



# Core Java

- Akshita Chanchlani



# Contents

---

- Java Buzzwords (Features)
- Access Specifier
- Class and Object
- Instantiation
- Reference
- this
- Constructor and its types



# Java Buzzwords

---

1. Simple
2. Object Oriented
3. Architecture Neutral
4. Portable
5. Robust
6. Multithreaded
7. Dynamic
8. Secure
- 9.High Performance
- 10.Distributed



# Simple

- Java is **simple** programming language.
  - **Syntax of Java is simpler than syntax of C/C++ hence it is considered as simple**
    - Ø No need of header files and macros.
    - Ø We can not define anything global
    - Ø Do not support structure and union. operator overloading.
    - Ø Do not support copy constructor and assignment operator function constructor member initializer list and default argument constant data member and constant member function.  
Delete operator, destructor , friend function, friend class.  
Multiple class (Multiple inheritance)
    - Ø No diamond problem and virtual base class.
    - Ø Do not support pointer and pointer arithmetic.
  - **Size of software(JDK), that is required to develop Java application is small hence Java is considered as simple programming language.**
- Java does not support multiple inheritance. To some extent, the interface feature provides the desirable features of multiple inheritance to a Java program without some of the underlying problems.(death of a diamond)



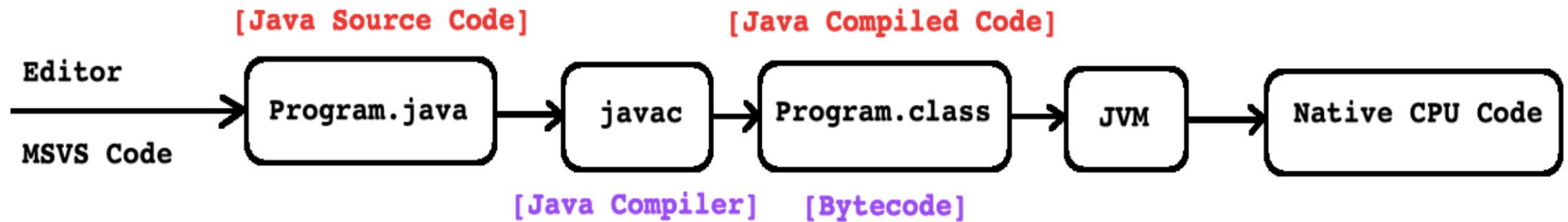
# Object Oriented

- Java is **object oriented** programming language.
  - Java Supports all the major and minor pillars of oops hence it is considered as object oriented programming language.
  - **Major pillars of oops.**
    1. Abstraction
    2. Encapsulation
    3. Modularity
    4. Hierarchy
  - **Minor pillars of oops.**
    1. Typing / Polymorphism
    2. Concurrency
    3. Persistence.

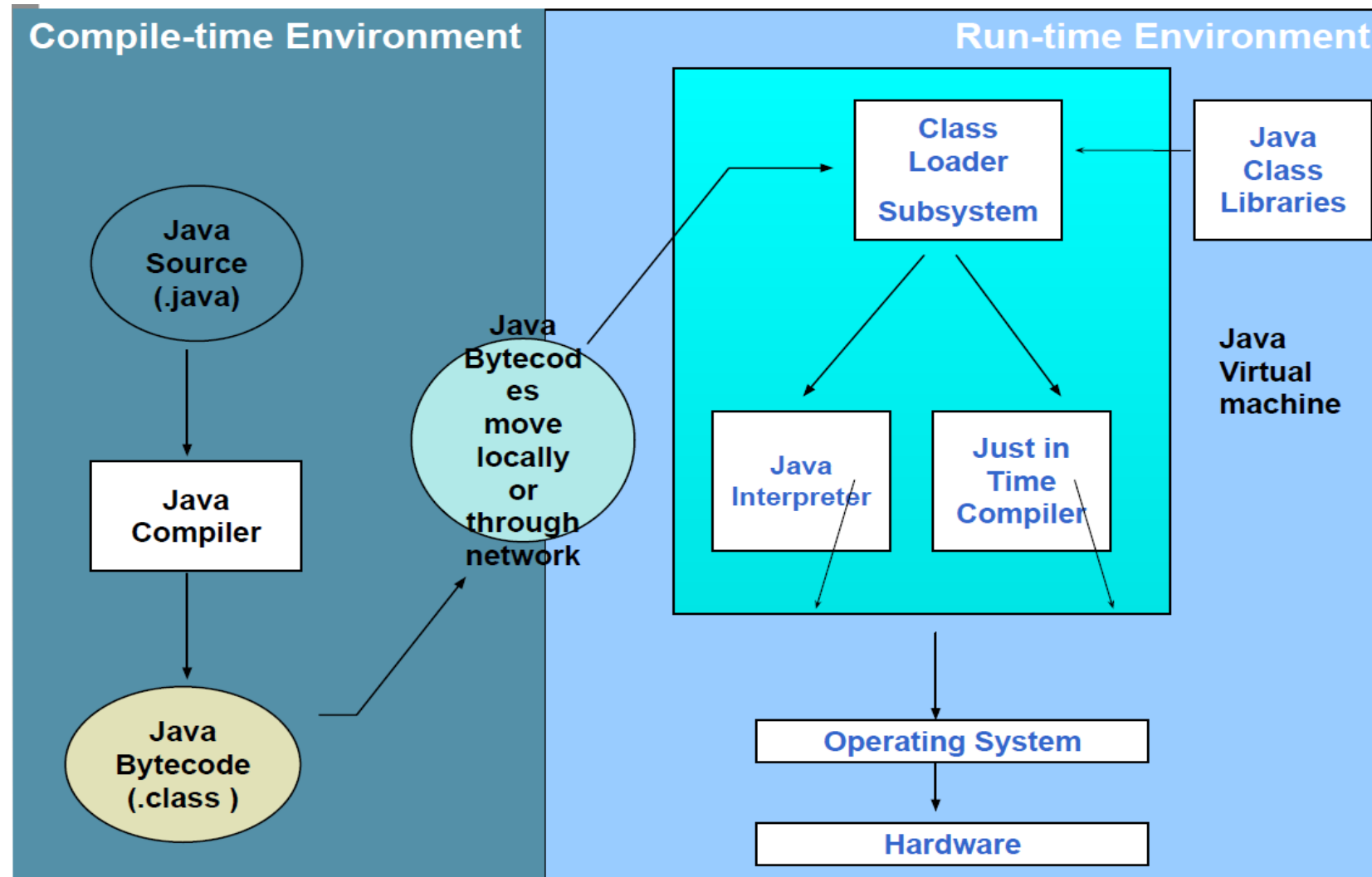


# Architecture Neutral

- Java is object **architecture neutral** programming language.
  - Java technology is designed to support applications that will be deployed into heterogeneous network environments. In such environments, applications must be capable of executing on a variety of hardware architectures. Within this variety of hardware platforms, applications must execute on the top of a variety of operating systems. To accommodate the diversity of operating environments, the Java Compiler product generates *bytecodes*--an *architecture neutral* intermediate format designed to transport code efficiently to multiple hardware and software platforms.



# How it Works



# Java Platform Independent

---

Java is independent only for one reason:

- Only depends on the Java Virtual Machine (JVM),
- code is compiled to bytecode, which is interpreted by the resident JVM,
- JIT (just in time) compilers attempt to increase speed. The Just-In-Time (JIT) compiler is an essential part of the JRE i.e. Java Runtime Environment, that is responsible for performance optimization of java based applications at run time.

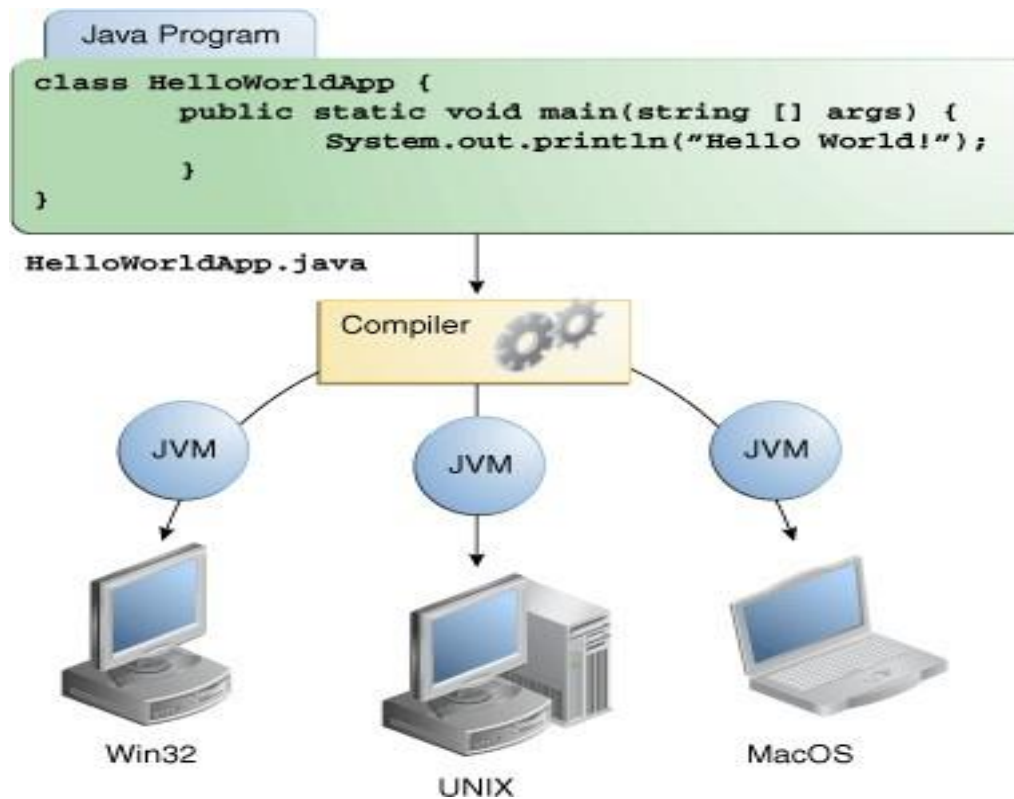
The Just-In-Time (JIT) compiler is a key component of the OpenJ9 VM that improves the performance of Java applications by compiling platform-neutral Java bytecode into native machine code at run time. Without the JIT, the VM has to interpret the bytecodes itself - a process that requires extra CPU and memory.





# Portable

- Java is **portable** programming language.
  - Architecture neutrality is just one part of a truly *portable* system.



# Portable

- Java is **portable** programming language.
  - o Java technology takes portability a stage further by being strict in its definition of the basic language.
  - o Java technology puts a stake in the ground and specifies the sizes of its basic types and the behavior of its data arithmetic operators.
  - o Your programs are the same on every platform--there are no data type **incompatibilities across hardware and software architectures.**

Sr.No.	Primitive Type	Size	Default Value For Field
1	boolean	Isn't Defined	FALSE
2	byte	1 Byte	0
3	char	2 Bytes	\u0000'
4	short	2 Bytes	0
5	int	4 Bytes	0
6	float	4 Bytes	0.0f
7	double	8 Bytes	0.0d
8	long	8 Bytes	0L



# Robust

- Java is **robust** programming language.
  - o The Java programming language is designed for creating highly *reliable* software. It provides extensive compile-time checking, followed by a second level of run- time checking.
  - o Java is robust because of following features:
    1. *Architecture Neutral.*
      - Ø Java developer is free from developing H/W or OS specific coding.
    2. *Object orientation.*
      - Ø Reusability reduces developer's effort.
    3. *Automatic memory management.*
      - Ø Developer need not to worry about memory leakage / program crashes.
    4. *Exception handling.*
      - Ø Java compiler helps developer to provide try-catch block.



# Multithreaded

- Java is **multithreaded** programming language.
  - o When we start execution of Java application then JVM starts execution threads hence Java is considered as multithreaded.
    1. Main thread
      - Ø It is user thread / non daemon thread.
      - Ø It is responsible for invoking main method.
      - Ø Its default priority is 5( Thread.NORM\_PRIORITY ).
    2. Garbage Collector / Finalizer
      - Ø It is daemon thread / background thread.
      - Ø It is responsible for releasing / deallocating memory of unused objects.
      - Ø Its default priority is 8( Thread.NORM\_PRIORITY + 3 ).
  - o The Java platform supports multithreading at the language level with the addition of sophisticated synchronization primitives: the language library provides the Thread class, and the run-time system provides monitor and condition lock primitives. At the library level, moreover, Java technology's high-level system libraries have been written to be thread safe: the functionality provided by the libraries is available without conflict to multiple concurrent threads of execution.



# Dynamic

- Java is **dynamic** programming language.
  - While the Java Compiler is strict in its compile-time static checking, the language and run-time system are *dynamic* in their linking stages. Classes are linked only as needed. New code modules can be linked in on demand from a variety of sources, even from sources across a network.
  - Java is designed to adapt to an evolving environment.
  - Libraries can freely add new methods and instance variables without any effect on their clients.
  - In Java finding out runtime type information is straightforward.
  - In Java, all the methods are by default virtual.



- Java is **secure** programming language.
  - o Java is intended to be used in networked/distributed environments. Toward that end, a lot of emphasis has been placed on security. Java enables the construction of virus-free, tamper-free systems.
  - o From the beginning, Java was designed to make certain kinds of attacks impossible, among them:
    1. Overrunning the runtime stack – a common attack of worms and viruses
    2. Corrupting memory outside its own process space
    3. Reading or writing files without permission

Pointer denial - reduces chances of virulent(harmful – virus prone) programs corrupting host,

■ Applets even more restricted -

– May not

- run local executables,
- Read or write to local file system,
- Communicate with any server other than the originating server.



# High Performance

- Java
  - is **high performance** programming language.
  - o The Java platform achieves superior performance by adopting a scheme by which the interpreter can run at full speed without needing to check the run-time environment.
  - o The *automatic garbage collector* runs as a low-priority background thread, ensuring a high probability that memory is available when required, leading to better performance.
  - o Applications requiring large amounts of compute power can be designed such that compute-intensive sections can be rewritten in native machine code as required and interfaced with the Java platform.
  - o In general, users perceive that interactive applications respond quickly even though they're interpreted.



# Distributed

- Java is **distributed** programming language.
  - Java has an extensive library of routines for coping with protocols like HTTP , TCP/IP and FTP.
  - Java applications can open and access objects across the Net via URL with the same ease as when accessing a local file system.
- o Nowadays, one takes this for granted, but in 1995, connecting to a web server from a C++ or Visual Basic program was a major undertaking.





# How to access members of package?

Package : p1

```
public class Complex{  
    //TODO  
}
```

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

1

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

2



# Modifier

---

1. ABSTRACT
2. FINAL
3. INTERFACE
4. NATIVE
5. PRIVATE
6. PROTECTED
7. PUBLIC
8. STATIC
9. STRICT
10. SYNCHRONIZED
11. TRANSIENT
12. VOLATILE



# Access Modifier

- If we want to control visibility of members of class then we should use access modifier.
- There are 4 access modifiers in Java:
  1. private
  2. package-level private / default
  3. protected
  4. public

Access Modifiers	Same Package			Different Package	
	Same class	Sub class	Non sub class	Sub class	Non Sub class
private	A	NA	NA	NA	NA
package level private/Default	A	A	A	NA	NA
protected	A	A	A	A	NA
public	A	A	A	A	A



# Class

- Consider following examples:
  1. day, month, year - related to - Date
  2. hour, minute, second - related to - Time
  3. red, green, blue - related to Color
  4. real, imag - related to - Complex
  5. xPosition, yPosition - related to Point
  6. number, type, balance - related to Account
  7. name, id, salary - related to Employee
- If we want to group related data elements together then we should use/define class in Java.

<pre>class Date{     int day;        //Field     int month;      //Field     int year;       //Field }</pre>	<pre>class Employee{     String name;    //Field     int id;         //Field     float salary;   //Field }</pre>
--	--



# Class and object

- class is a non primitive/reference type in Java.
- Classes and Objects are basic concepts of Object Oriented Programming which revolve around the real life entities.

A **class** is a user defined blueprint or prototype or template : from which objects are created.

It represents the set of properties(DATA) and methods(ACTIONS) that are common to all objects of one type.

## Class declaration includes

1. Access specifiers : A class can be public or has default access
2. Class name: The name should begin with a capital letter & then follow camel case convention
3. Superclass(if any): The name of the class's parent (superclass), if any, preceded by the keyword extends. (Implicit super class of all java classes is java.lang.Object)
4. Interfaces(if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.  
eg : public class Emp extends Person implements Runnable,Comparable{...}
5. Body: The class body surrounded by braces, { }.
6. Constructors are used for initializing state of the new object/s.
7. Fields are variables that provides the state of the class and its objects
8. Methods are used to implement the behavior of the class and its objects.

eg : Student,Employee,Flight,PurchaseOrder, Shape ,BankAccount.....



# Object

- It is a basic unit of Object Oriented Programming and represents the real life entities. A typical Java program creates many objects, which interact by invoking methods.
- An object consists of :
  - State : It is represented by attributes of an object. (properties of an object) / instance variables(non static)
  - Behavior : It is represented by methods of an object (actions upon data)
  - Identity : It gives a unique identity to an object and enables one object to interact with other objects. eg : Emp id / Student PRN / Invoice No
- Creating an object
  - The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the class constructor.



# Class

- **Field**
  - Ø A variable declared inside class / class scope is called a field.
  - Ø Field is also called as attribute or property.
- **Method**
  - Ø A function implemented inside class/class scope is called as method.
  - Ø Method is also called as operation, behavior or message.
- **Class**
  - Ø Class is a collection of fields and methods.
  - Ø Class can contain
    1. Nested Type
    2. Field
    3. Constructor
    4. Method
- **Instance** : In Java, Object is also called as instance.

Note: All stand-alone C++ programs require a function named main and can have numerous other functions. Java does not have stand alone functions, all functions (called methods) are members of a class. All classes in Java ultimately inherit from the Object class, while it is possible to create inheritance trees that are completely unrelated to one another in C++. In this sense , Java is a pure Object oriented language, while C++ is a mixture of Object oriented and structure language.



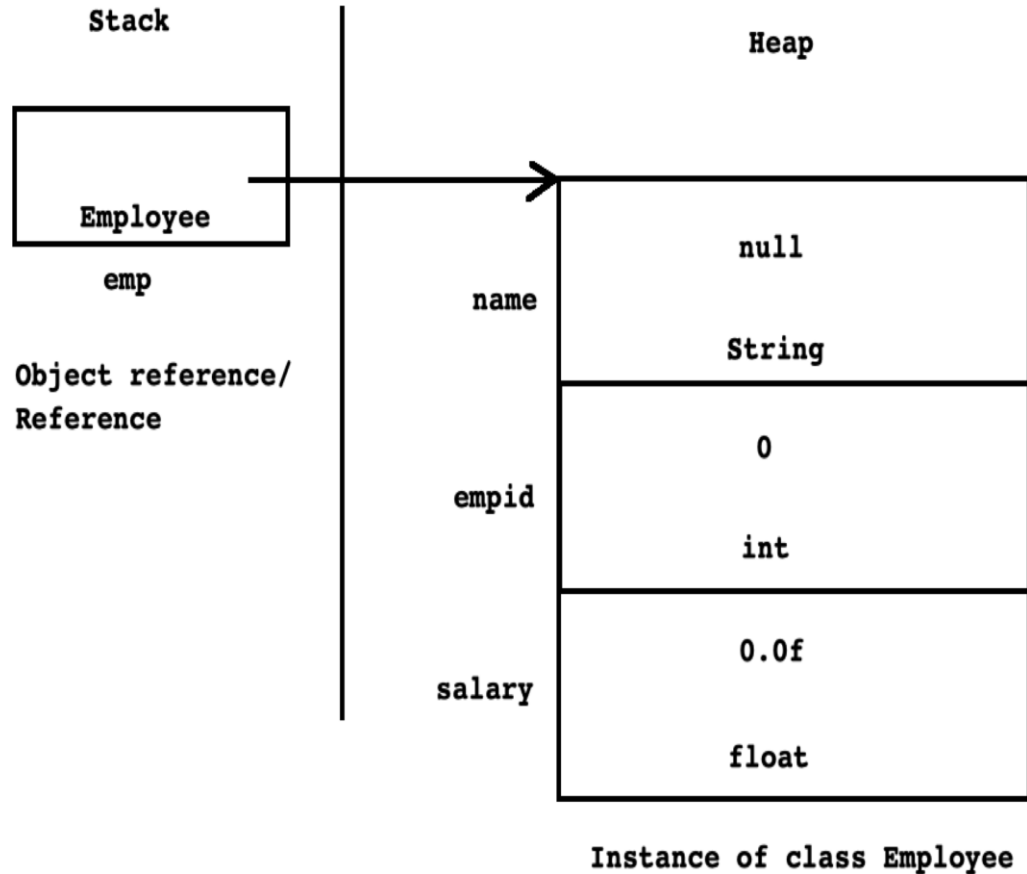
# Instantiation

- Process of creating instance/object from a class is called as instantiation.
- In C programming language
  - Ø Syntax : struct StructureName identifier\_name; struct
  - Ø Employee emp;
- In C++ programming language
  - Ø Syntax : [class] ClassName identifier\_name;
  - Ø Employee emp;
- In Java programming language
  - Ø Syntax : ClassName identifier\_name = new ClassName( );
  - Ø Employee emp = new Employee();
- **Every instance on heap section is anonymous.**



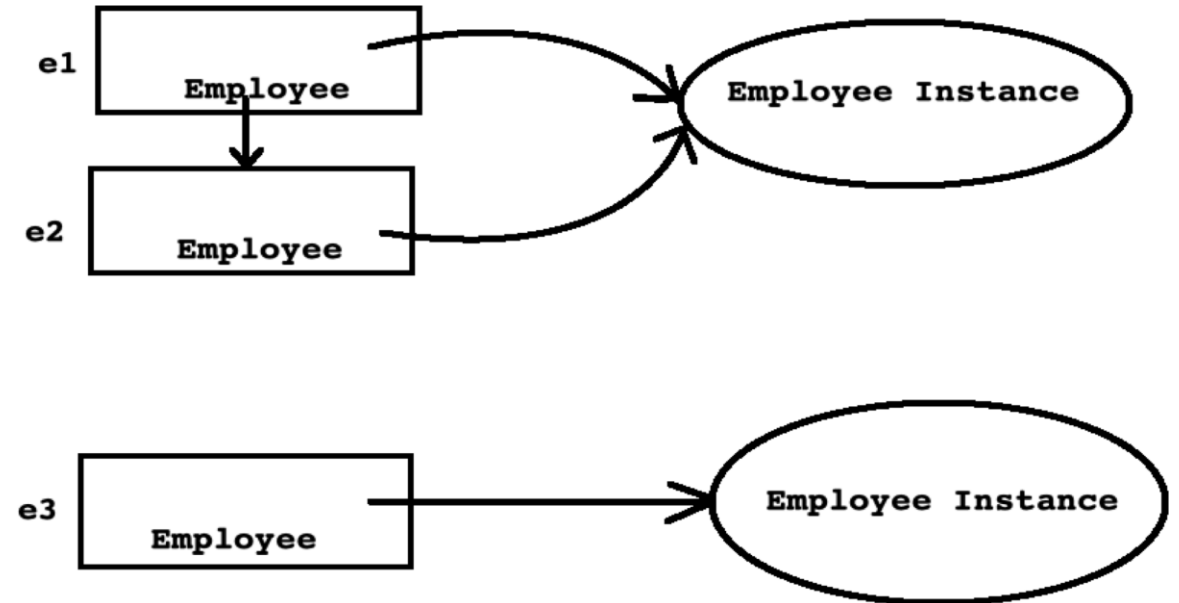


# Instantiation

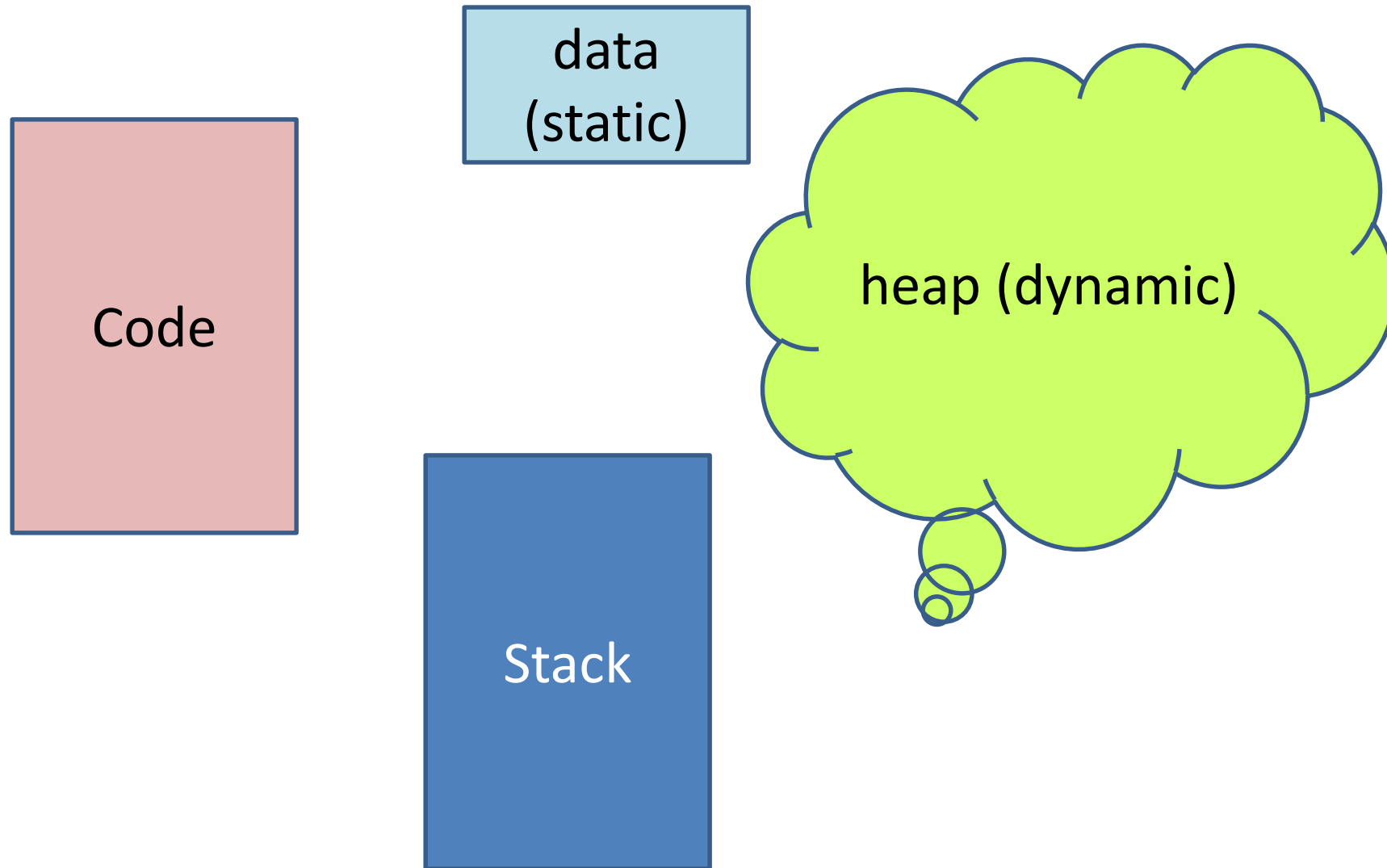


For eg :

1. `Employee e1 = new Employee();`
2. `Employee e2 = e1;`
3. `Employee e3 = new Employee();`



# Types of memory used by executable task



# this current object reference

- If we call non static method on instance( actually object reference ) then compiler implicitly pass, reference of current/calling instance as a argument to the method implicitly. To store reference of current/calling instance, compiler implicitly declare one reference as a parameter inside method. It is called this reference.
- **Using this reference, non static fields and non static methods are communicating with each other. Hence this reference is considered as a link/connection between them.**
- Definition
  - Ø **"this" is implicit reference variable that is available in every non static method of class which is used to store reference of current/calling instance.**
- Inside method, to access members of same class, use this keyword is optional

## Uses of this keyword :

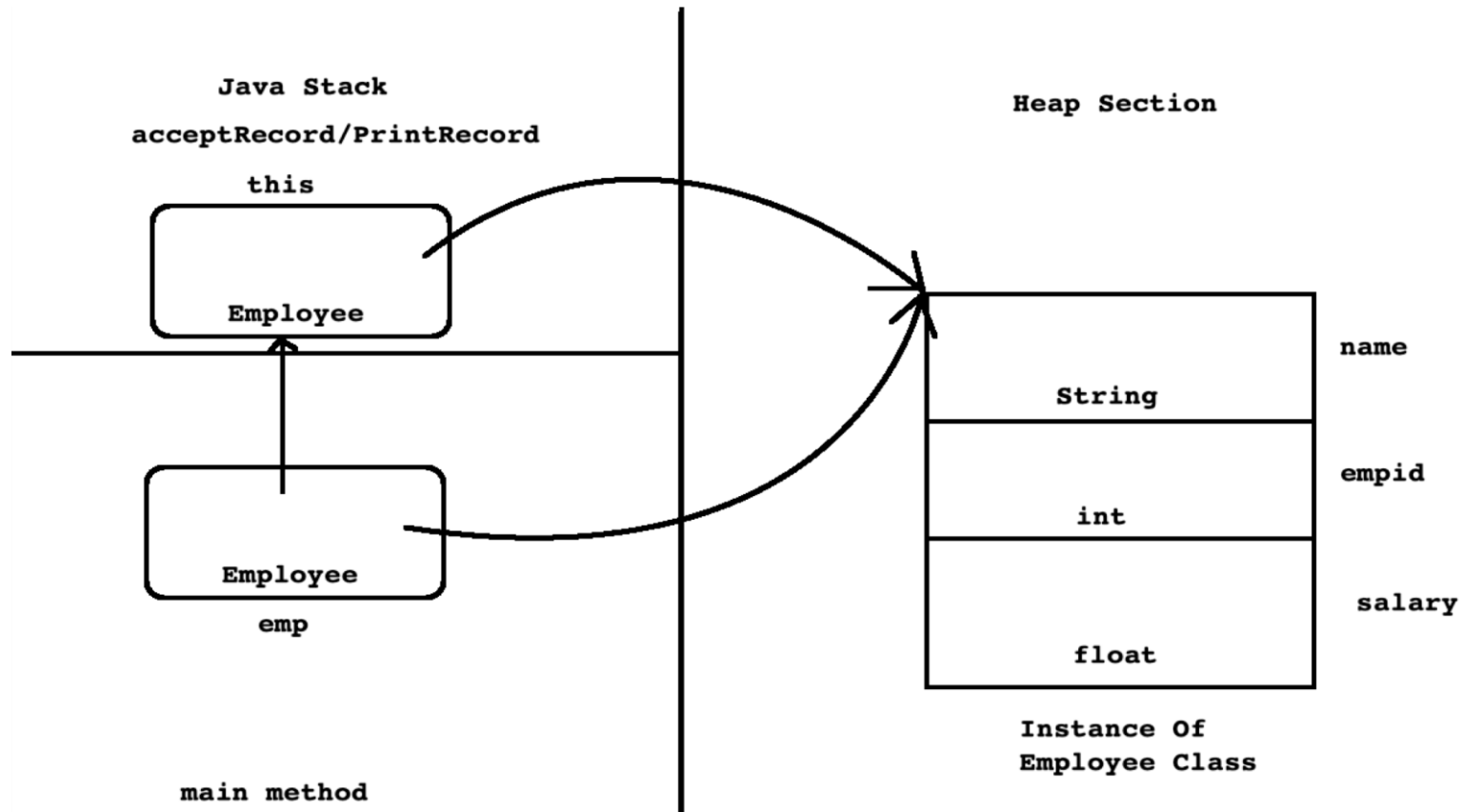
1. To unhide , instance variables from method local variables.(to resolve the conflict)

eg : this.name=name;

2. To invoke the constructor , from another overloaded constructor in the same class.(constructor chaining , to avoid duplication)



# this reference



# this reference

- If name of local variable/parameter and name of field is same then preference is always given to the local variable.

```
class Employee{  
    private String name;  
    private int empid;  
    private float salary;  
    public void initEmployee(String name, int empid, float salary ){  
        this.name = name;  
        this.empid = empid;  
        this.salary = salary;  
    }  
}
```



# Constructor

- If we want to initialize instance then we should define constructor inside class.
- Constructor look like method but it is not considered as method.
- It is special because:
  - Its name is same as class name.
  - It doesn't have any return type.
  - It is designed to be called implicitly
  - It is called once per instance.
- We can not call constructor on instance explicitly

```
Employee emp = new Employee();  
emp.Employee( ); //Not Ok
```

- **Types of constructor:**
  1. Parameterless constructor
  2. Parameterized constructor
  3. Default constructor .



# Parameterless Constructor

- If we define constructor without parameter then it is called as parameterless constructor.
- It is also called as zero argument / user defined default constructor.
- If we create instance without passing argument then parameterless constructor gets called.

```
public Employee( ){  
    //TODO  
}
```

```
Employee emp = new Employee( ); //Here on instance parameterless ctor will call.
```



# Parameterized Constructor

- If we define constructor with parameter then it is called as parameterized constructor.
- If we create instance by passing argument then parameterized constructor gets called.

```
public Employee( String name, int empid, float salary ){  
    //TODO  
}
```

```
Employee emp = new Employee( "ABC", 123, 8000 ); //Here on instance parameterized ctor will call.
```





# Default Constructor

- If we do not define any constructor inside class then compiler generate one constructor for the class by default. It is called default constructor.
- Compiler generated default constructor is parameterless.
- Compiler never generate default parameterized constructor. In other words, if we want to create instance by passing arguments then we must define parameterized constructor inside class.



# Constructor Chaining

- We can call constructor from another constructor. It is called constructor chaining.
- For constructor chaining, we should use this statement.
- this statement must be first statement inside constructor body.
- Using constructor chaining, we can reduce developers effort.

```
class Employee{
    //TODO : Field declaration
    public Employee( ){
        this( "None", 0, 8500 );    //Constructor Chaining
    }
    public Employee( String name, int empid, float salary ){
        this.name = name;
        this.empid = empid;
        this.salary = salary;
    }
}
```



# Why java doesn't support c++ copy constructor?

Java does. They're just not called implicitly like they are in C++ .

Firstly, a copy constructor is nothing more than:

```
public class Demo {  
    private int val;  
  
    public Demo() { } // public no-args constructor  
    public Demo(Demo b) { val = b.val; } // copy constructor  
}
```

Now C++ will implicitly call the copy constructor with a statement like this:

```
Demo b2 = b1;
```

Cloning/copying in that instance simply makes no sense in Java because all b1 and b2 are references and not value objects like they are in C++. In C++ that statement makes a copy of the object's state. In Java it simply copies the reference. The object's state is not copied so implicitly calling the copy constructor makes no sense.



# null

If reference variable contains null value then it is called null reference variable / null object.

```
class Program{  
    public static void main(String[] args) {  
        Employee emp; //Object reference / reference  
        emp.printRecord(); //error: variable emp might not have been initialized  
    }  
}
```

```
public static void main(String[] args) {  
    Employee emp = null; //null reference variable / null object  
    emp.printRecord();    //NullPointerException  
}
```



# Value type VS Reference Type

Sr. No.	Value Type	Reference Type
1	primitive type is also called as value type.	Non primitive type is also called as reference type.
2	boolean, byte, char, short, int, float, double, long are primitive/value type.	Interface, class, type variable and array are non primitive/reference type.
3	Variable of value type contains value.	Variable of reference type contains reference.
4	Variable of value type by default contains 0 value.	Variable of reference type by default contain null reference.
5	We can not create variable of value type using new operator.	It is mandatory to use new operator to create instance of reference type.
6	variable of value type get space on Java stack.	Instance of reference type get space on heap section.
7	We can not store null value inside variable of value type.	We can store null value inside variable reference type.
8	In case of copy, value gets copied.	In case of copy, reference gets copied.



# Coding Convention

- **Pascal Case Coding/Naming Convention:**

- Example

1. System
2. StringBuilder
3. NullPointerException
4. IndexOutOfBoundsException

- In this case, including first word, first character of each word must in upper case.

- We should use this convention for:

1. Type Name( Interface, class, Enum, Annotation )
2. File Name



# Coding Convention

- **Camel Case Coding/Naming Convention:**

- o Example

1. main
2. parseInt
3. showInputDialog
4. addNumberOfDays

- o In this case, excluding first word, first character of each word must in upper case.

- o We should use this convention for:

1. Method Parameter and Local variable
2. Field
3. Method
4. Reference



# Coding Convention

- **Naming Convention for package:**

- We can specify name of the package in uppercase as well as lower-case. But generally it is mentioned in lower case.

- Example

- 1. `java.lang`

- 2. `java.lang.reflect`

- 3. `java.util`

- 4. `java.io`

- 5. `java.net`

- 6. `java.sql`





# Coding Convention

- **Naming Convention for constant variable and enum constant:**

- Example

1. `public static final int SIZE;`

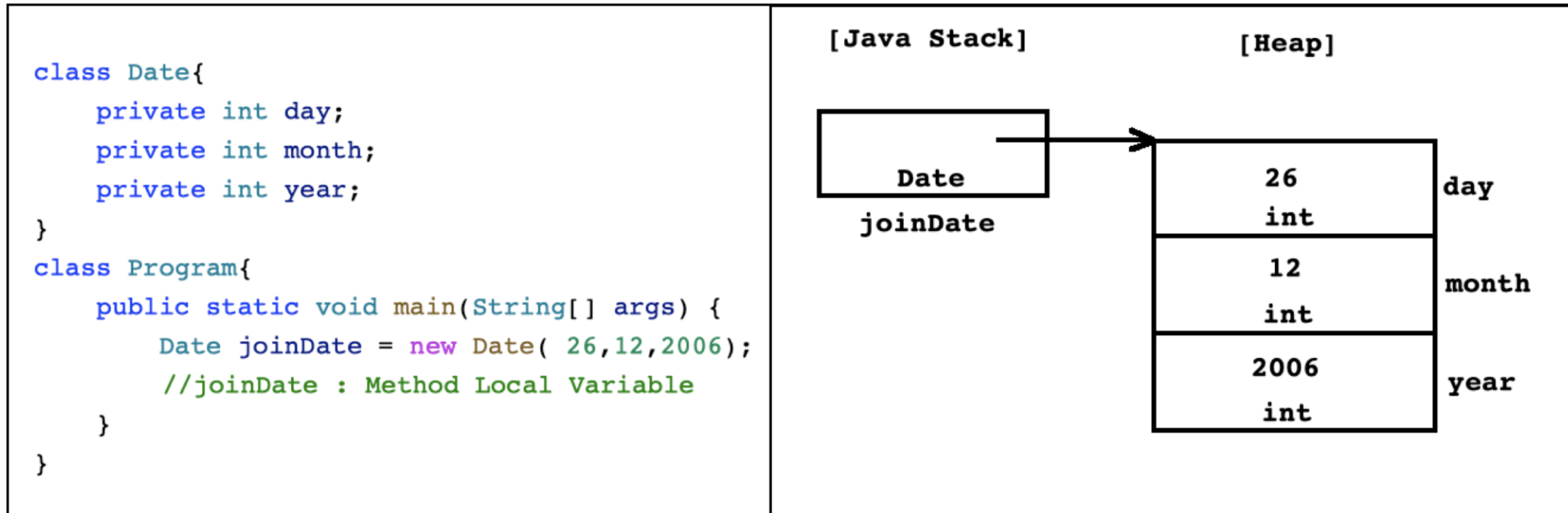
2. `enum Color{ RED, GREEN, BLUE }`

3. Name of the final variable and name of the enum constant should be in upper case.



# Reference

- Local reference variable get space on Java Stack.



- In above code joinDate is method local reference variable hence it gets space on Java Stack.



# Reference

- Class scope reference variable get space on heap.

```
class Employee{
    private String name;
    private int empid;
    private float salary
    private Date joinDate; //joinDate : Field
    public Employee( String name, int empid, float salary, Date joinDate ){
        this.name = name;
        this.empid = empid;
        this.salary = salary;
        this.joinDate = joinDate;
    }
}
```

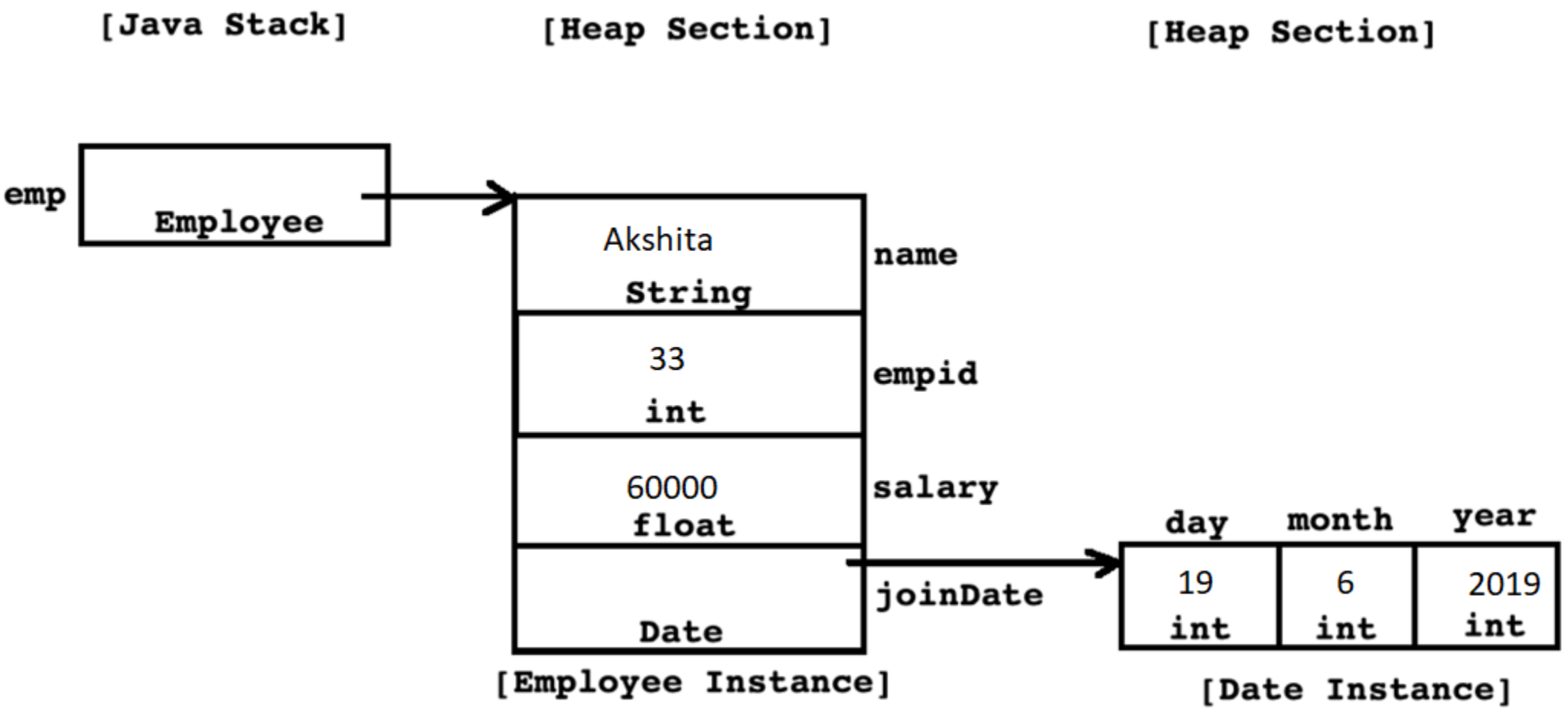
## Class Program

```
{
    public static void main(String[] args)
    {
        Employee emp = new Employee ("Akshita",33,60000,new Date(19,06,2019));
    }
}
```

- In above code, emp is method local reference variable hence it gets space on Java Stack. But joinDate is field of Employee class hence it will get space inside instance on Heap.



# Reference





**Thank you.**  
**akshita.Chanchlani@sunbeaminfo.com**

