static class LinkedListNode{  //Static nested class
        private int data;
        private LinkedListNode;
        //TODO: constructor
      }

      private LinkedListNode head;
      private LinkedListNode tail;
      //TODO: constructor and LinkedList methods

      private class LinkedListIterator{ //Non static nested class
        private LinkedListNode trav = head;
        //TODO: constructor and other methods
      }
    }
    ```

- **Inner Class**
  - Non static field declared inside class is called as instance variable and it gets space once per instancce. It means that instance variable is associated with instance.
  - Non static method defined inside class is called as instance method and it is designed to call on instance. It means that instance method is associated with instance and it has access to all the instance members as well as class level members.
  - Similarly, non static nested class is also called as inner class and it is associated with instance of enclosing class.
  - Since inner class is associated with instance of enclosing class, it has access to fields and methods of enclosing class.
  - Because an inner class is associated with an instance, it cannot define any static members itself.
  - **Note:** for simplicity consider non static nested class as non static/instance method of the class.
    - Inside non static method we can access static as well as non static members of the class directly.
    - To call instance method, instance is required.
  - **Instantiation:**

```
class Outer{
  class Inner{
  }
}
```

    - Create instance of top level class.

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

    - Create instance of non static nested class:
      - Method 1:

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

      - Method 2:

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

  - **Memory Representation**
    - Each instance of the outer class holds a reference to its inner class instance if it creates one. This reference is stored in memory along with other instance variables of the outer class.
    - When an instance of the inner class is created, it holds an implicit reference to the outer class instance that created it.
    - Since the inner class instance holds a reference to the outer class instance, it can directly access the instance variables and methods of the outer class.
    - If the outer class instance is garbage collected, the inner class instance will also be eligible for garbage collection.
  - **Example 1**

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

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

  public void printRecord( ) {
    System.out.println("Num1  :   "+num1);    //OK
    System.out.println("Num2  :   "+num2);    //OK
    //System.out.println("Num3    :   "+num3);    //Not OK
    //System.out.println("Num4    :   "+num4);    //Not OK

    Inner in = new Inner();
    System.out.println("Num3  :   "+in.num3); //OK
    System.out.println("Num4  :   "+Inner.num4);  //OK
  }
}
```

```
public class Program {
  public static void main(String[] args) {
    Outer out = new Outer();
    out.printRecord();
  }
}
```

- **Example 2**

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

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

    public void printRecord( ) {
      System.out.println("Num1    :   "+num1);    //OK
      System.out.println("Num2    :   "+num2);    //OK
      System.out.println("Num3    :   "+num3);    //OK
      System.out.println("Num4    :   "+num4);    //OK
    }
  }
}
```

```
public class Program {
  public static void main(String[] args) {
    Outer.Inner in = new Outer().new Inner();
    in.printRecord();
  }
}
```

- **Example 3**

```
class Outer{          //Top level class
  private int num1 = 10;          //OK

  public class Inner{ //Non static nested class
    private int num1 = 20;            //OK

    public void printRecord( ) {
      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.printRecord();
    }
}
```

- **Static Nested Class**

- Static field declared inside class is called as class level variable and it gets space once per class. It means that class level variable is associated with class.
- Static method defined inside class is called as class level method and it is designed to call on class. It means that class level method is associated with class and it has access to all the class level members.
- Similarly, static nested class is associated with enclosing class.
- Since static nested class is associated with Outer class, it has access to static fields and static methods of enclosing class. A static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.
- **Note:** for simplicity, consider static nested class as static method of the class.
  - Inside static method we can access only static members of the class directly. Using instance, we can access non static members.
  - To call static method, only class name is required.
- **Instantiation:**

```java
class Outer{
  static class Inner{
  }
}
```

  - Create instance of top level class.

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

  - Create instance of static nested class:
    - Method 1:

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

    - Method 2:

```java
import org.example.Outer.Inner;
Inner in = new Inner();
```

- **Memory Representation:**
  - Since a nested class is declared as static, it does not have an implicit reference to an instance of the outer class. This means that instances of the static nested class can exist independently of any instances of the outer class.
- **Example 1**

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

  public static class Inner{  //Non static nested class
    private int num3 = 30;             //OK
    private static int num4 = 40; //OK
  }

  public void printRecord( ) {
    System.out.println("Num1  :   "+num1);    //OK
    System.out.println("Num2  :   "+num2);    //OK
    //System.out.println("Num3    :   "+num3);    //Not OK
    //System.out.println("Num4    :   "+num4);    //Not OK

    Inner in = new Inner();
    System.out.println("Num3  :   "+in.num3); //OK
    System.out.println("Num4  :   "+Inner.num4);  //OK
  }
}
```

```
public class Program {
  public static void main(String[] args) {
    Outer out = new Outer();
    out.printRecord();
  }
}
```

- **Example 2**

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

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

        public void printRecord() {
          // System.out.println("Num1 : "+num1); //Not OK
          System.out.println("Num1    :   " + new Outer().num1); // Not OK
          System.out.println("Num2    :   " + num2); // OK
          System.out.println("Num3    :   " + num3); // OK
          System.out.println("Num4    :   " + num4); // OK
        }
    }
}
```

```
public class Program {
  public static void main(String[] args) {
    Outer.Inner in = new Outer.Inner();
    in.printRecord();
  }
}
```

- Reference: https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html
- In Java, we can define class inside block/method, It is called as **local class**.
- Types of local class:
  - Method local inner class
  - Method local anonymous inner class
- **Method local inner class**
  - In Java, we can not use static modifier inside method. Hence local class / method local class is also called as method local inner class.
  - **Remember,** we can not create reference or instance of method local inner class outside the method.

```
public class Program {
  //private Inner in; //Not OK
  public static void main(String[] args) {

    class Inner{
      private int num1 = 10;
      private final static int num2 = 20;

      public void printRecord( ) {
        System.out.println("Num1  :   "+this.num1);
        System.out.println("Num2  :   "+Inner.num2);
      }
    }//End of Local class

    Inner in = new Inner();
    in.printRecord();
  }
}
```

  - .class file for class Program will be `Program.class`
  - .class file for class Inner will be `Program$1Inner.class`

- **Method local Anonymous inner class**
    - In Java, we can define class without name. It is called as class.
    - To define anonymous class we must use new operator.
    - We can define anonymous class inside block or method only. Hence it is also called as method local anonymous inner class.
    - Consider Example:

    ```
    public class Program{
      public static void main( String[] args ){
        Object o1;  //o1 is reference of java.lang.Object class
        o1 = new Object( ); //instance of java.lang.Object class

        Object o2 = new Object( );  //Initialization of o2 reference with instance.

        Object o3 = new Object(){ };  //Anonymous Inner class which is sub class of java.lang.Object class

        Object o4 = new Object( ){
          private String name = "Sandeep Kulange";
          public String toString( ){
            return "Hello,"+this.name;
          }
        };
        String str = o4.toString( );
        System.out.println( str );
      }
    }
    ```

        - In above code, we have defined 2 method local anonymous inner classes. Their .class files will be Program$1.class and Program$2.class respectively.
    - If we want one-time inline implementation then we should define anonymous inner class. Consider below Code:

    ```
    public static void main( String[] args ){
      Thread th = new Thread( new Runnable(){
        @Override
        public void run( ){
          System.out.println("Hello from run()")
        }
      });
      th.start( );
    }
    ```

        - This way, passing method as a argument to the another method is called as a **behavior parameterization**.
        - Behavior parameterization helps developer to decide and define behavior at runtime.

Introduction of Design Patterns

- A design pattern is a general, reusable solution to a commonly occurring problem.
- Design patterns provide a structured approach to design and to help address issues like scalability, maintainability, and flexibility in software systems.
- For your information, below are the most commonly used design patterns in Object Oriented Programming.
    - **Creational Patterns:**
        - Singleton Pattern
        - Factory Method Pattern
        - Abstract Factory Pattern
        - Builder Pattern
        - Prototype Pattern
    - **Structural Patterns:**
        - Adapter Pattern
        - Bridge Pattern
        - Composite Pattern
        - Decorator Pattern
        - Facade Pattern
        - Flyweight Pattern
        - Proxy Pattern
    - **Behavioral Patterns:**
        - Chain of Responsibility Pattern
        - Command Pattern
        - Interpreter Pattern

    - Iterator Pattern
    - Mediator Pattern
    - Memento Pattern
    - Observer Pattern
    - State Pattern
    - Strategy Pattern
    - Template Method Pattern
    - Visitor Pattern
- Reference: Gangs of Four Design Patterns
    - The book **"Design Patterns: Elements of Reusable Object-Oriented Software"**.

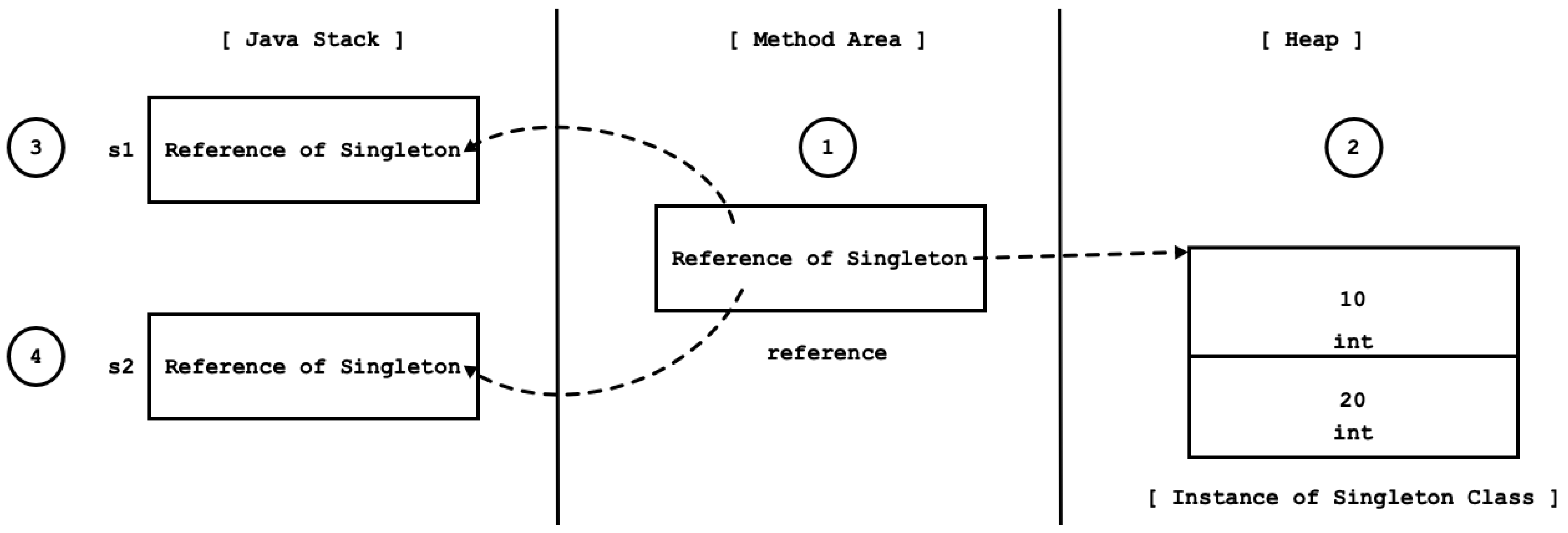
What is Singleton class?

- A class from which, we can create only once instance is called as singleton class.
- Implementation 1:

```java
class Singleton{
  private static Singleton reference;
  static{
    reference = new Singleton();
  }
  public static Singleton getReference( ){
    return reference;
  }
}

public class Program{
  public static void main( String[] args ){
    Singleton s1 = Singleton.getReference( );
    Singleton s2 = Singleton.getReference( );
  }
}
```

- Implementation 2:

```java
class Singleton{
  private int num1;
  private int num2;
  private Singleton( ){
    this.num1 = 10;
    this.num2 = 20;
  }
  private static Singleton reference;
  public static Singleton getReference( ){
    if( refeference == null ){
      reference = new Singleton( );
    }
    return reference;
  }
}

public class Program{
  public static void main( String[] args ){
    Singleton s1 = Singleton.getReference( );
    Singleton s2 = Singleton.getReference( );
  }
}
```



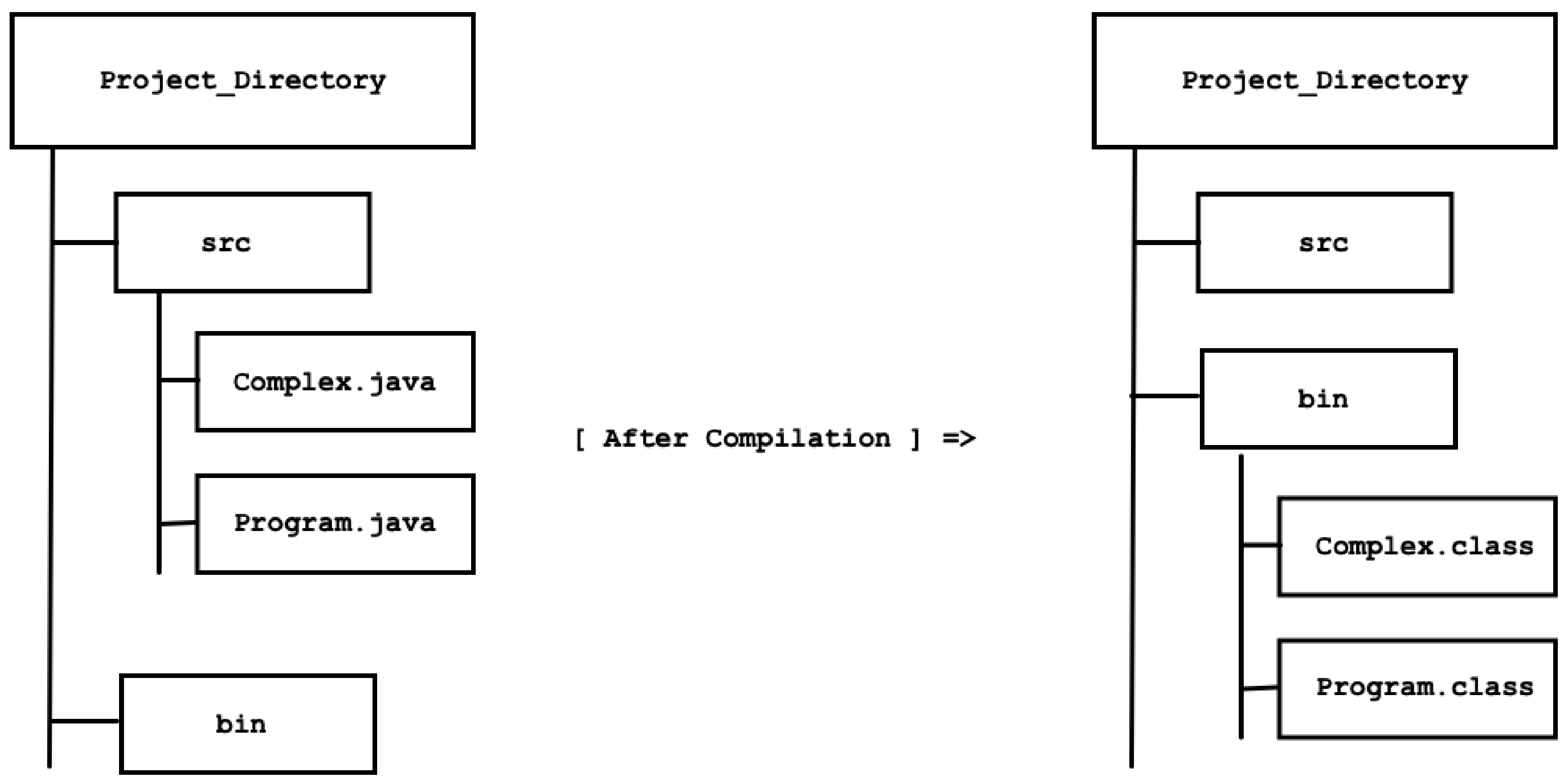
- Implementation 3:

```
class Singleton{
  private Singleton( ) {
    //TODO
  }
  private static class SingletonUtil{
    private static final Singleton reference = new Singleton();
  }
  public static Singleton getReferece( ) {
    return SingletonUtil.reference;
  }
}
public class Program {
  public static void main(String[] args) {
    Singleton s1 = Singleton.getReferece();
    Singleton s2 = Singleton.getReferece();
  }
}
```

Package

- Reference: https://docs.oracle.com/javase/tutorial/java/package/index.html
- **What is a package?**
  - Package is a Java language feature which is used,
    - To avoid name clashing, name collision, or name ambiguity and
    - To group functionally equivalent or functionally related types together.
- **Package members:**
  - Sub Package
  - Interface
  - Class
  - Enum
  - Exception
  - Error
  - Annotation Type
- **Terminology:**
  - Consider example of java.lang.Object:
    - java is called as main package
    - lang is called as sub package
    - Object is called as type name. Type can be interface/class/enum/error/exception/annotation.
- **What is path and classpath?**
  - In the context of Java, Path is OS platform environment variable which is used to locate.
    - In Windows: set path="C:\Program Files\Java\jdk1.8.0_XXX\bin";
    - In Linux or Mac OS: export PATH=/usr/bin
  - Classpath is Java platforms environment variable which is used to locate .class/.jar files.
    - In Windows: set classpath=.\bin;
    - In Linux or Mac OS: export CLASSPATH=./bin;

- **Example 1**



  - Consider class Complex(Example_1/src/Complex.java) without package:

```
class Complex{
  private int real = 10;
  private int imag = 20;
  public void printRecord( ){
    System.out.println(this.real+","+this.imag);
  }
}
```

  - Consider class Program(Example_1/src/Program.java) without package:

```
class Program{
  public static void main( String[] args ){
    Complex c1 = new Complex( );
    c1.printRecord( );
  }
}
```

  - How to compile and execute Java Application in Windows?

```
//Make Sure JAVA_HOME is set
javac -d .\bin .\src\Complex.java //Output: Complex.class
set classpath=.\bin;
javac -d .\bin .\src\Program.java //Output: Program.class
java Program  //10,20
```

  - How to compile and execute Java Application in Linux/Mac OS?

```
//Make Sure JAVA_HOME is set
javac -d ./bin ./src/Complex.java //Output: Complex.class
export CLASSPATH=./bin;
javac -d ./bin ./src\Program.java //Output: Program.class
java Program  //10,20
```

- package is a keyword in java which is used to define type inside package. Let us see how to define type in package:

```
package p1; //OK
class Complex{
  //TODO
}
```

- If we define any class/type inside package then it is called as packaged class/packaged type.
- In above code, class Complex is packaged class.
- Any class can be member of single package only. It is not possible to define partial class in one package and partial class in another.
- Package declaration statement must be first statement inside .java file. Hence below code examples are invalid:

```
package p1; //OK
package p2; //NOT OK
class Complex{
  //TODO
}
```

```
class Complex{
  //TODO
}
package p1; //NOT OK
```

- Package name is physically mapped to the folder/directory. It means, if we compile packaged class then compiler create directory for package which contains .class file.
- Consider example of unpackaged class:

```
//Program.java
class Complex{  //Unpackaged class
  //TODO
}
```

```
javac -d .\bin .\src\Complex.java //Output: ( bin\Complex.class)
```

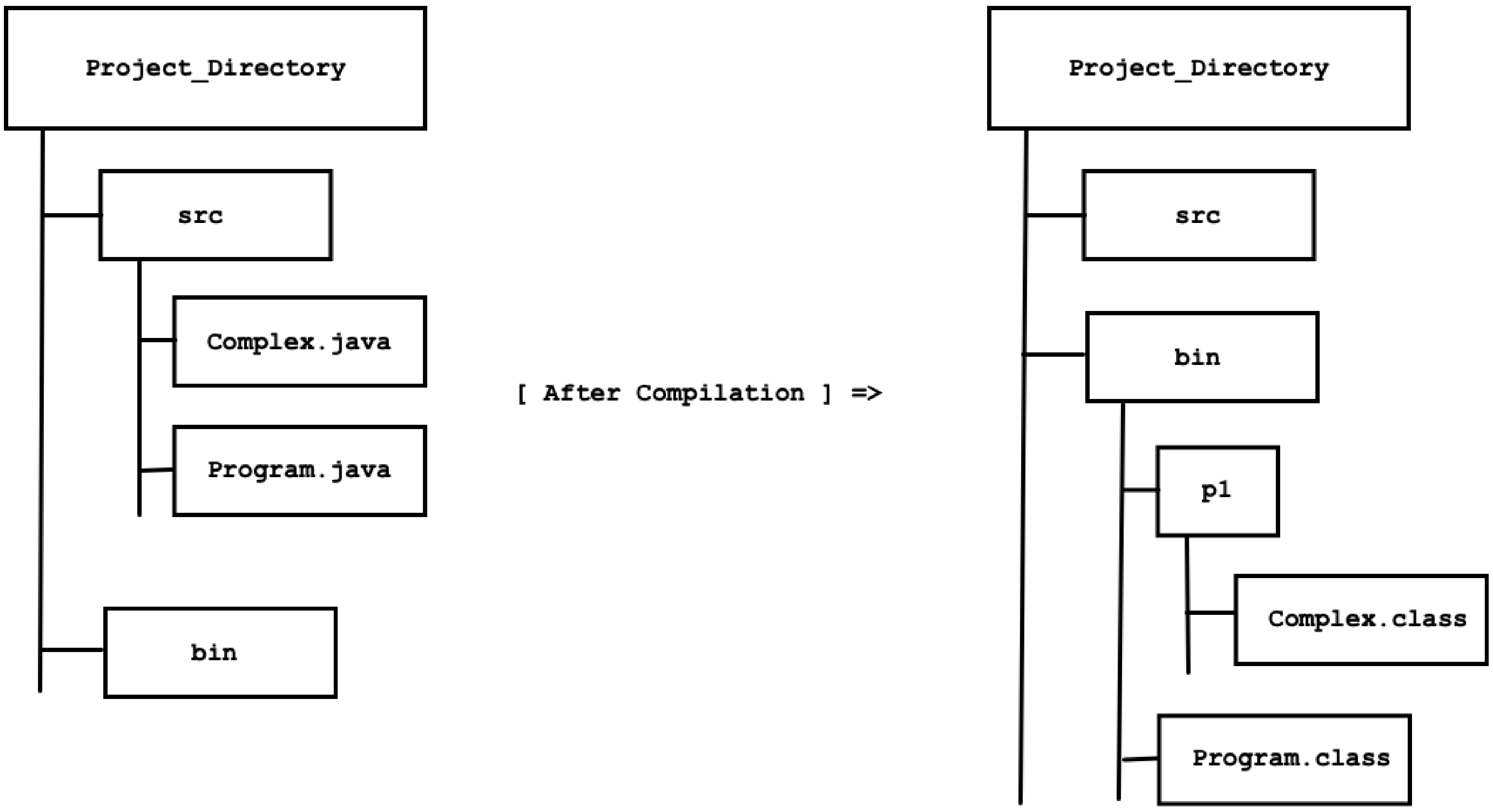
- Consider example of packaged class:

```
//Program.java
package p1;
class Complex{  //packaged class
  //TODO
}
```

```
javac -d .\bin .\src\Complex.java //Output: ( bin\p1\Complex.class)
```

  - bin directory will contain p1 and p1 will contain Complex.class.

- **Example 2**



  - Consider class TComplex(Example_2/src/Complex.java) with package. **Note,** here class name and file name is different.

```
package p1;
class TComplex{  //Packaged class
  private int real = 10;
  private int imag = 20;
  public void printRecord( ){
    System.out.println(this.real+","+this.imag);
  }
}
```

  - Consider class Program(Example_2/src/Program.java) without package:

```
class Program{  //Unpackaged class
  public static void main( String[] args ){
    TComplex c1 = new TComplex( );
    c1.printRecord( );
  }
}
```

  - How to compile and execute Java Application in Windows?

```
//Make Sure JAVA_HOME is set
javac -d .\bin .\src\Complex.java //Output: p1/TComplex.class
set classpath=.\bin;
javac -d .\bin .\src\Program.java //Output: Compiler Error
```

- If we want to use any packaged class then either we should use F.Q. class name or we should use import statement.
- Points to Remember:
  - import is keyword in Java
  - We can not define import statement before package declaration statement.
  - There can be one/muliple import statements after package declaration statement.
  - We can use import statement in packaged as well as unpackaged classes.
- Let us redefine class Program:
  - Option 1

```
class Program{  //Unpackaged class
  public static void main( String[] args ){
    p1.TComplex c1 = new p1.TComplex( );
    c1.printRecord( );
  }
}
```

- Option 2

```
import p1.TComplex;
class Program{  //Unpackaged class
  public static void main( String[] args ){
    TComplex c1 = new TComplex( );
    c1.printRecord( );
  }
}
```

- Now let us recompile the code:

```
//Make Sure JAVA_HOME is set
javac -d .\bin .\src\Complex.java //Output: p1/TComplex.class
set classpath=.\bin;
javac -d .\bin .\src\Program.java //Output: Compiler Error
```

```
error: TComplex is not public in p1; cannot be accessed from outside package
```

- Consider class TComplex again:

```
package p1;
??? class TComplex{  //Packaged class
  private int real = 10;
  private int imag = 20;
  public void printRecord( ){
    System.out.println(this.real+","+this.imag);
  }
}
```

  - By default access modifier of any class/type is package private(also called as default).
  - If access modifier is package private then we can not use class/type outside package.
  - If we want to allow any type to be used in same package as well as different package then its access modifier must be public.
  - **Note:** we can not declare class/type as private or protected. It means, access modifier of a type can be either package level private or public only.

```
package p1;
private class TComplex{  //Not OK
  //TODO
}
```

```
package p1;
protected class TComplex{  //Not OK
  //TODO
}
```

```
package p1;
class TComplex{  //OK
  //TODO
}
```

```
package p1;
public class TComplex{  //OK
```

```
      //TODO
    }
```

- According to Java Language Specification, name of the public class/type and name of the .java file must be same. Hence we can not define muliple public classes in single .java file.

- Now let us rewrite the code and compile it.
  - Consider class Complex(Example_2/src/Complex.java) with package.

```
package p1;
public class Complex{  //Now OK
  private int real = 10;
  private int imag = 20;
  public void printRecord( ){
    System.out.println(this.real+","+this.imag);
  }
}
```

  - Consider class Program(Example_2/src/Program.java) without package:

```
import p1.Complex;
class Program{  //Unpackaged class
  public static void main( String[] args ){
    //p1.Complex c1 = new p1.Complex( );  //OK
    Complex c1 = new Complex( );
    c1.printRecord( );
  }
}
```

  - How to compile and execute Java Application in Windows?

```
//Make Sure JAVA_HOME is set
javac -d .\bin .\src\Complex.java //Output: p1/Complex.class
set classpath=.\bin;
javac -d .\bin .\src\Program.java //Output: Program.class
java Program  //10,20
```

- **Conclusion:** We can use any packaged class/type inside unpackaged class.
- **Example 3**
  - Consider class Complex(Example_3/src/Complex.java) without package.

```
class Complex{  //unpackaged class
  private int real = 10;
  private int imag = 20;
  public void printRecord( ){
    System.out.println(this.real+","+this.imag);
  }
}
```

  - Consider class Program(Example_3/src/Program.java) with package:

```
package p1;
class Program{  //Packaged class
  public static void main( String[] args ){
    Complex c1 = new Complex( );
    c1.printRecord( );
  }
}
```

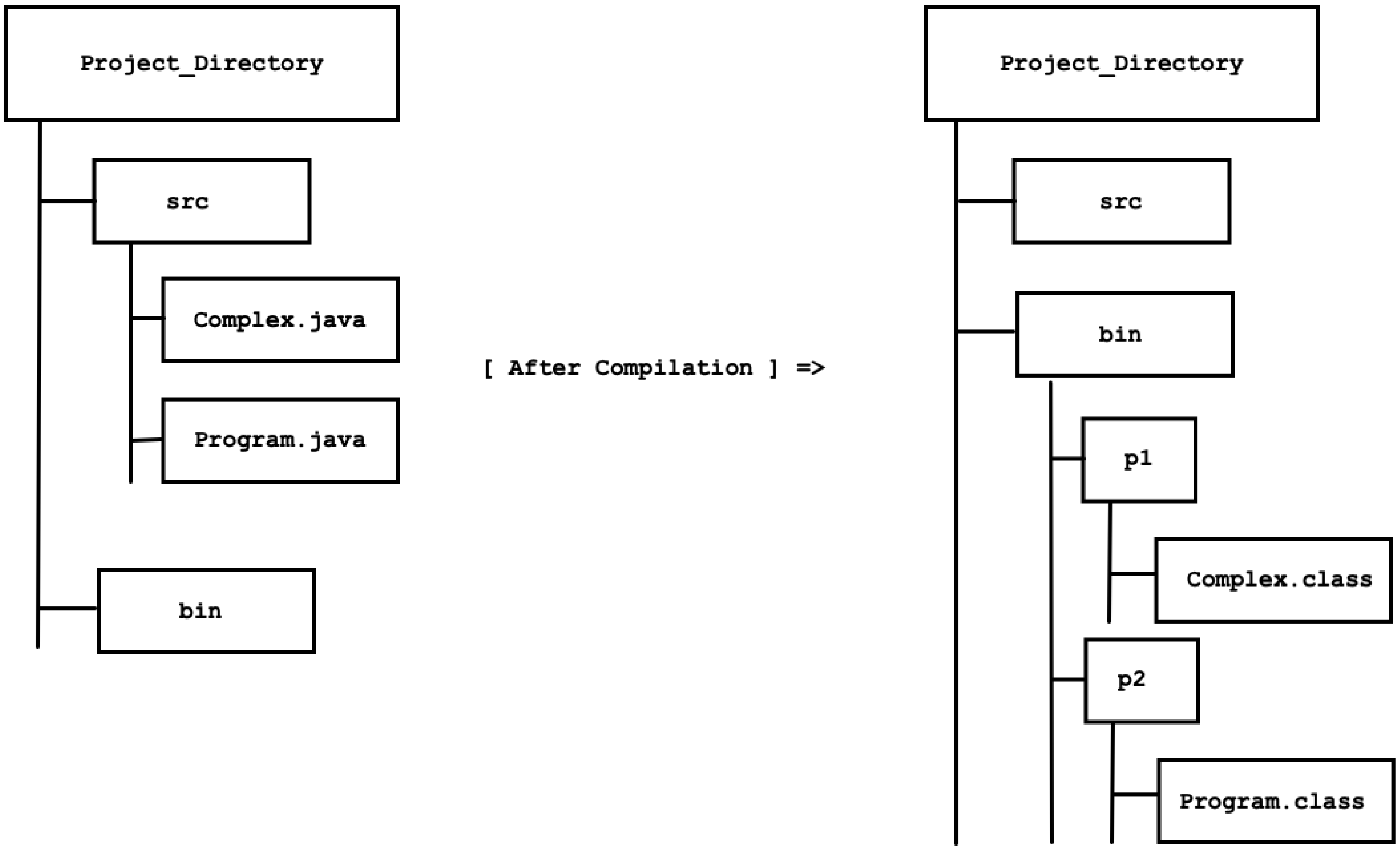
  - How to compile and execute Java Application in Windows?

```
//Make Sure JAVA_HOME is set
javac -d .\bin .\src\Complex.java //Output: Complex.class
set classpath=.\bin;
javac -d .\bin .\src\Program.java //Output: Compiler Error
```

```
error: cannot find symbol Complex
```

- If we define class/type without package then it is considered as a member of default package. **Remember** we can not import default package.

- **Conclusion:** It is impossible to access unpackaged type from packaged type.
- **Example 4**



- Consider class Complex(Example_4/src/Complex.java) with package p1.

```
package p1;
  public class Complex{  //Packaged class
    private int real = 10;
    private int imag = 20;
    public void printRecord( ){
      System.out.println(this.real+","+this.imag);
    }
  }
```

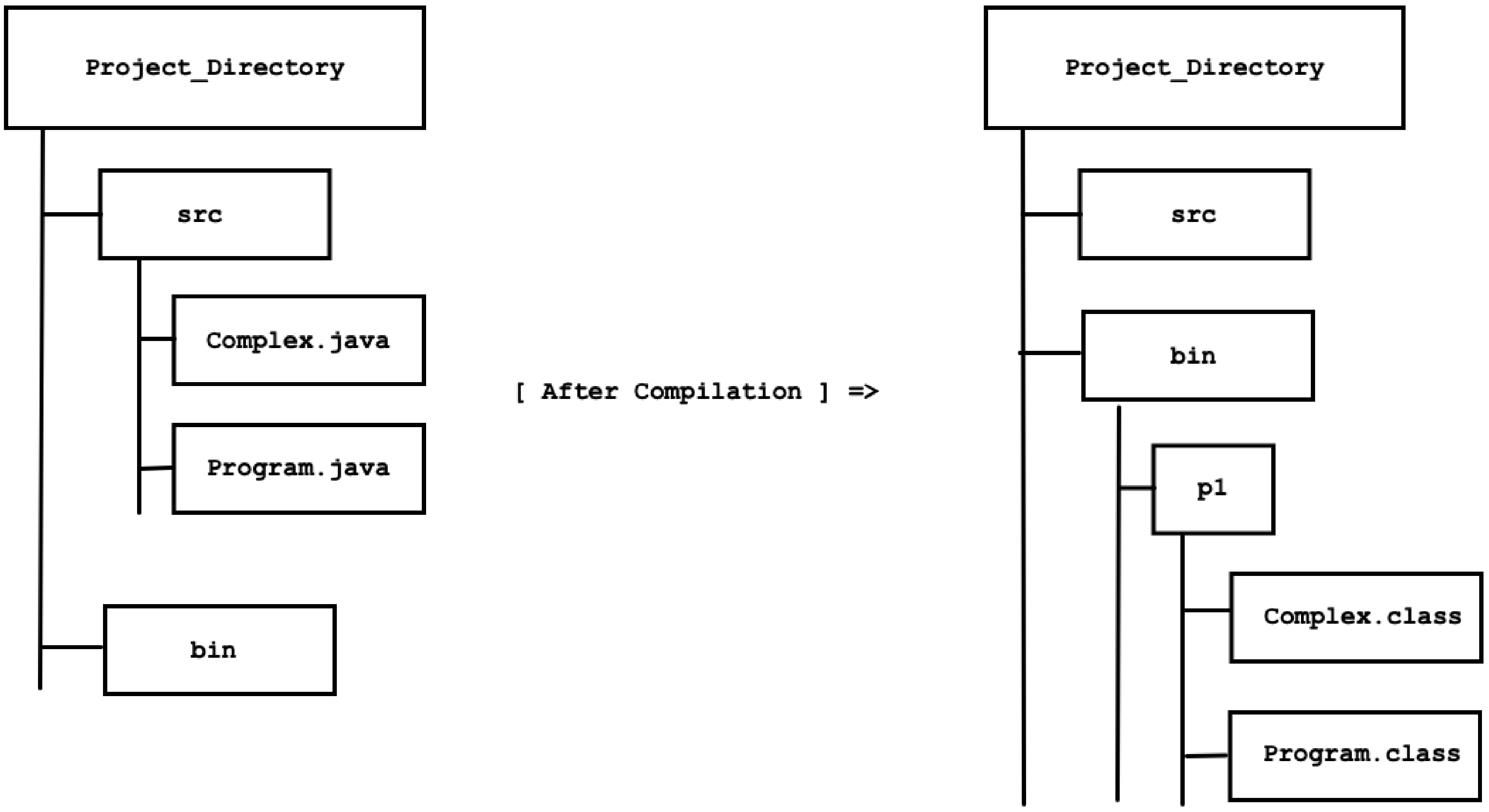
- Consider class Program(Example_4/src/Program.java) with package p2:

```
  package p2;
  import p1.Complex;
  public class Program{  //Packaged class
    public static void main( String[] args ){
      Complex c1 = new Complex( );
      c1.printRecord( );
    }
  }
```

- How to compile and execute Java Application in Windows?

```
//Make Sure JAVA_HOME is set
javac -d .\bin .\src\Complex.java //Output: p1/Complex.class
set classpath=.\bin;
javac -d .\bin .\src\Program.java //Output: p2/Program.class
java p2.Program //10,20
```

- **Conclusion** We can group unrelated types in different package.
- **Example 5**



  - Consider class Complex(Example_5/src/Complex.java) with package p1.

```
package p1;
  public class Complex{  //Packaged class
    private int real = 10;
    private int imag = 20;
    public void printRecord( ){
      System.out.println(this.real+","+this.imag);
    }
  }
```

  - Consider class Program(Example_5/src/Program.java) with package p1:

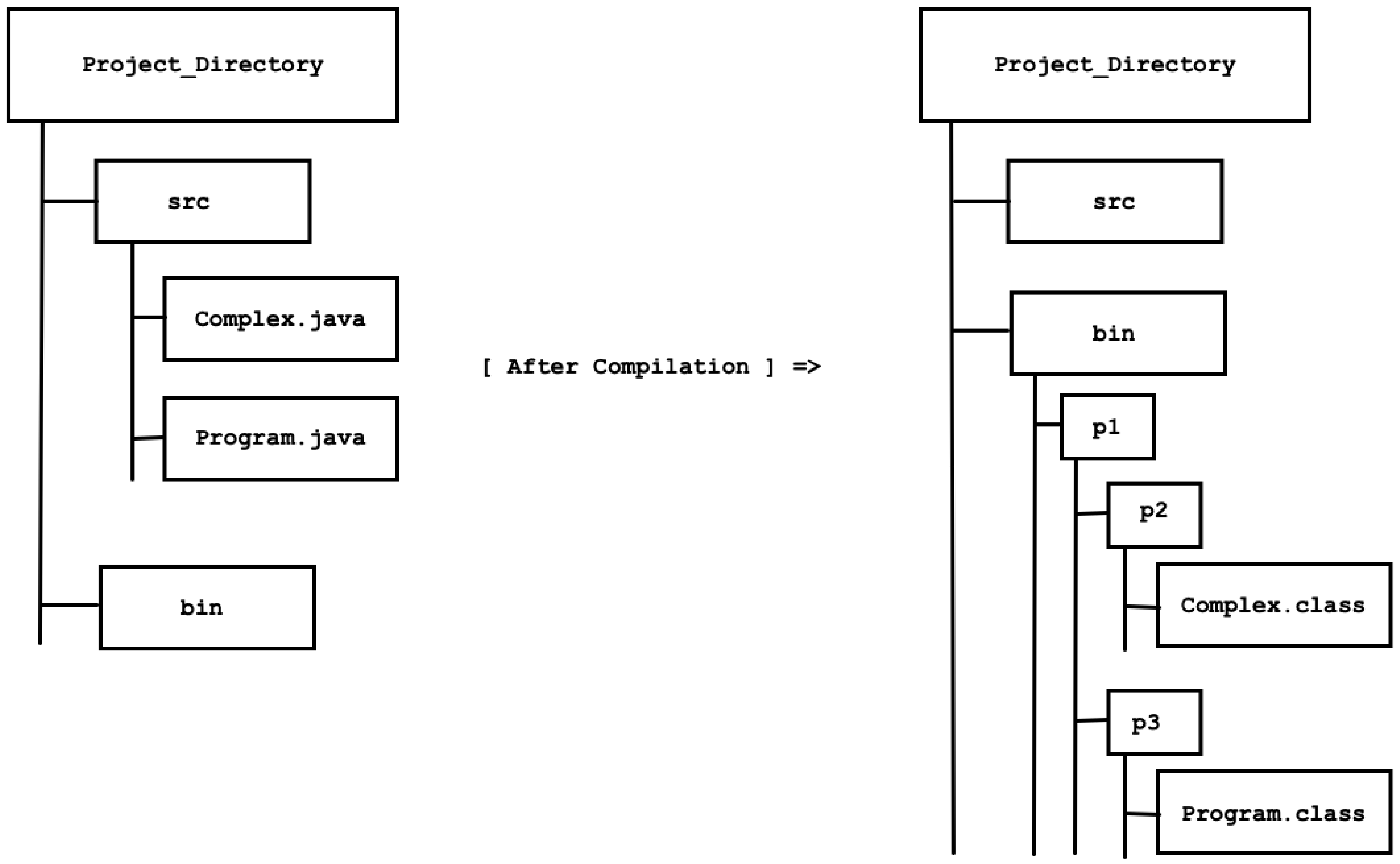
```
  package p1;
  import p1.Complex;  //Optional to import
  public class Program{  //Packaged class
    public static void main( String[] args ){
      Complex c1 = new Complex( );
      c1.printRecord( );
    }
  }
```

  - How to compile and execute Java Application in Windows?

```
//Make Sure JAVA_HOME is set
javac -d .\bin .\src\Complex.java //Output: p1/Complex.class
set classpath=.\bin;
javac -d .\bin .\src\Program.java //Output: p1/Program.class
java p1.Program //10,20
```

- **Conclusion** We can group related types in same package.

- **Example 6**



  - Consider class Complex(Example_6/src/Complex.java) with package p1.p2.

```java
package p1.p2;
  public class Complex{  //Packaged class
    private int real = 10;
    private int imag = 20;
    public void printRecord( ){
      System.out.println(this.real+","+this.imag);
    }
  }
```

  - Consider class Program(Example_6/src/Program.java) with package p1.p3:

```java
package p1.p3;
import p1.p2.Complex;  //Optional to import
public class Program{  //Packaged class
  public static void main( String[] args ){
    Complex c1 = new Complex( );
    c1.printRecord( );
  }
}
```

  - How to compile and execute Java Application in Windows?

```
//Make Sure JAVA_HOME is set
javac -d .\bin .\src\Complex.java //Output: p1/p2/Complex.class
set classpath=.\bin;
javac -d .\bin .\src\Program.java //Output: p1/p3/Program.class
java p1.Program //10,20
```

- **Conclusion** We can group related types in sub package too.
- **Example 7**
  - Option 1

```
class Program{
  public static void main( String[] args ){
    double radius = 10.5d;
    double area = Math.PI * Math.pow( radius, 2 );  //OK
    System.out.println("Area  : "+area);
  }
}
```

- Option 2

```
import static java.lang.System.out;
import static java.lang.Math.PI;
import static java.lang.Math.pow;
class Program{
  public static void main( String[] args ){
    double radius = 10.5d;
    double area = PI * pow( radius, 2 );  //OK
    out.println("Area  : "+area);
  }
}
```

- Sometimes, we need to use final fields and methods from certain classes often. Typing the class name repeatedly can make the code messy. Static import helps by letting us directly use these final fields and methods without typing the class name each time.
- **What we learn about import and static import?**
  - If we want to use public types(interface/class/enum) defined in package outside that package then we should use import.
  - If we want to use static members of the class defined in same/different package without class name then we should use static import.
- **How to import entire package?**

```
import java.lang.*;
import java.lang.refect.*;
```

- **Why we do not import java.lang package?**
  - java.lang package contains all the fundamental types(class/interface) of core Java language.
  - By default, java.lang package gets imported in every .java file. Hence to use any type declared in java.lang package, import statement is not required.
- **Naming convention for package**
  - Package names are written in all lower case to avoid conflict with the names of classes or interfaces.
  - Companies use their reversed Internet domain name to begin their package names.
  - Below are few examples of package names.
    - com.exmple
    - com.google.gson
    - com.mysql.jdbc
    - com.microsoft.sqlserver.jdbc
    - org.example.main
    - org.apache.commons
    - in.cdac.javase.pojo
    - oracle.jdbc.driver