

Object Oriented Programming in Java

PSG COLLEGE OF TECHNOLOGY, COIMBATORE

DEPARTMENT OF APPLIED MATHEMATICS AND COMPUTATIONAL SCIENCES

20XW52 – JAVA PROGRAMMING

Programming Paradigms

programming “technique”
way of thinking about programming
view of a program

Java Introduction

- Java is a **true OO language** and therefore the underlying structure of all Java programs is classes.
- Anything we wish to represent in Java must be encapsulated in a class that defines the “state” and “behaviour” of the basic program components known as objects.
- Classes create objects and objects use methods to communicate between them. They provide a convenient method for **packaging a group of logically related data items and functions that work on them.**
- A class essentially serves as a template for an object and behaves like a basic data type “int”. It is therefore important to understand how the fields and methods are defined in a class and how they are used to build a Java program that incorporates the basic **OO concepts such as encapsulation, inheritance, and polymorphism.**

Programming Paradigms

- Procedural Programming
 - program as a collection of statements and procedures affecting data (variables)
- Object-Oriented Programming
 - program as a collection of classes for interacting objects
- Functional Programming
 - program as a collection of (math) functions
- Others

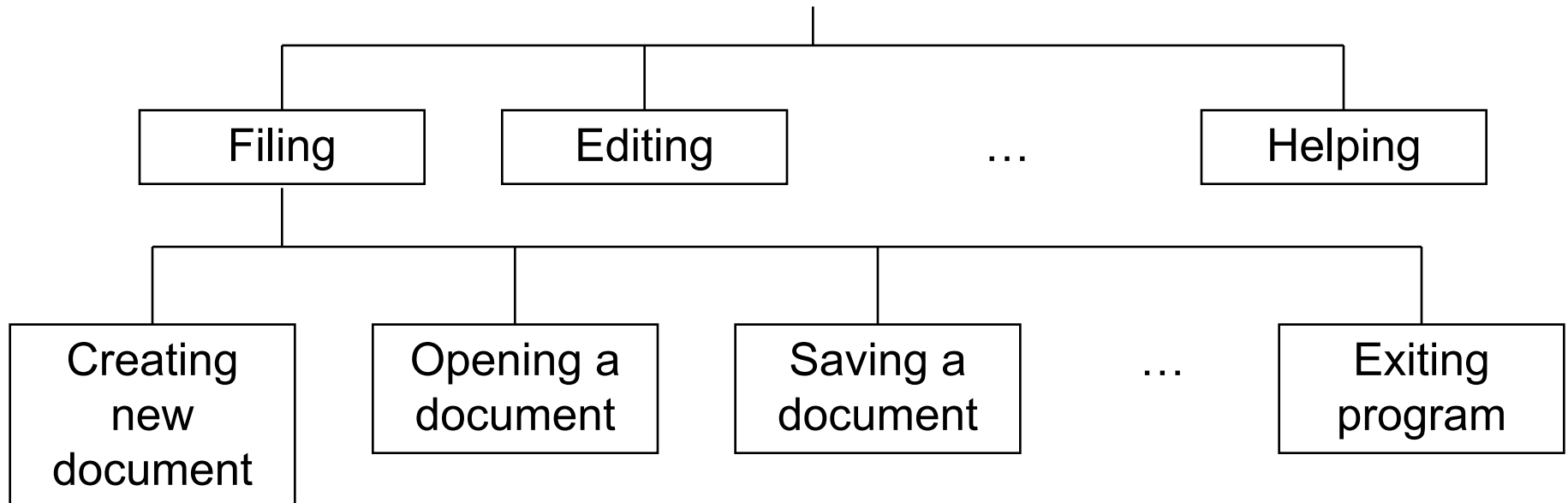
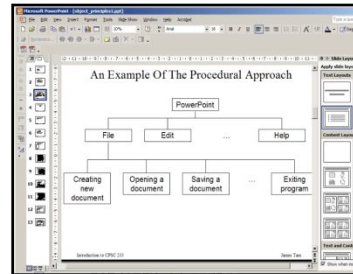
Some Languages by Paradigm

- Imperative (also called Structured or Procedural) Programming
 - FORTRAN, BASIC, COBOL, Pascal, C
- Object-Oriented Programming
 - SmallTalk, C++, Java
- Functional Programming
 - LISP, MetaLanguage, Haskell

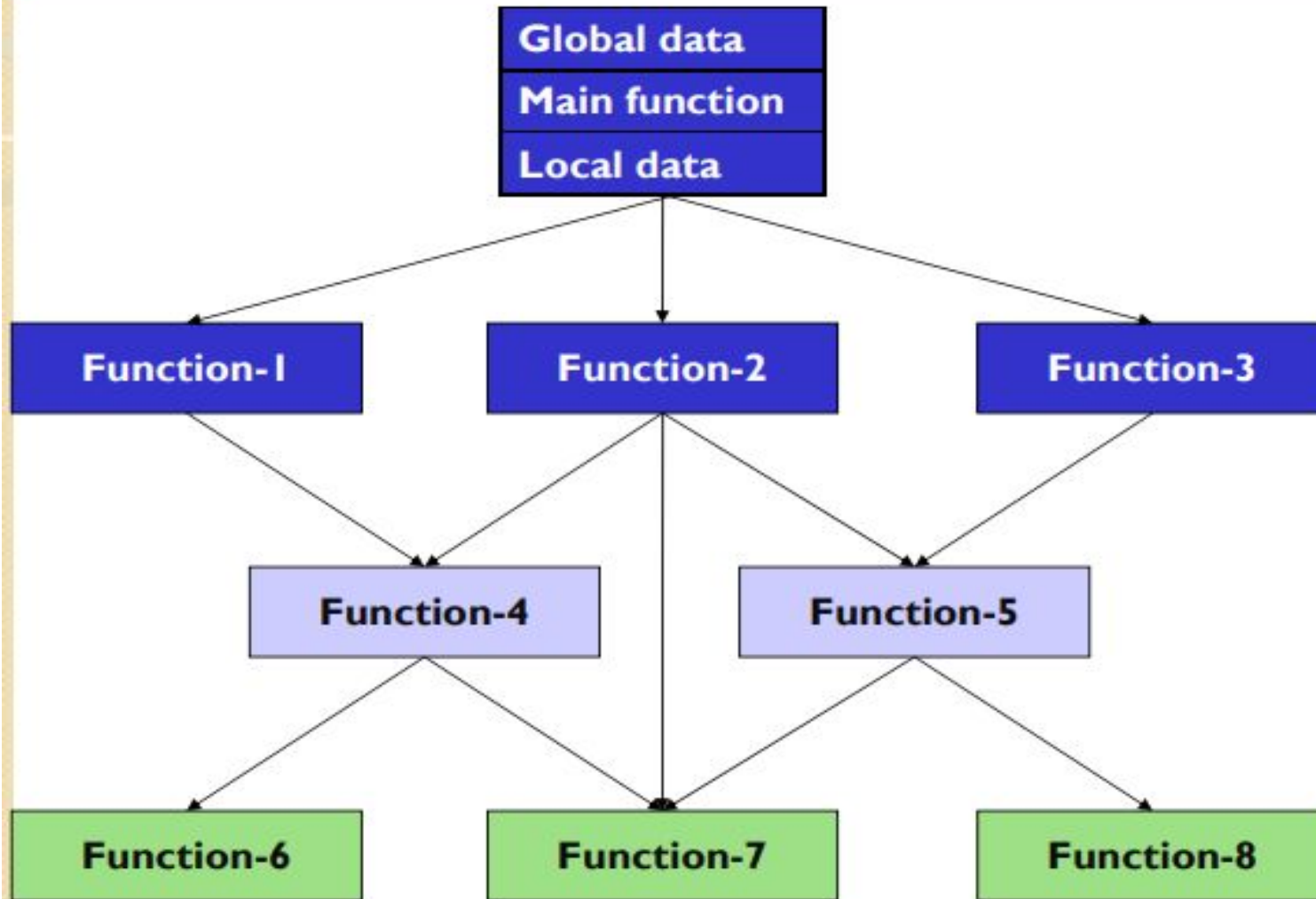
An Example Of The Procedural Approach (Presentation Software)

PowerPoint

Break down the
program by what it does
(described with
actions/verbs)



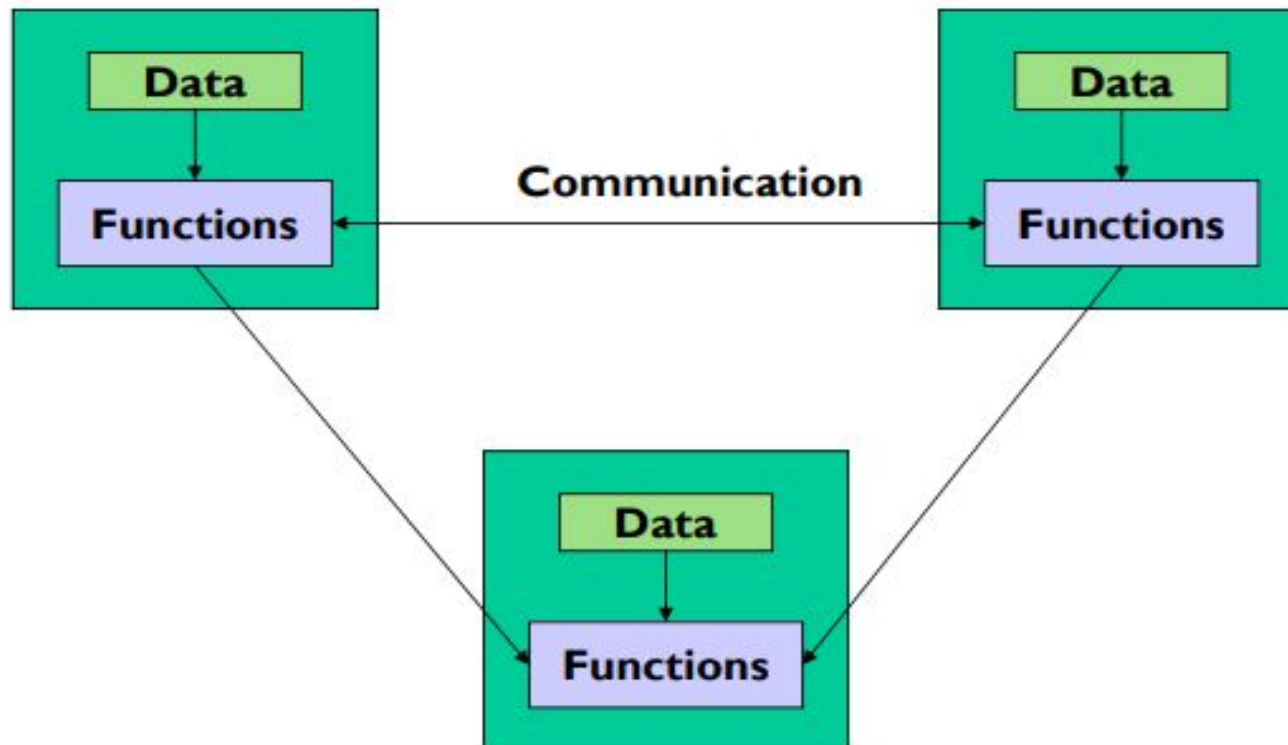
Typical structure of procedure-oriented program



Paradigm Change from Procedural to OO

- Grady Booch has defined object-oriented programming as “a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships”.
- **“Object Oriented Programming models real-world objects with software counterparts”, H.M.Deitel & P.J.Deitel**
- Case study: from C to C++(stack)
 - Evolution from procedural, to modular, to object-based, to object-oriented programming
 - IN OOP we think in terms of **OBJECTS**

Organization of data and functions in OOP

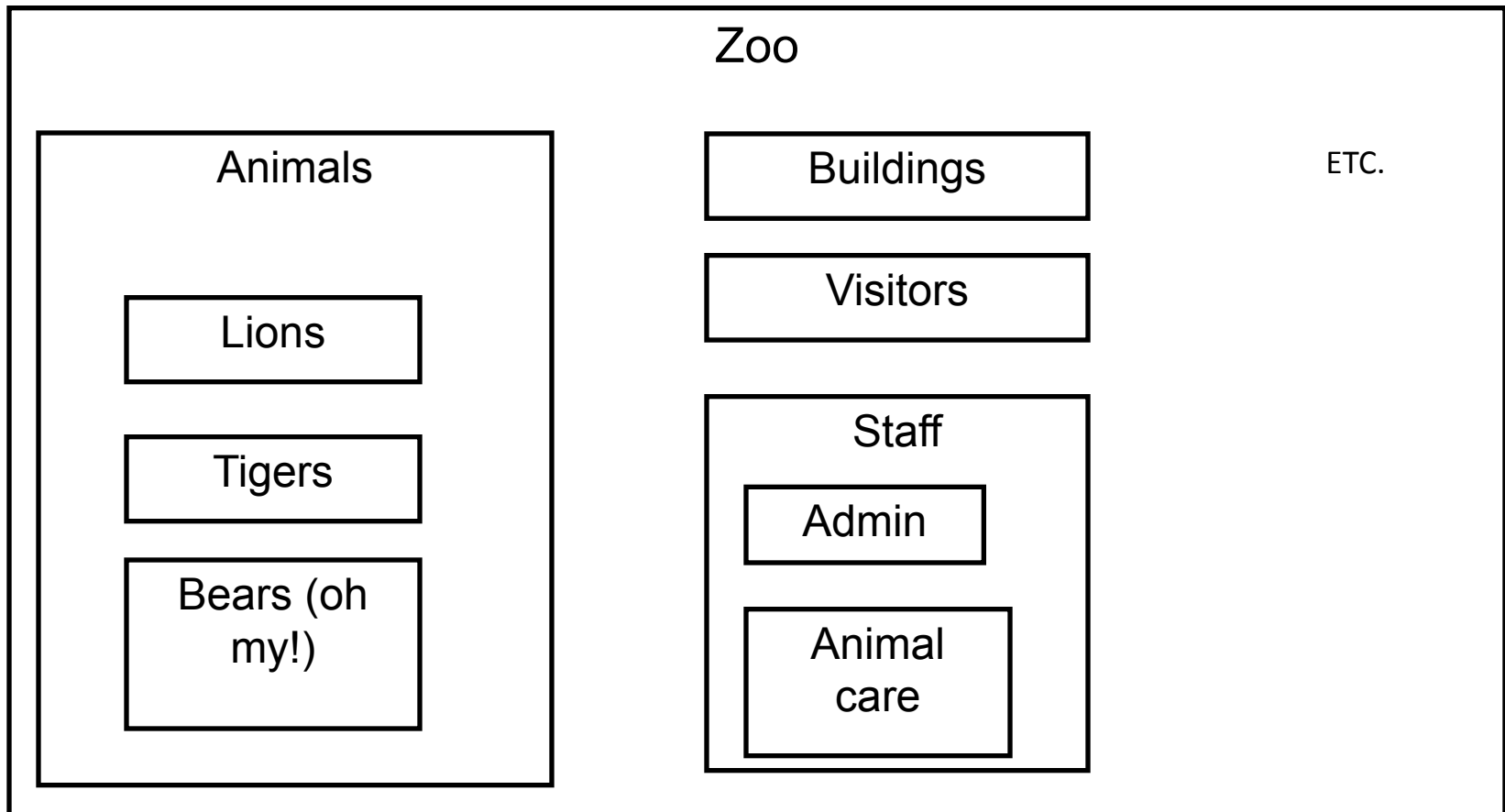


Finding the objects-Natural lang analysis

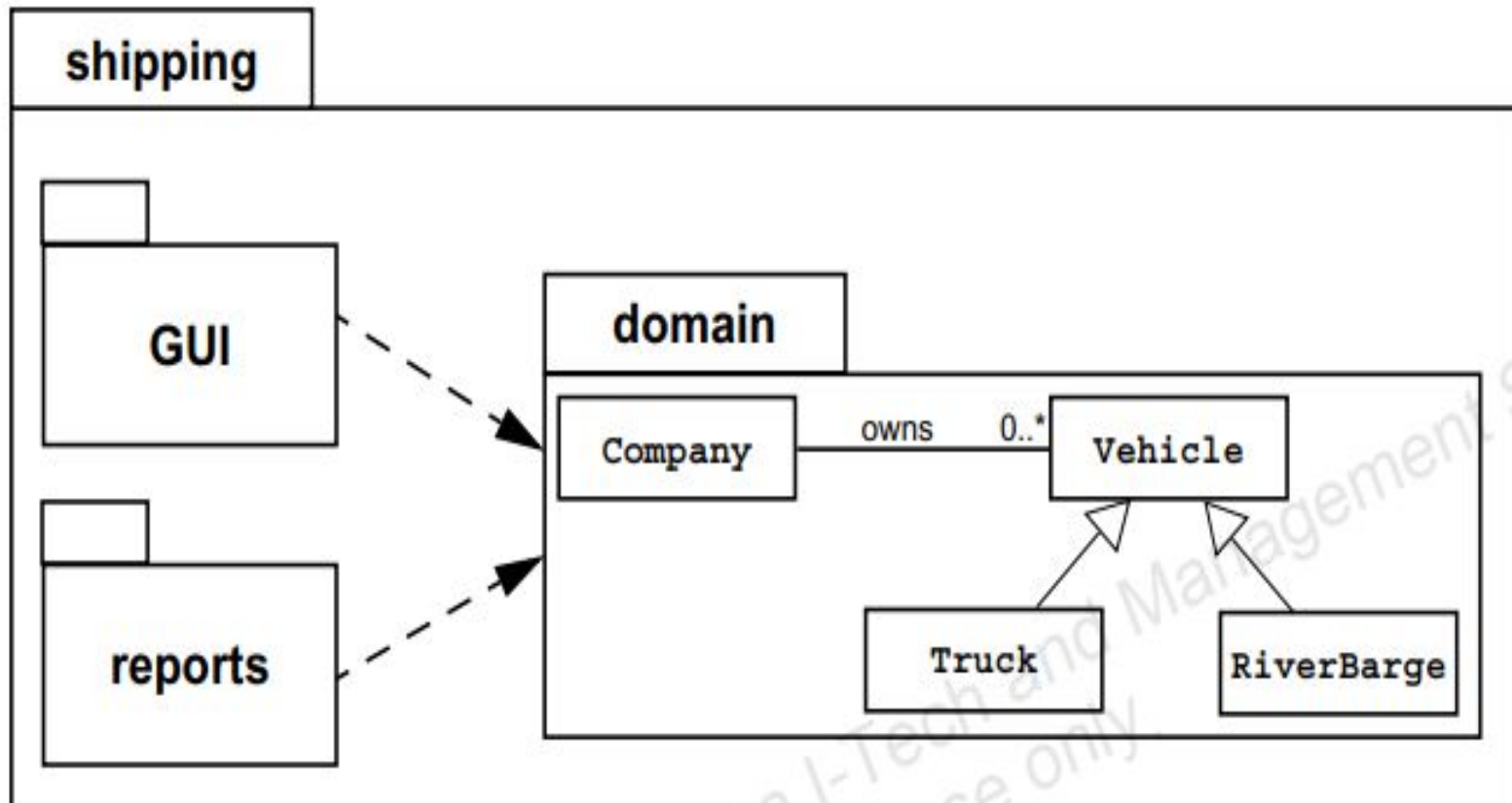
| Part of speech | model component | Examples |
|----------------|-----------------------------|-----------------------------------|
| Proper noun | Instance (object) | Alice, Ace of Hearts |
| Common noun | Class (or attribute) | Field Officer, PlayingCard, value |
| Doing verb | Operation | Creates, submits, shuffles |
| Being verb | Inheritance | Is a kind of, is one of either |
| Having verb | Aggregation/Composition | Has, consists of, includes |
| Modal verb | Constraint | Must be |
| Adjective | Helps identify an attribute | a <i>yellow</i> ball (i.e. color) |

An Example Of The Object-Oriented Approach (Simulation)

- Break down the program into entities (*classes/objects* - described with *nouns*) *methods--verbs*



Shipping project



OOP: Languages

- Concerning the **degree of object orientation**, following distinction can be made:
 - Languages called "**pure**" **OO languages**, because everything in them is treated consistently as an object, from primitives such as characters and punctuation, all the way up to whole classes, prototypes, blocks, modules, etc. They were designed specifically to facilitate, even enforce, OO methods. Examples: **Smalltalk, Eiffel, Ruby, JADE**.
 - Languages designed mainly for **OO programming, but with some procedural elements**. Examples: **C++, C#, Java, Scala, Python**.
 - Languages that are **historically procedural languages, but have been extended with some OO features**. Examples: **VB.NET** (derived from VB), **Fortran 2003, Perl, COBOL 2002, PHP**.

OOP: Variations

- object-based

- objects + classes - inheritance
- classes are declared and objects are instantiated
- no inheritance is defined between classes
- No polymorphism is possible

- example: VisualBasic

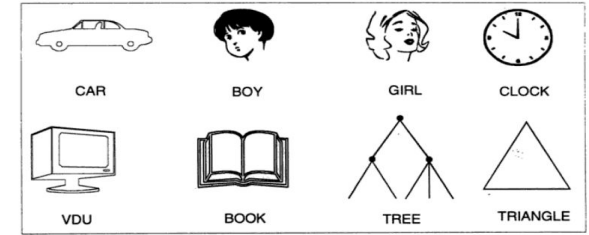
OOP: Variations

- object-oriented
 - objects + classes + inheritance + polymorphism
 - This is recognized as true object-orientation
 - examples: Simula, Smalltalk, Eiffel, Python, Ruby, Java, C++, C#, etc...

Pillars of OOP

- Classes
- Objects
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism(static and dynamic)

Objects



- Objects are basic run time entities
- **state**_(attributes/datastructure),
- **behaviour**_(methods or functions),
- **identity.**
- Object interact with other objects thro messages
- Not necessary to know the details of data or code
- Type of message is accepted and gets a response
- In OOPS -Think in terms of objects !!!!!!!!!!!



www.colorbox.com



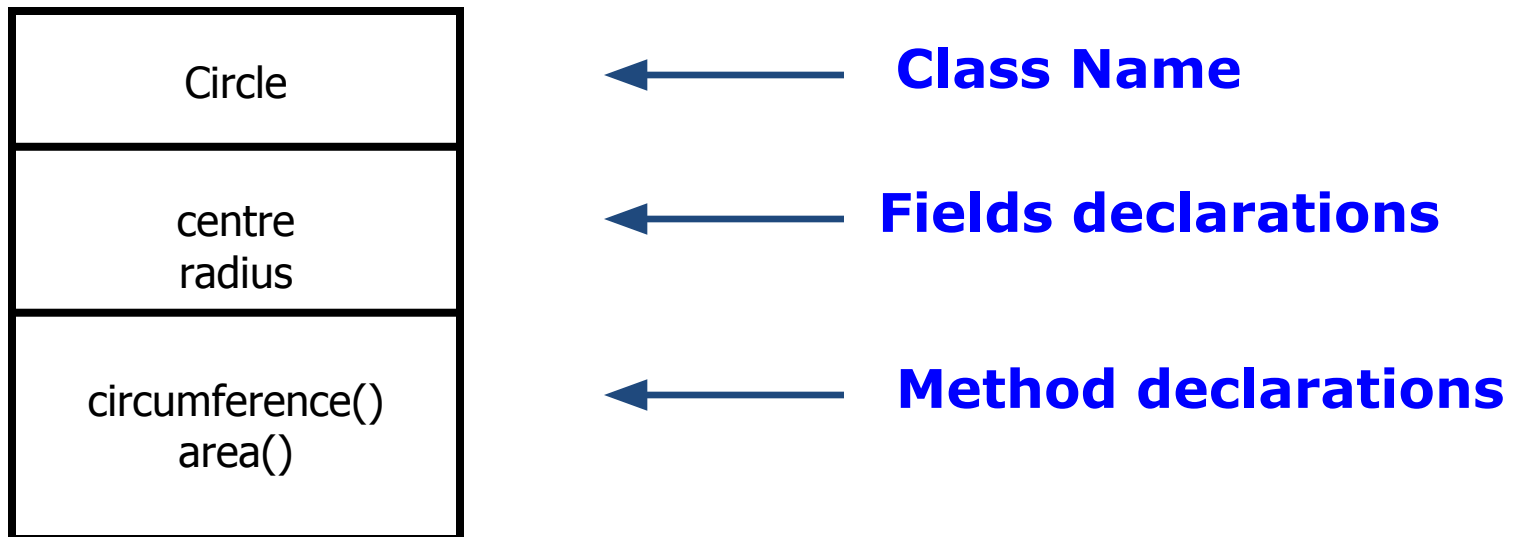
Image
s:
James
Tam

Class

- A class is a pattern, a blue print, or a template for a category of structurally identical entities
 - Created entities are called objects or instances
 - Class is like instance factory
 - Static entity
- A class has three components
 - Name
 - Attributes (also termed as variables, fields, or properties)
 - Methods (also termed as operations, features, behavior, functions)

Classes

- A *class* is a collection of *fields* (data) and *methods* (procedure or function) that operate on that data.



Classes/Objects

- Each class of object includes **descriptive data**.
 - Example (animals):
 - Species
 - Color
 - Length/height
 - Weight
 - Etc.
- Also each class of object has an **associated set of actions**
 - Example (animals):
 - Sleeping
 - Eating
 - Excreting
 - Etc.

1.Encapsulation

- The wrapping up of data and functions into a single unit
- Data can be accessed only through methods/
public interface for the clients
- Insulation of data from direct access by the program is called data hiding or information hiding(Access Modifiers)
- The unit of encapsulation in an O-O PL is a **class**

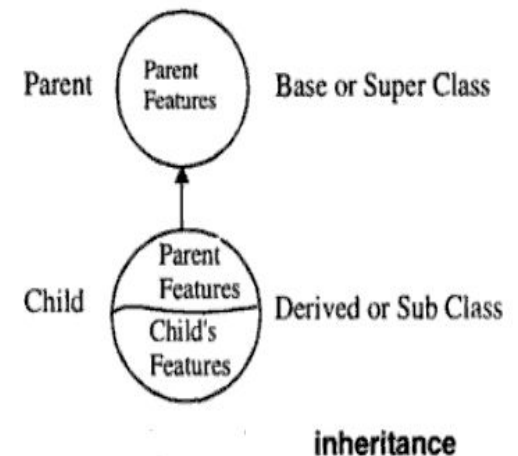
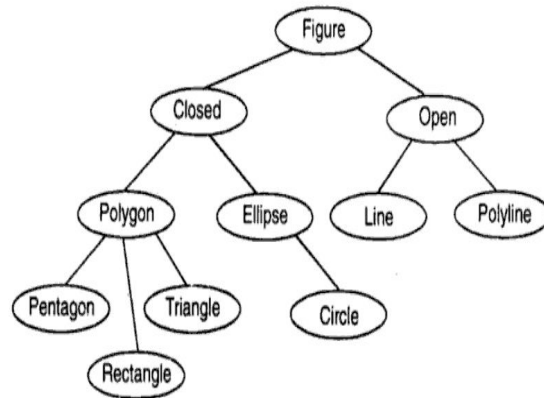
2.ABSTRACTION

- The act of representing essential features without including the background details or explanations
- Class uses the concept of abstraction
- Abstract classes and Interfaces
- In interfaces, only the methods are exposed to the end-user.

3. Inheritance

The process by which one object acquires the properties of the Other objects

Heirarchical classification reusability



4. Polymorphism

- Polymorphism is the ability of
 - objects belonging to different types
 - respond to method calls of the same name
 - each one according to an appropriate type-specific behavior.

Defining A Java Class,Fields and Methods

```
--  
<modifier>* class <class_name> {  
    <attribute_declaration>*  
    <constructor_declaration>*  
    <method_declaration>*  
}
```

```
<modifier>* <type> <name> [ = <initial_value>];
```

```
<modifier>* <return_type> <name> ( <argument>* ) {  
    <statement>*  
}
```

Example:

```
public class Person
{
    private int age; // Attribute
    public Person() { // Method
        age = in.nextInt();
    }
    public void sayAge() { // Method

        System.out.println("My age is " +
age);
    }
}
```

Example Exercise: Basic Real-World Alarm Clock/Pen

- What descriptive data is needed?
- What are the possible set of actions?

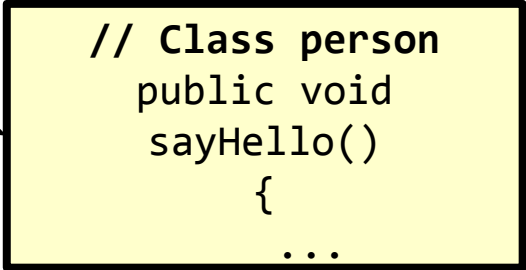


The First Object-Oriented Example

- Program design: each class definition (e.g., `public class <class name>`) must occur its own “dot-java” file).
- Example program consists of two files in the same directory:
 - (From now on your programs must be laid out in a similar fashion):
 - `Driver.java`
 - `Person.java`

The Driver Class

```
public class Driver
{
    public static void main(String [] args)
    {
        Person aPerson = new Person();
        aPerson.sayHello();
    }
}
```



```
// Class person
public void
sayHello()
{
    ...
}
```

```
I don't wanna say hello.
```

Class Person

```
class Person
{
    public void sayHello()
    {
        System.out.println("I don't wanna say hello.");
    }
}
```

main() Method

- Language requirement: There must be a `main()` method - or equivalent – to determine the starting execution point.
- Style requirement: the name of the class that contains `main()` is often referred to as the “Driver” class.
 - Makes it easy to identify the starting execution point in a big program.
- Do not instantiate instances of the `Driver`¹
- For now avoid:
 - Defining attributes for the `Driver`¹
 - Defining methods for the `Driver` (other than the `main()` method)¹

Write a Simple class for bulb.



A bulb:

1. It's a real-world thing. object
2. Can be switched on to generate light and switched off. methods
3. It has real features like the glass covering, filament and holder.
4. It also has conceptual features like power. member variables
5. A bulb manufacturing factory produces many bulbs based on a basic description / pattern of what a bulb is. class

Classes

- The basic syntax for a class definition:

```
Modifier class ClassName [extends  
  SuperClassName]  
{  
    [fields declaration]  
    [methods declaration]  
}
```

| Members of JAVA | Private | Default | Protected | Public |
|-----------------|---------|---------|-----------|--------|
| Class | No | Yes | No | Yes |
| Variable | Yes | Yes | Yes | Yes |
| Method | Yes | Yes | Yes | Yes |
| Constructor | Yes | Yes | Yes | Yes |
| interface | No | Yes | No | Yes |

Access modifiers in java

| Visibility | Public | Protected | Default | Private |
|---|--------|---------------------------------|---------|---------|
| From the same class | Yes | Yes | Yes | Yes |
| From any class in the same package | Yes | Yes | Yes | No |
| From a subclass in the same package | Yes | Yes | Yes | No |
| From a subclass outside the same package | Yes | Yes, <i>through inheritance</i> | No | No |
| From any non-subclass class outside the package | Yes | No | No | No |

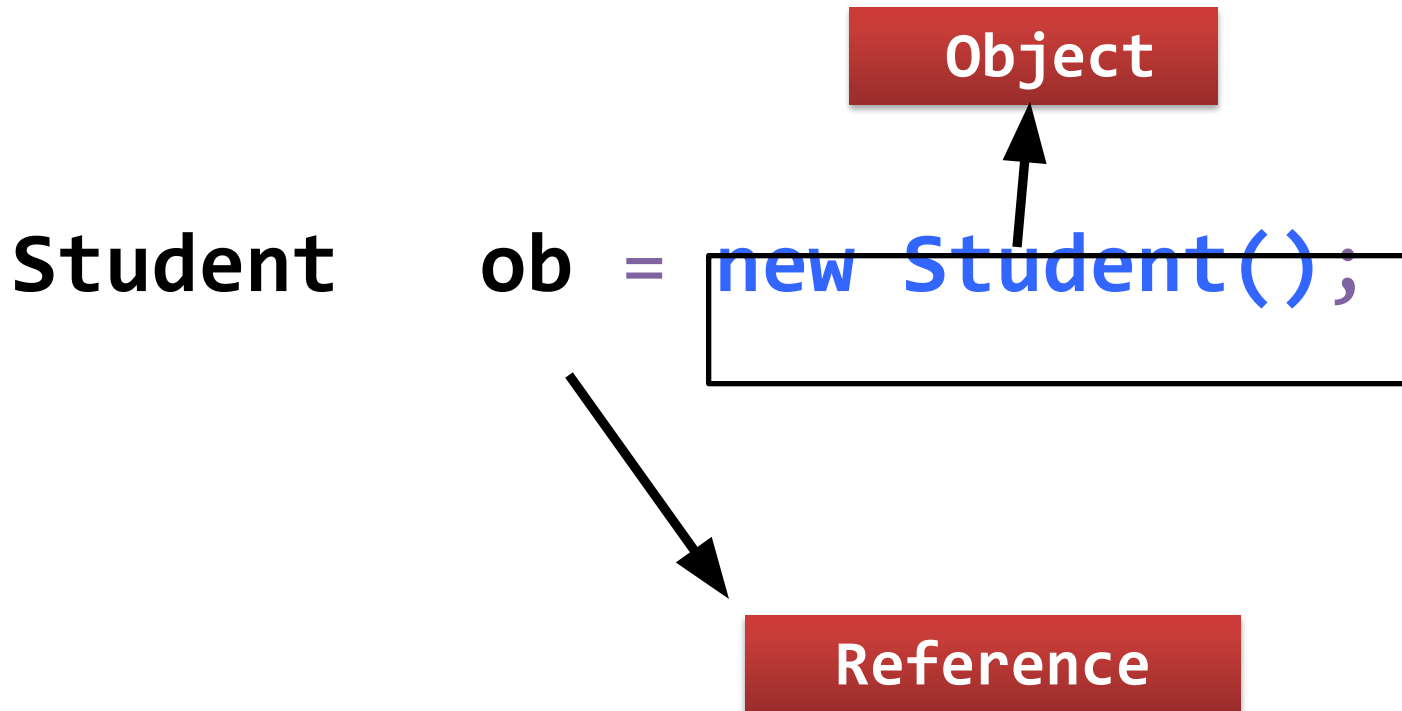
Non Access Modifier

- They are used with classes, methods, variables, constructors etc to provide information about their behavior to JVM.
- static
- final
- Abstract
- synchronized
- transient
- volatile
- Native
- strictfp

Class Modifiers

- Class modifiers include the following:
 - **default**/*<none>* - When no modifier is present, by default the class is accessible by all the classes within the same package.
 - **public** - A public class is accessible by any class.
 - **abstract** - An abstract class contains abstract methods.
 - **final** - A final class may not be extended, that is, have subclasses.
 - Cannot have private or protected(inner class)
 - A single Java file can only contain **one class that is declared public.**

Object creation in Java



Class

```
class Student
{
    int id;
    void display()
    {
        cout<<id;
    }
}
```



Variables and Methods

Types of Variables and Methods

- **Local variable.**
- **Instance variable and Instance methods (**non-static**).**
- **Class variable and Class methods (**static**).**

Local Variables

- Declared inside constructor, methods & blocks
- Garbage value
- Scope-within the block
- No modifier
- Memory allotted in stack.

Static variable and Instance Variable

```
class Demo
{
    static int a;
    int b;
}
```

Instance Vs Static Variables

- **Instance** variables : One copy per **object**.
Every object has its own instance variable.
 - E.g :a
- **Static** variables : One copy per **class**.
 - E.g. b

Static Methods

- Java supports definition of **global methods** and variables that can be accessed without creating objects of a class. Such members are called Static members.
- A class can have methods that are defined as static (e.g., main method).
- Static methods can be accessed without using objects. Also, there is NO need to create objects.
- They are prefixed with keyword “static”
- Static methods are generally used to group related library functions that don't depend on data members of its class. For example, **Math library functions**.

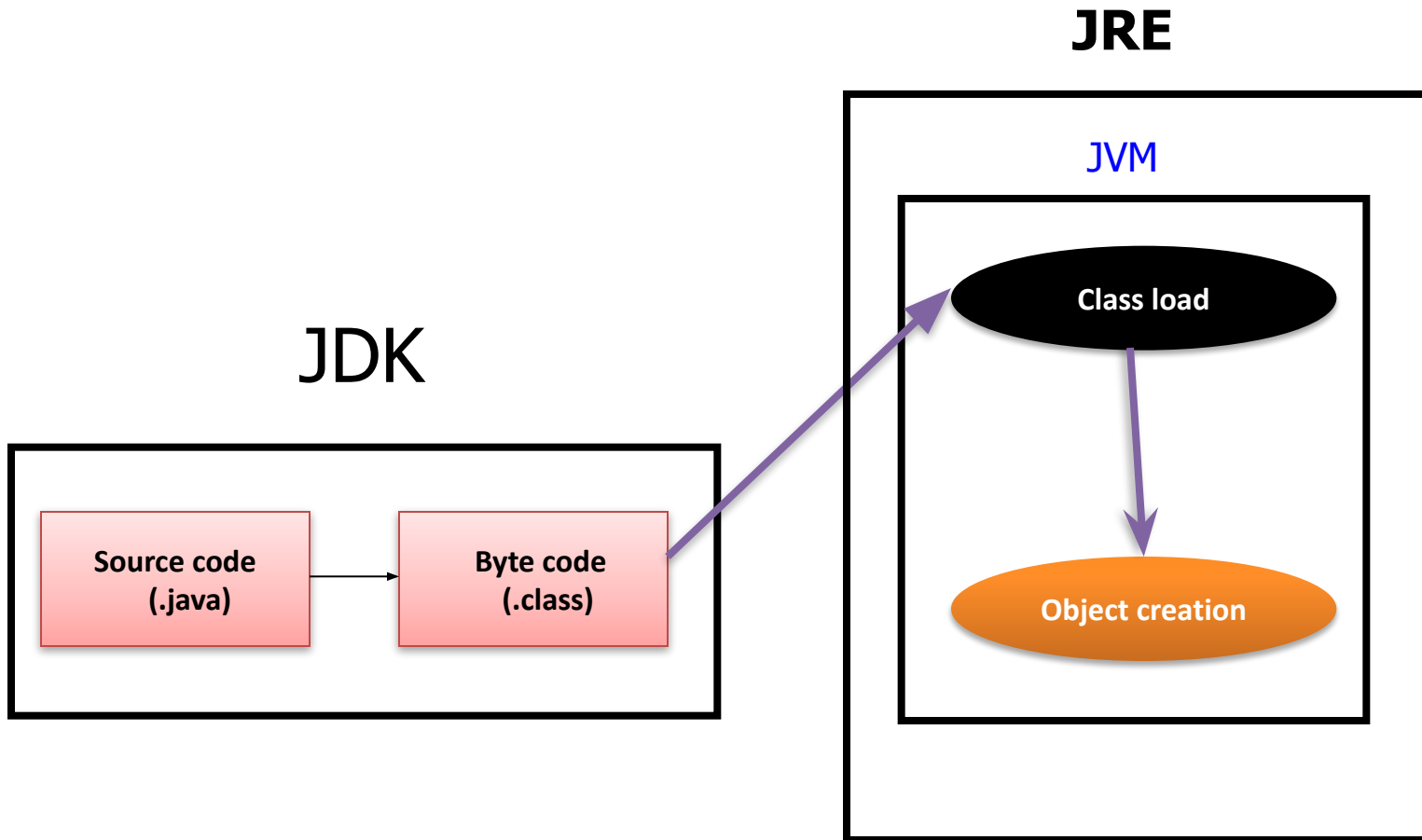
Static methods restrictions

- They can only call other static methods.
- They can only access static data.
- They cannot refer to “this” or “super” (more later) in anyway.

class A

```
{ int a=40;//non static  
  public static void main(String args[])  
  {   System.out.println(a);  } //error  
}
```

Platform Independence



Accessing Static and Instance Variable

```
class Demo
```

```
{
```

```
    static int a;
```

```
        int b;
```

```
}
```

```
Demo.a = 5000;
```

```
Demo obj=new Demo();
```

```
obj.b=1000;
```

Instance variable and method

```
class Demo
{
    int a;
    void sum()
    {
        cout<<a;
    }
}
```

```
Demo o1=new Demo();
```

```
o1.a=1000;
```

```
o1.sum();
```

```
Demo o2=new Demo();
```

```
o2.a=3000;
```

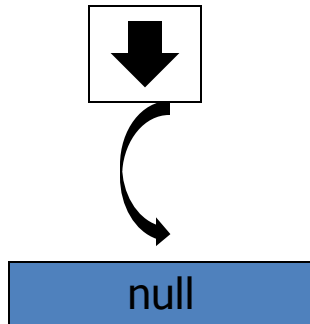
```
o2.sum();
```

sent 'message' to O2 object

Class of Demo contd..

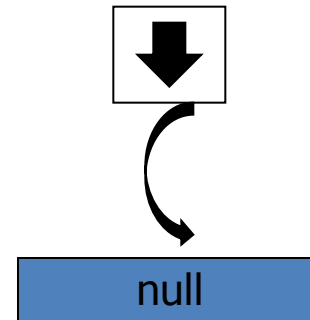
- Demo O1,O2;

O1



Points to nothing (Null Reference)

O2

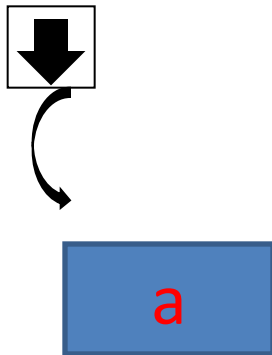


Points to nothing (Null Reference)

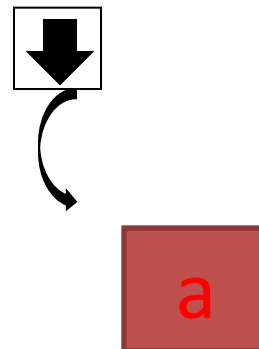
Creating objects of a class

- Objects are created dynamically using the *new* keyword.
- O1 and O2 refer to Demo objects

Demo o1=new Demo();



Demo o2=new Demo();



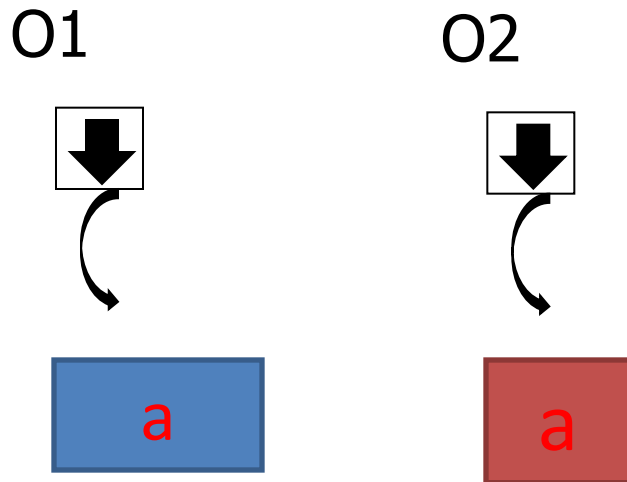
Creating objects of a class

```
Demo o1=new Demo();
```

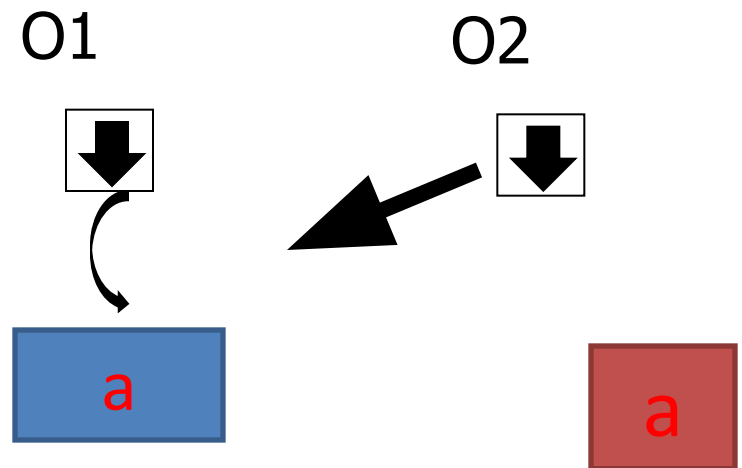
```
Demo o2=new Demo();
```

```
o2=o1;
```


Before Assignment



Before Assignment



Automatic garbage collection

- The object  does not have a reference and cannot be used in future.
- The object becomes a candidate for automatic **garbage collection**.
- Java automatically collects garbage periodically and releases the memory used to be used in the future.

Class variable and method

```
class Demo
{
    static int a;
    static void sum()
    {
        cout<<a;
    }
}
```

```
Demo.a = 5000;
Demo.sum();
```

All classes extends object class

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

```
class Demo extends Object
```

```
{
```

```
    Demo()
```

```
    {
```

```
    }
```

```
}
```

ACCESS AND NON ACCESS MODIFIERS

Visibility/Access Modifier

- Properties
 - Controls access to class members
 - Applied to instance variables & methods
- Four types of access in Java
 - Public Most visible
 - Protected
 - Package(private)
 - Default if no modifier specified
 - Private Least visible



- **Access Modifiers for Top-level Classes & Interfaces:**

- Access modifiers-public, default,
- Non access modifiers- abstract, final, strictfp

Access Modifiers for Members:

public Members, protected Members, default Members, private Members, static Members, final Members, abstract Methods, synchronized Methods, native Methods, transient Fields, volatile Fields.

Class Modifiers

- Syntax for class modifiers:

```
[ClassModifier] class ClassName {...}
```

- Example:

```
public class Employee {...}
```

Method Modifiers

- Method Modifiers include the following:
 - *<none>* - When no modifier is present, by default the method is accessible by all the classes within the same package.
 - *public* - A public method is accessible by any class.(outside package)
 - *protected* -A protected method is accessible by the class itself, all its subclasses.(outside package)
 - *private* - A private method is accessible only by the class itself.

Method Modifiers

- **static** - accesses only static fields.
- **final** - may not be overridden in subclasses.
- **abstract** - defers implementation to its subclasses.
- **synchronized** - atomic in a multithreaded environment
- **native** - compiled to machine code by Assembler, C or C++.

Method Modifiers

- Syntax for method modifiers:

```
[MethodModifiers] ReturnType MethodName  
    ([ParameterList) {}
```

- Example:

```
private void GetEmployee(int ID) { }
```

Constructor

- Constructor is a special method that gets invoked “automatically” at the time of object creation.
- Constructor is normally used for initializing objects with default values unless different values are supplied.
- Constructor has the same name as the class name.
- Constructor cannot return values.
- A class can have more than one constructor as long as they have different signature (i.e., different input arguments syntax).

Defining a Constructor

- Like any other method

```
public class ClassName {  
  
    // Data Fields...  
  
    // Constructor  
    public ClassName()  
    {  
        // Method Body Statements initialising Data Fields  
    }  
  
    //Methods to manipulate data fields  
}
```

- Invoking:
 - There is NO explicit invocation statement needed:
When the object creation statement is executed, the constructor method will be executed automatically.

Multiple Constructors--overloading

- Sometimes want to initialize in a number of different ways, depending on circumstance.
- This can be supported by having multiple constructors having different input arguments.
- Thread class has 8 types of constructors.

// Constructor Overloading

class Box

{

double width, height, depth;

// constructor used when all dimensions

// specified

Box(double w, double h, double d)

{

width = w;

height = h;

depth = d;

}

// constructor used when no dimensions

Box()

{

width = height = depth = 0;

}

Box(double len)

{

width = height = depth = len;

}

```
// Driver code
public class Test
{
    public static void main(String args[])
    {
        // create boxes using the various
        // constructors
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);
    }
}
```

Copy Constructor

- Java doesn't create a default copy constructor
- The user should provide it
- A copy constructor in a Java class is a constructor that **creates an object using another object of the same Java class.**

Shallow Copy

- `Complex(Complex c) {`
- `//Shallow copy`
- `System.out.println("Copy constructor called");`
- `re = c.re;`
- `im = c.im;`
- `}`

Deep Copy

```
public class Person
{
    private Name name;
    private Address address;
    public Person(Person otherPerson)
    {
        this.name = new Name(otherPerson.name);
        this.address = new Address(otherPerson.address);

    }
    [...]
}
```

Array of Objects

```
private static class Employee{  
    public String name;  
    public int    employeeld;  
  
    public Employee(String name, int employeeld){  
        this.name      = name;  
        this.employeeld = employeeld;  
    }  
}
```

Array of objects-Sort

- **Employee[] employeeArray = new Employee[3];**
- **employeeArray[0] = new Employee("Xander", 1);**
- **employeeArray[1] = new Employee("John" , 3);**
- **employeeArray[2] = new Employee("Anna" , 2);**
- **Remove Duplicates**

Binary search on employee objects

```
public static int binarySearch(employee [] emp, employee
    target)
{
    int low = 0; int high = employee.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        int comp = employee[mid].compareTo(target);
        if (comp == 0) {return mid; }
        else if (comp < 0) {low = mid + 1; }
        else {high = mid - 1; } } return -1; }
```

Array Implementation in C

```
char Store[MAX];  
int top = 0;  
void push(char x)  
{  
    if (top < MAX)  
        Store[top++] = x;  
    else  
        printf("full\n");  
}
```

```
char pop()  
{  
    if (top > 0)  
        return Store[--top];  
    else  
        printf("empty\n");  
}  
...
```

Using the Stack

```
void application()  
{  
    ...  
    push('x');  
    ...  
    result = pop();  
    ...  
}
```

Procedural Programming

- Focus is on writing good functions and procedures
 - use the most appropriate implementation and employ correct efficient algorithms
- Stack example (assume array implementation)
 - one source file
 - Store and top are global variables
 - stack and application functions defined at the same level (file)

Problems

- Application can alter implementation details
 - can directly manipulate top and Store from application()
 - integrity of stack not ensured
- Stack code and application code are not separated