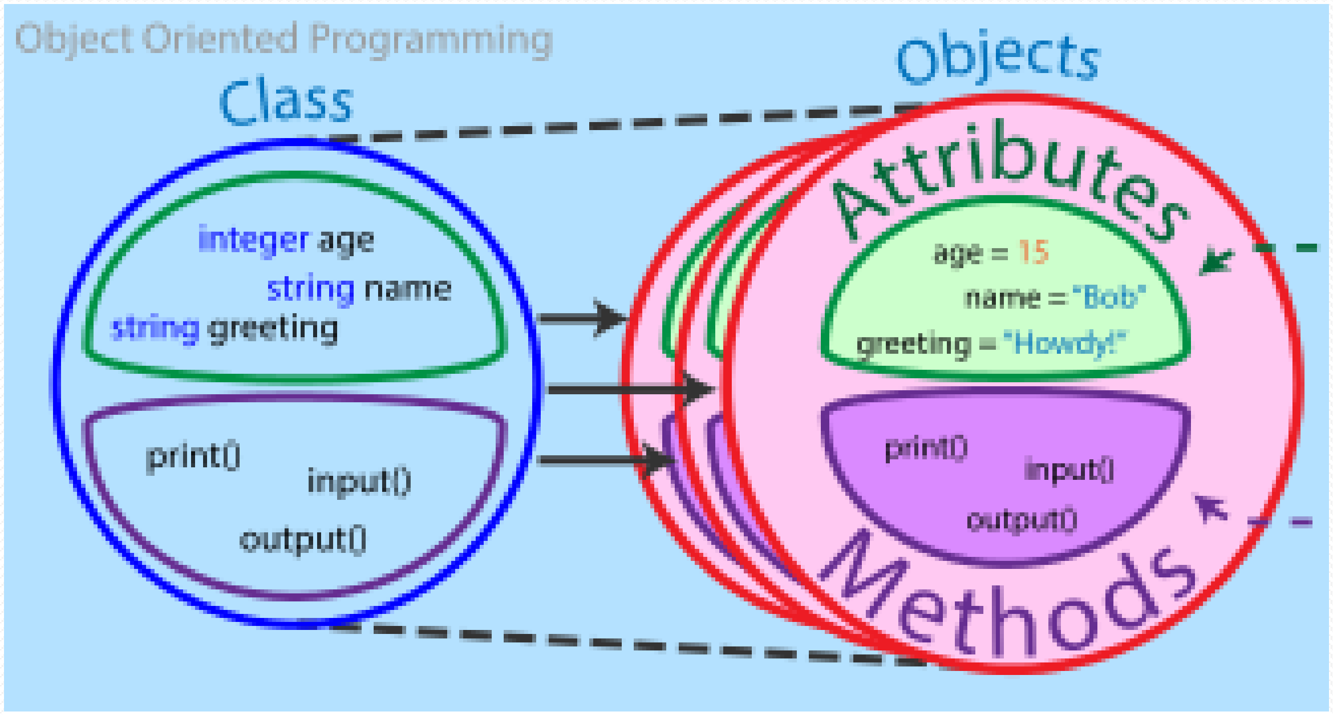
Object Oriented Programming

Module 3. Object Oriented Programming

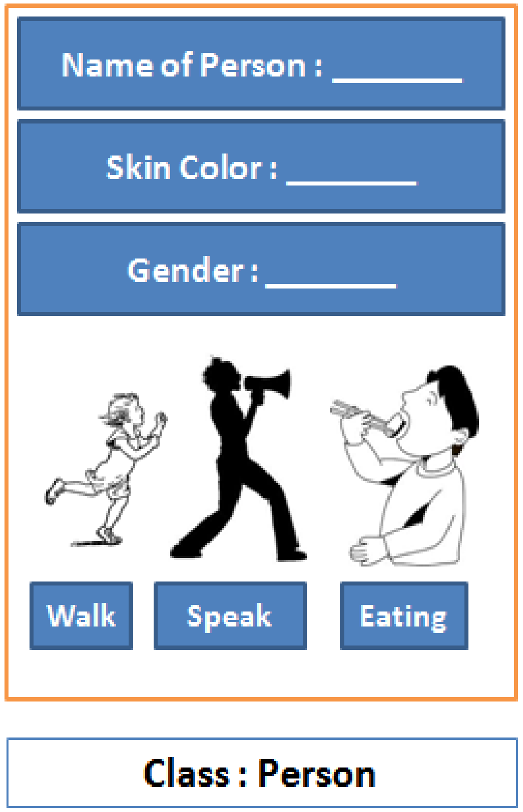


Elements of a Class

Elements of a Class

Reminder

- A class is a **blueprint** or prototype from which objects are created.
- A class is a classifier which describes a set of objects with characteristics and behaviour.
- Such descriptors are the elements (members) of a class and are so-called **properties and methods** respectively.



Elements of a Class: Attributes

- Class “member variables” are called **“attributes"**. You may also see them referred to using other terms such as “**properties**" or "**fields**“.

- An attribute may include an initialization, but this initialization must be a constant value (based on the type).

```
public int circleRadius = 15;
```

Elements of a Class: Methods

- First understand what a FUNCTION is!!
- **A Function** is a combination of instructions coupled together to achieve some result. It may take arguments and return result. If a function doesn't return a result it is usually called a procedure.
- **A Method** is a “Member Function”, they belongs to classes or objects and usually expresses the verbs of the objects/class.
- In short, a method is an **Action** defined by the class.

Elements of a Class: Methods

- Example:

```
/** Method that returns the minimum between two numbers
*/
public int getMinNumber(int n1, int n2) {
   int min;
   if (n1 > n2)
      min = n2;
   else
      min = n1;

   return min;
}
```

Elements of a Class: Methods

Reminder

- Methods implement the behaviour of objects.
- Methods have a consistent structure comprised of a **header** and a **body**.
- **Accessor methods** provide information about an object.
- **Mutator methods** alter the state of an object.
- Other sorts of methods accomplish a variety of tasks.

  Some methods can perform a unit of work without taking any information in or returning any information to the code that invoked it.

Elements of a Class: Methods

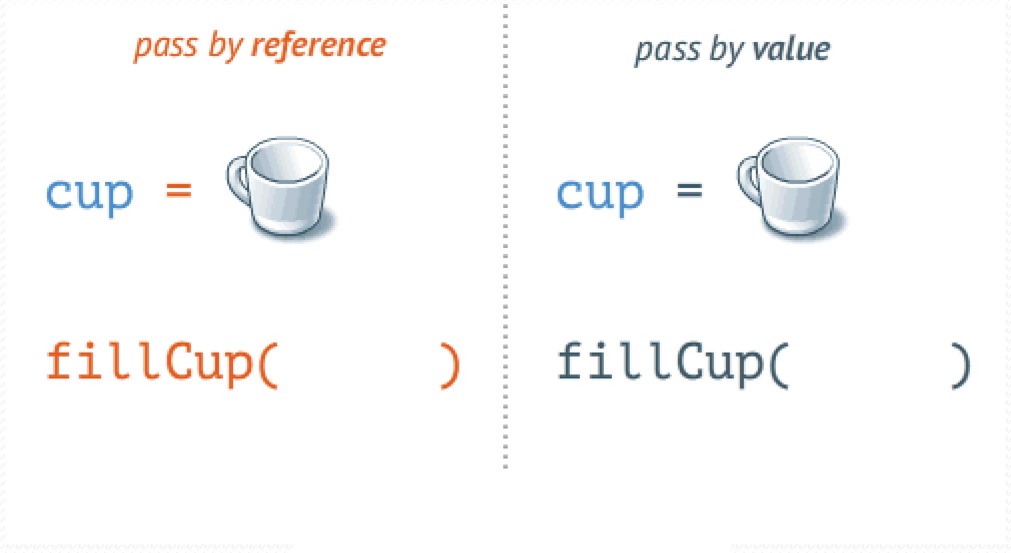
- The list of parameters is a sequence of:
  <type> <varName>
  separated by coma.
- For example:

```
public void sayHelloAndAge (String name, int age) {
    System.out.println(“Hello ” + name);
    System.out.println(“Your age is ” + age + “ years old”);
}
```

Elements of a Class: Methods

- In Java there are to ways of passing parameters, by **value** and by **reference**. By value you create a copy of the variable value, by reference you only pass the memory address where the value is stored.

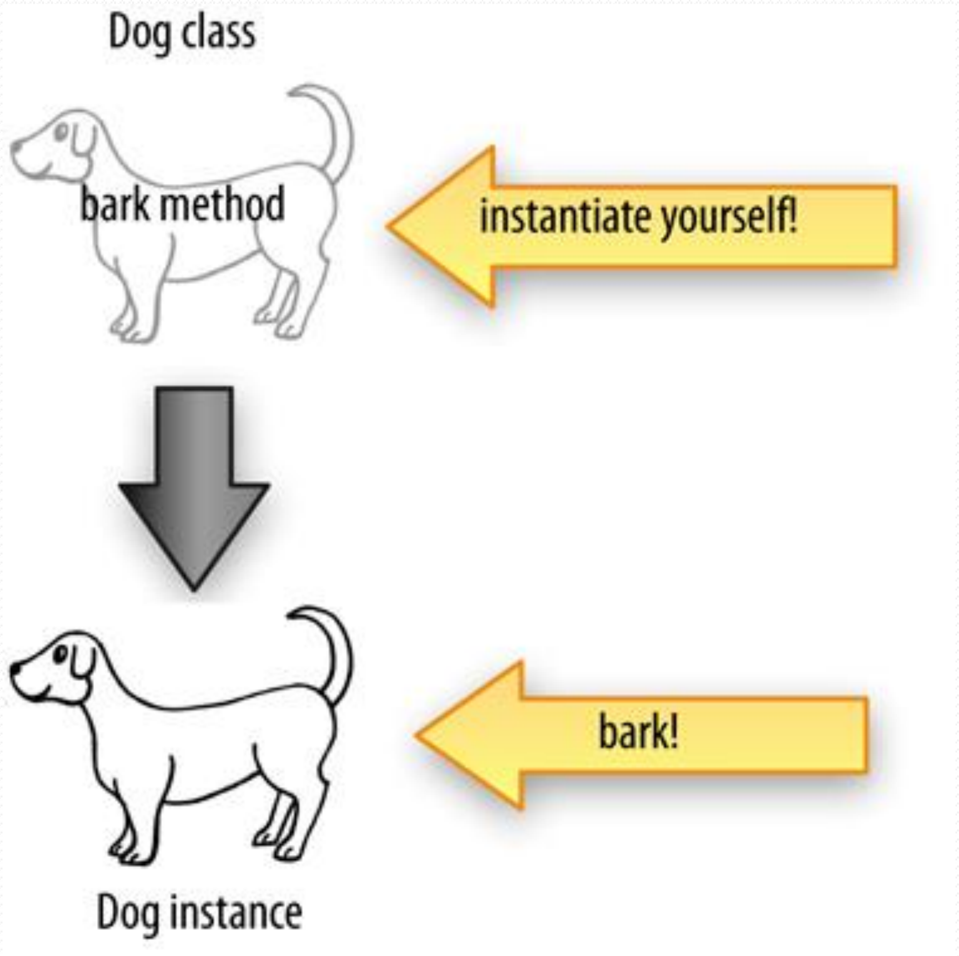
- All primitive types are passed by value and all class type are passed by reference.



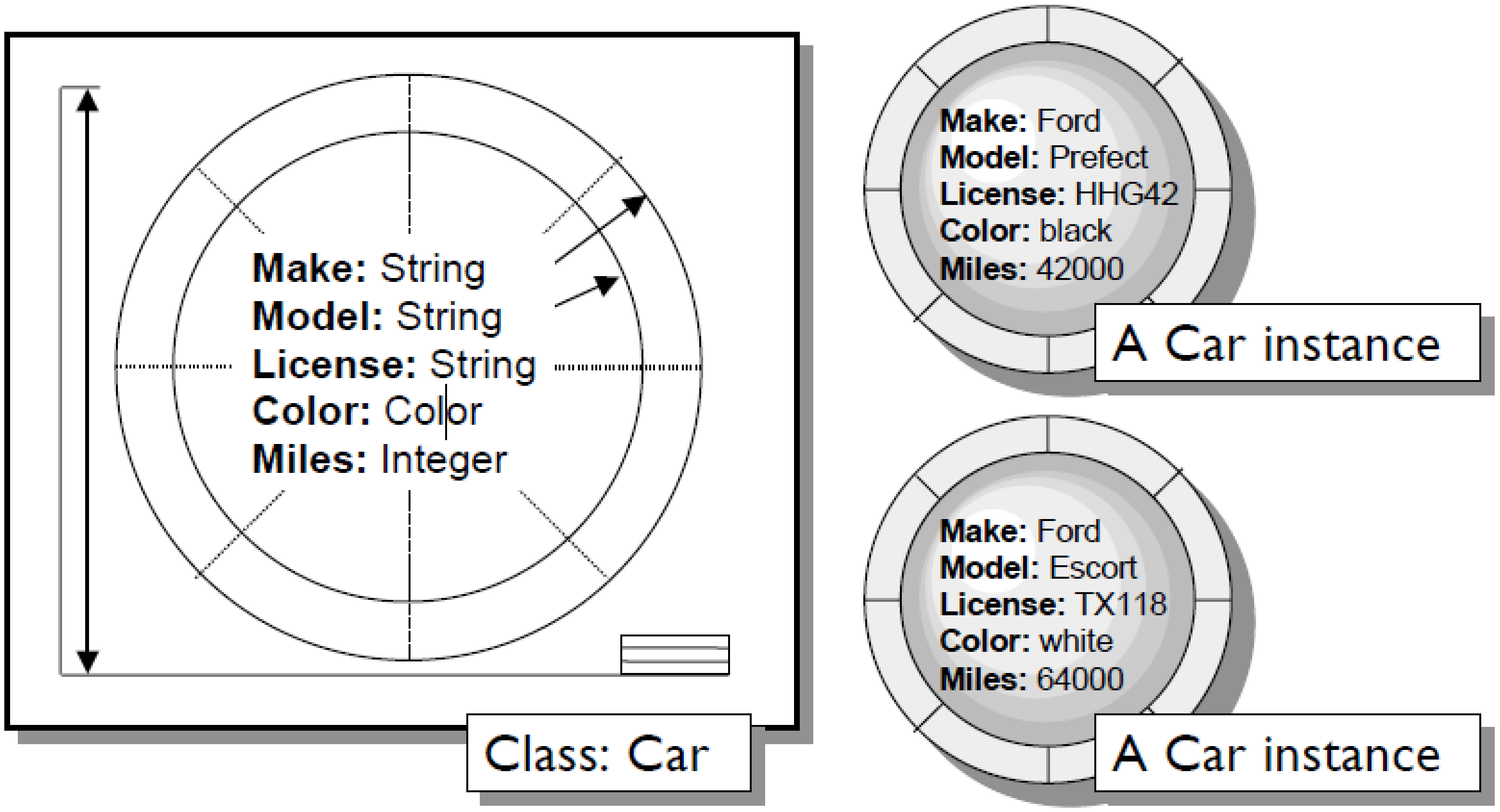
So… What Can Go Inside a Class?

```
Class A {
        int i;
        A(){}
        A(int a){…}
        public void foo(){…}
        public int getI(){…}
        public void setI(){…}
}
```

Classes and Instances



Classes and Instances



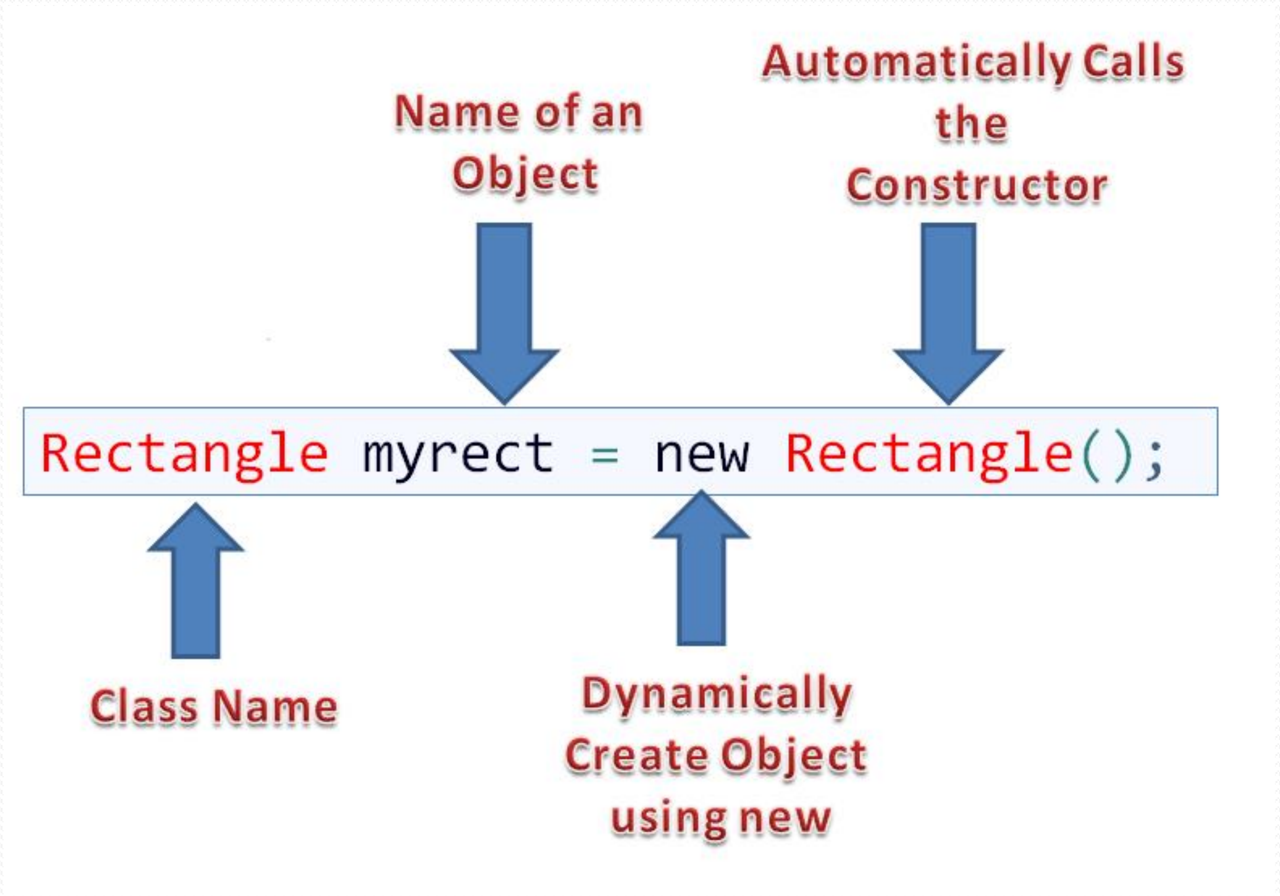
Instance Variables

- An **instance variable** (attribute) of an object is a piece of information attached to an **instance** (object).
  - The name of a Person object, the model and year of a Car object, etc.
- The instance variables that an object has are defined in the object's class:
  - An object can usually have many instance variables, of many different types.
- Assigning a new value to an instance variable of one object **does not affect** the instance variables of any other object.

Defining Objects

- When an object of a class is created, the space for all data members defined in the class is allocated in the memory according to their data types.
- An object is also known as **instance**.
- Defining an object is similar to defining a variable of any data type.
- The process of creating an object of a class is also called **instantiation**.
- Syntax: ClassName ObjectName;

Defining Objects: Initialization


Name of an Object
Automatically Calls the Constructor
Class Name
Dynamically Create Object using new

```
Rectangle myrect = new Rectangle();
```

Defining Objects: Initialization

- **Constructors** ensure correct initialization of all data. They are automatically called at the time of object creation.
- **Destructors** on the other hand ensure the de allocation of resources before an object dies or goes out of scope.

Defining Objects: Initialization

- Example:

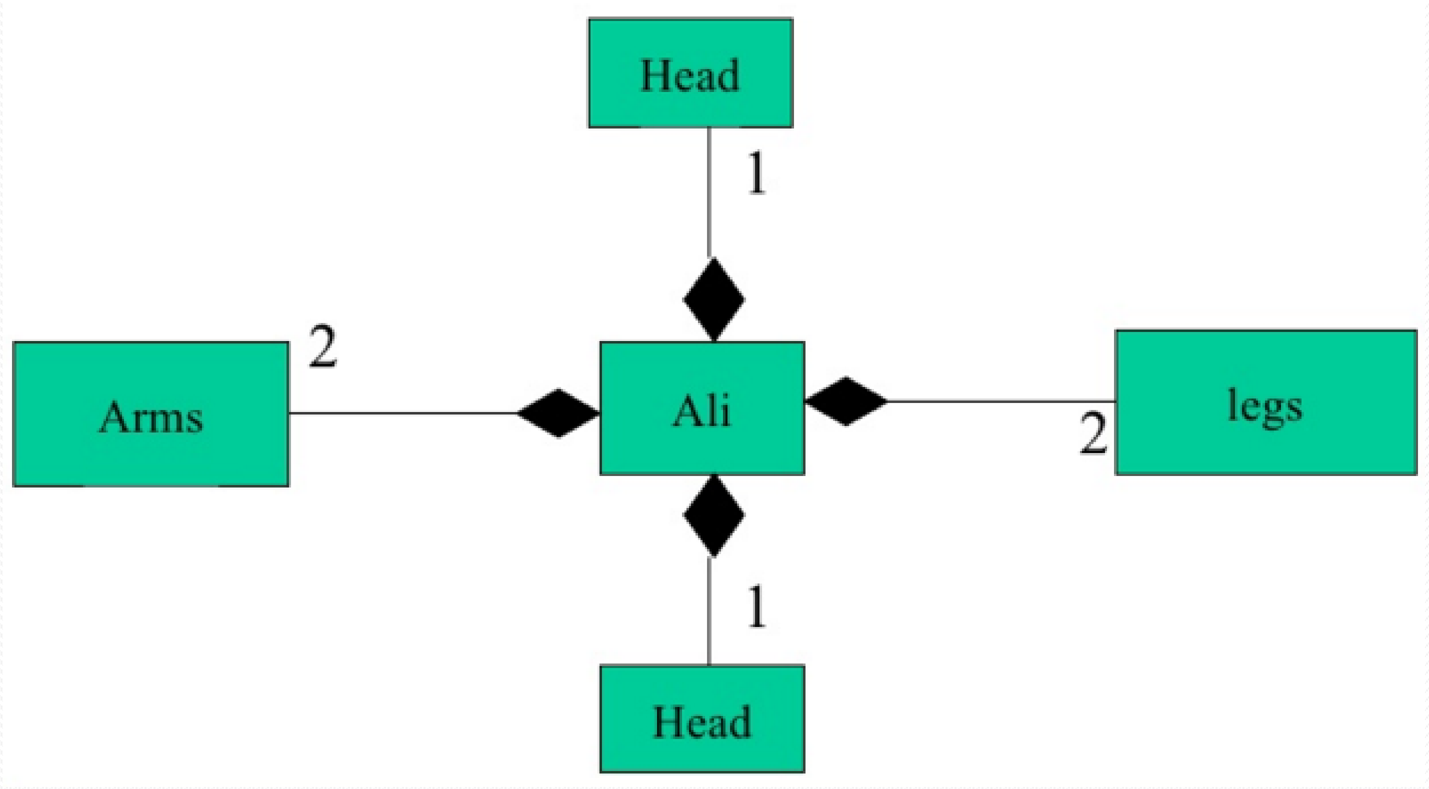
```
public class Puppy {
        public Puppy(String name) {
        /* This constructor has one parameter, name. */
        System.out.println("Passed Name is :" + name );
        }
        public static void main(String []args) {
        /* Following statement would create an object myPuppy */
        Puppy myPuppy = new Puppy( “Gober" );
        }
}
```

```
Output: Passed Name is :Gober
```

Defining Objects: Executing Methods

- An object of a particular class contains all data members (attributes) as well as methods defined in that class.
- The data members contains the value related to the object.
- The methods are used to manipulate data members.
- The methods can be executed only after creating an object.
- Syntax: ObjectName.method();

Relationships With Other Classes



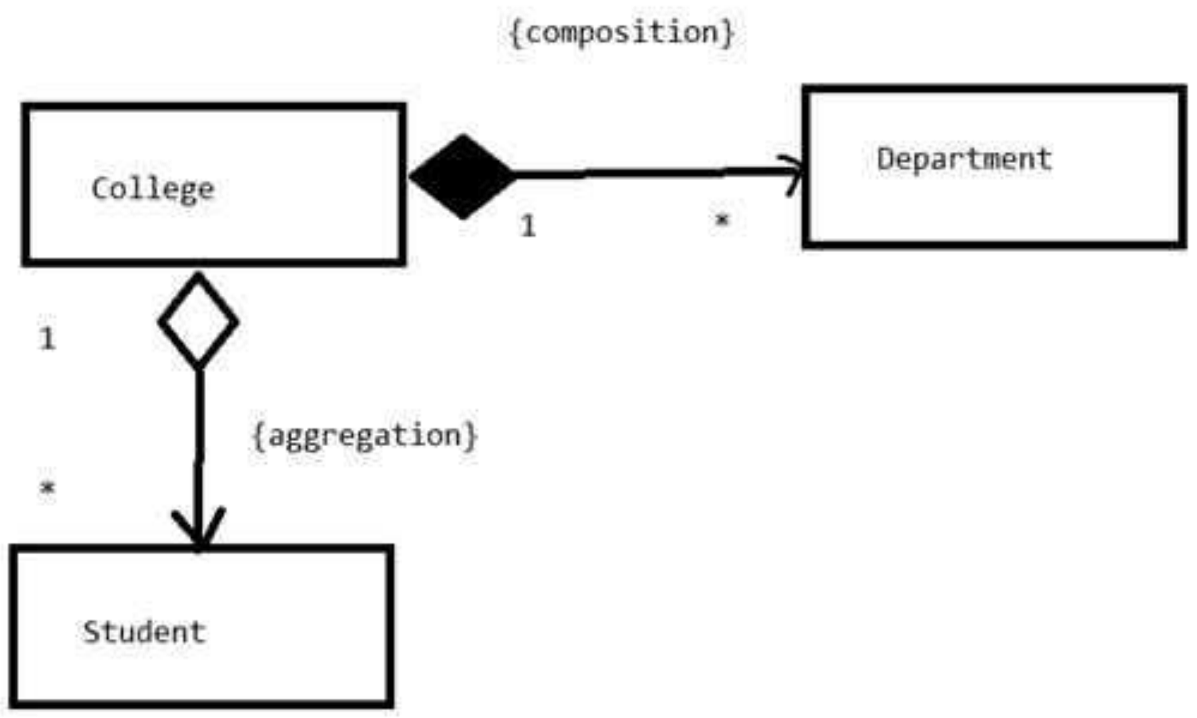
Relationships

- There are two kinds of relationships among classes:
  - Generalization (inheritance)
  - Association
- Associations can be further classified as:
  - Aggregation
  - Composition
  - Dependency/usage

Relationships

- Association means **HAS-A** relationship.
- Both composition and aggregation are associations.
- Aggregation -> **Weak** Has-A relationship.
- Composition ->**Strong** Has-A relationship.

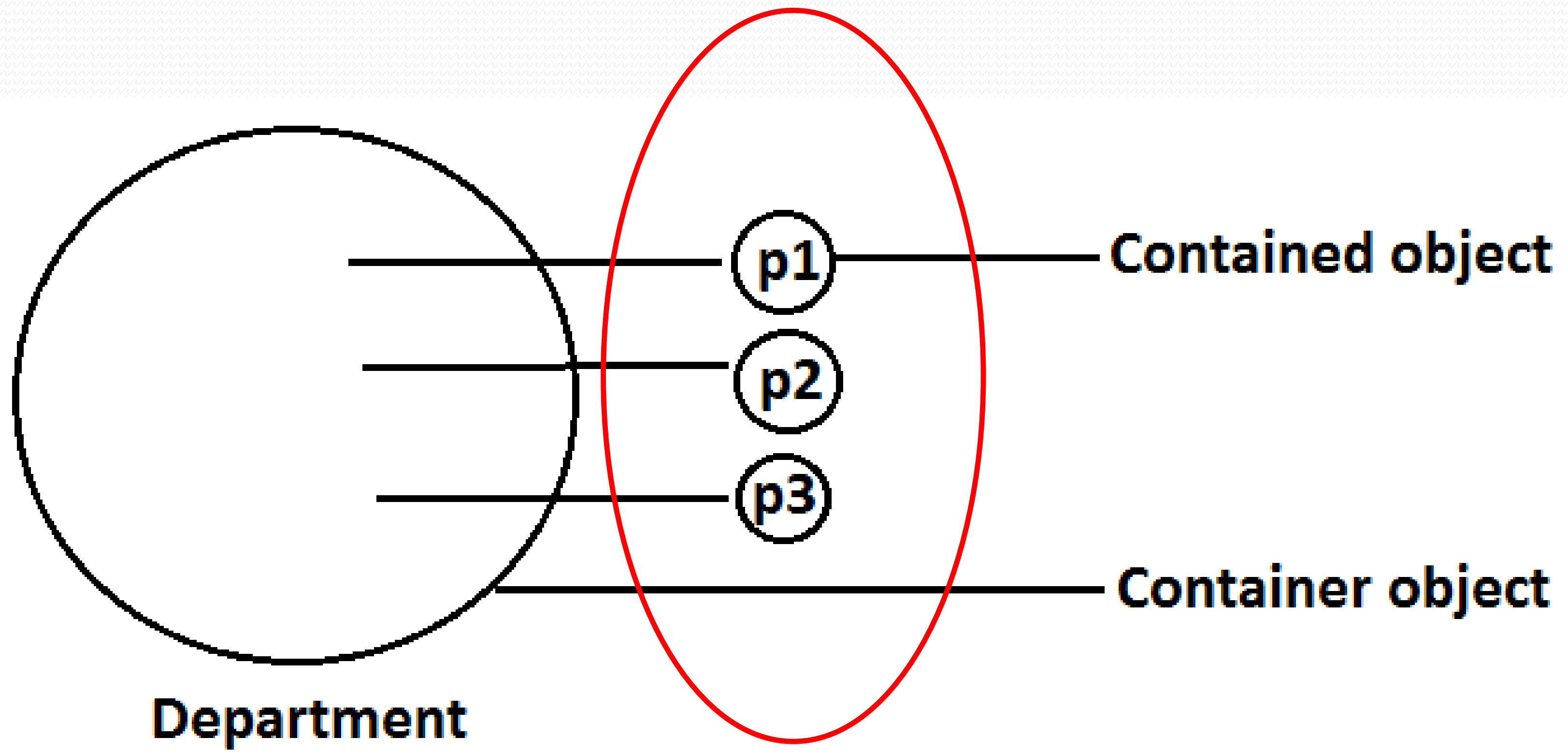
Aggregation Relationships



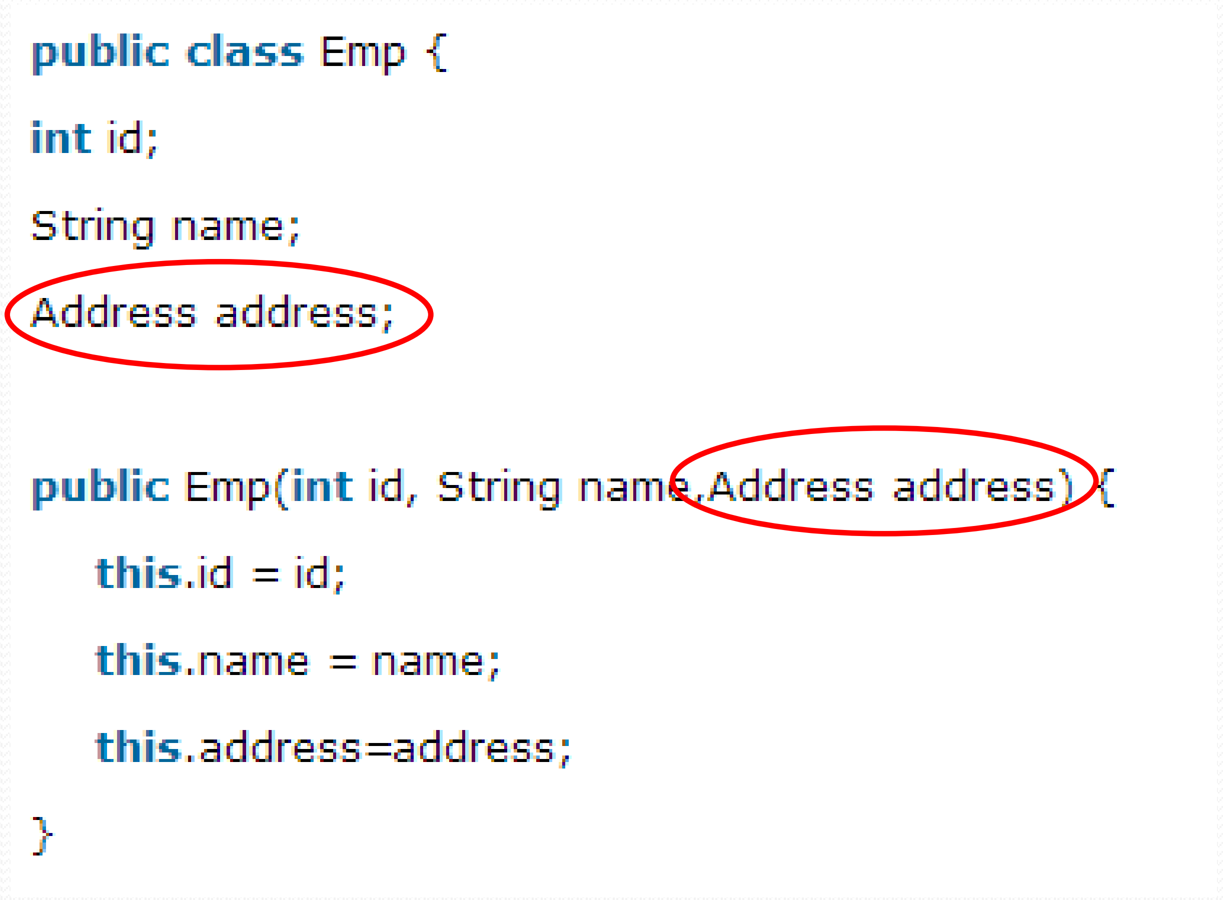
Aggregation Relationships

- An **aggregate** is an object that is made up of other objects.
- Therefore aggregation is a **Has-A** relationship.
- An aggregate object contains references to other objects as instance data.
- The aggregate object is defined in part by the objects that make it up.

Aggregation Relationships



Aggregation in Java



```
public class Emp {

int id;

String name;

Address address;


public Emp(int id, String name,Address address) {

    this.id = id;

    this.name = name;

    this.address=address;

}
```

The This Reference

- The **this** reference allows an object to refer to itself.
- The this reference, used inside a method, refers to the object through which the method is being executed.
- The this reference can be used to **distinguish** the instance variables of a class from corresponding method parameters with the same names.

The This Reference

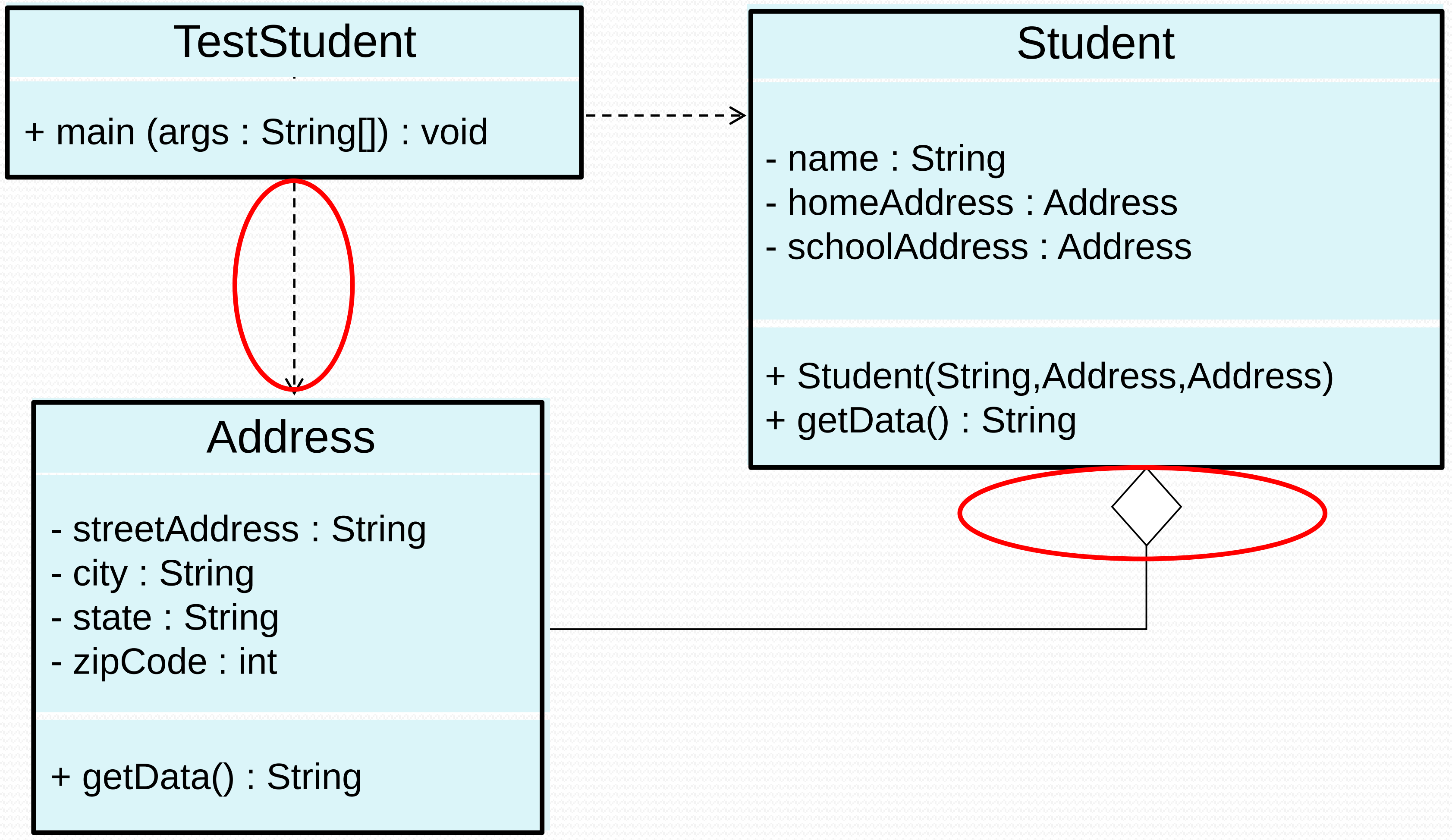
- For example:

```
public class Account{
    private String name;
    private long acctNumber;
    private double balance;
    public Account (String name, long acctNumber, double balance)
    {
        this.name = name;
        this.acctNumber = acctNumber;
        this.balance = balance;
    }
}
```

Aggregation in Java

- Student object is made of Address objects.
- Each student has two addresses:
  - School address and home address.
- An aggregation association is shown in a UML class diagram using an open diamond at the aggregate end.

Aggregation in Java



Aggregation in Java

```
public class Address {
    private String streetAddress;
    private String city;
    private String state;
    private long  zipCode;
    Address(String theStreet, String theCity, String theState, long theCode)
    {
        this.streetAddress=theStreet;
        this.city =theCity;
        this.state = theState;
        this.zipCode = theCode;
    }
    //..getters & setters
}
```

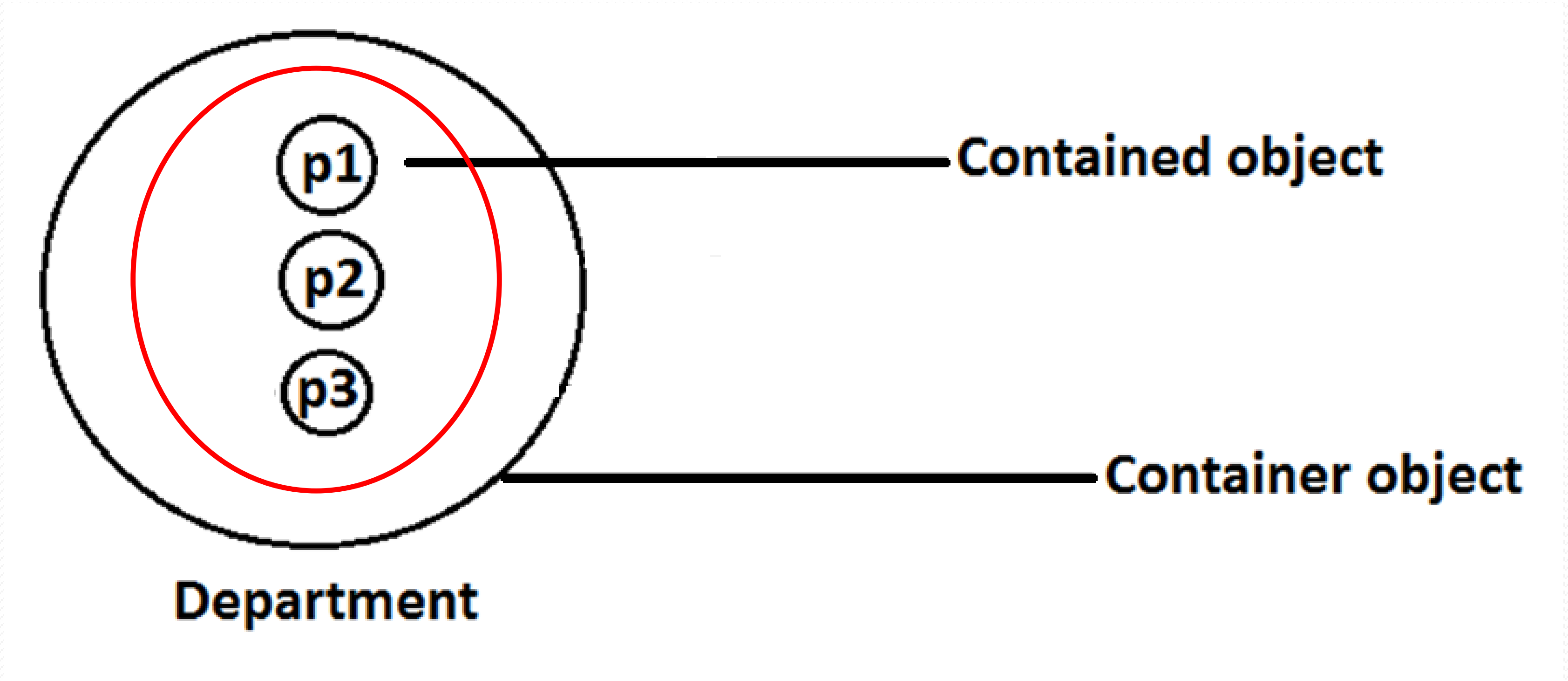
```
public class Student{
        private String name;
        private Address homeAddress;
        private Address schoolAddress;

        Student(String theName, Address theHomeAddr, Address theSchoolAdd){
             this.name=theName;
             this.homeAddress = theHomeAddr;
             this.schoolAddress = theSchoolAddr;
        }
        //..getters & setters
}

public class TestStudent(){
        public static void main(String args[]){
                Address homeAdd = new Address("street", "city", "state", 12345);
                Address schoolAdd= new Address("Epigmenio G.","Queretaro","Queretaro", 33333);
                Student obj = new Student("Silvana", homeAdd. schoolAdd);
                obj.toString();

                //use setters and change the values
        }
}
```

Composition Relationships

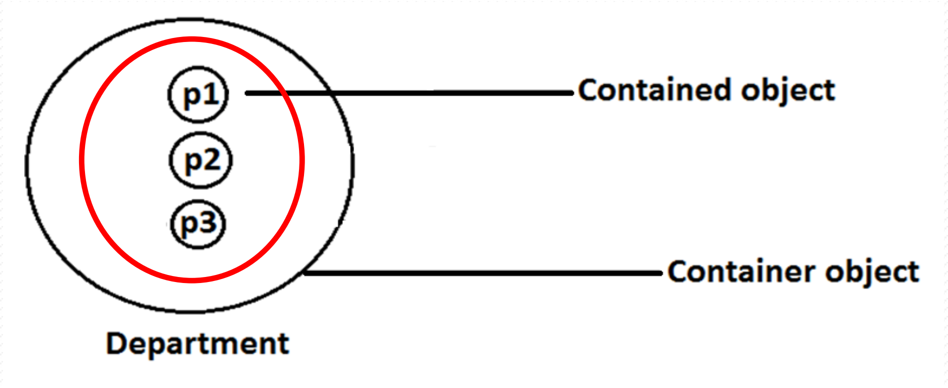
- Composition is a strong relationship because:
  - Composed objects become part of the composer.
  - Composed objects can’t live independently.

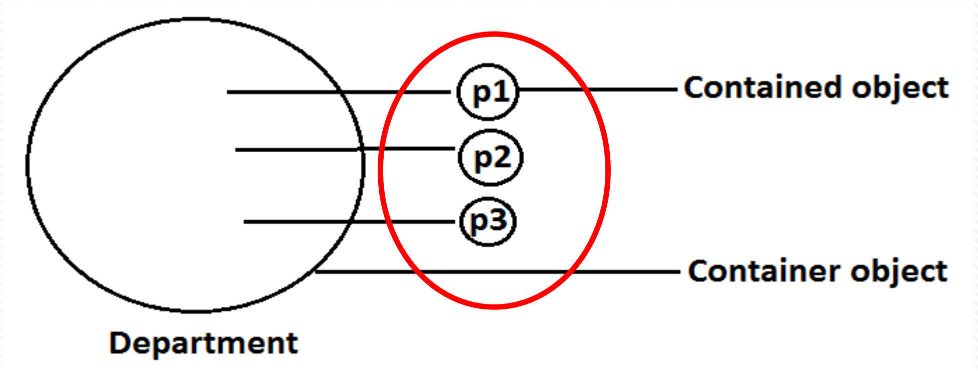
Classes **define** types. A class name can be used as the type for a variable. Variables that have class as their type can store objects of that class.

Composition Relationships



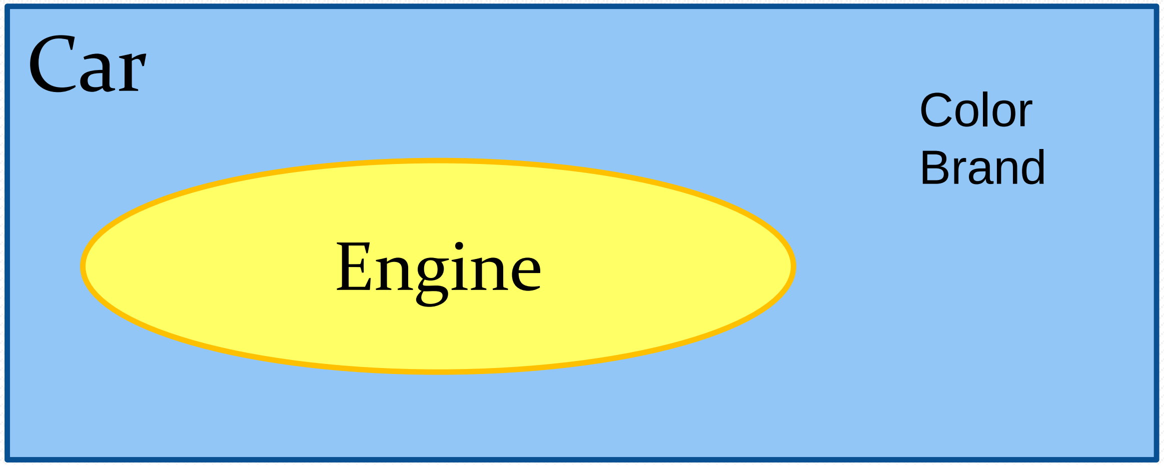
Composition Vs. Aggregation





Composition in Java

- **HAS A** relationship between objects.
- Implemented using instance variables.
- Code reuse.
- Hide visibility to client classes.

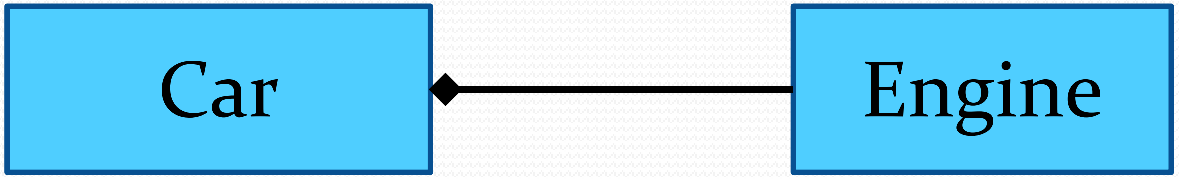


Composition in Java

- For example: Car has Engine.

```
class Car  {
    String color;
    String brand;
    Engine carEngine;

     public Car(){
       carEngine = new Engine();
       ...
      }
    ...
    }
```

```
class Engine  {
    public void start(){
        //start the engine;
        }
    }
```



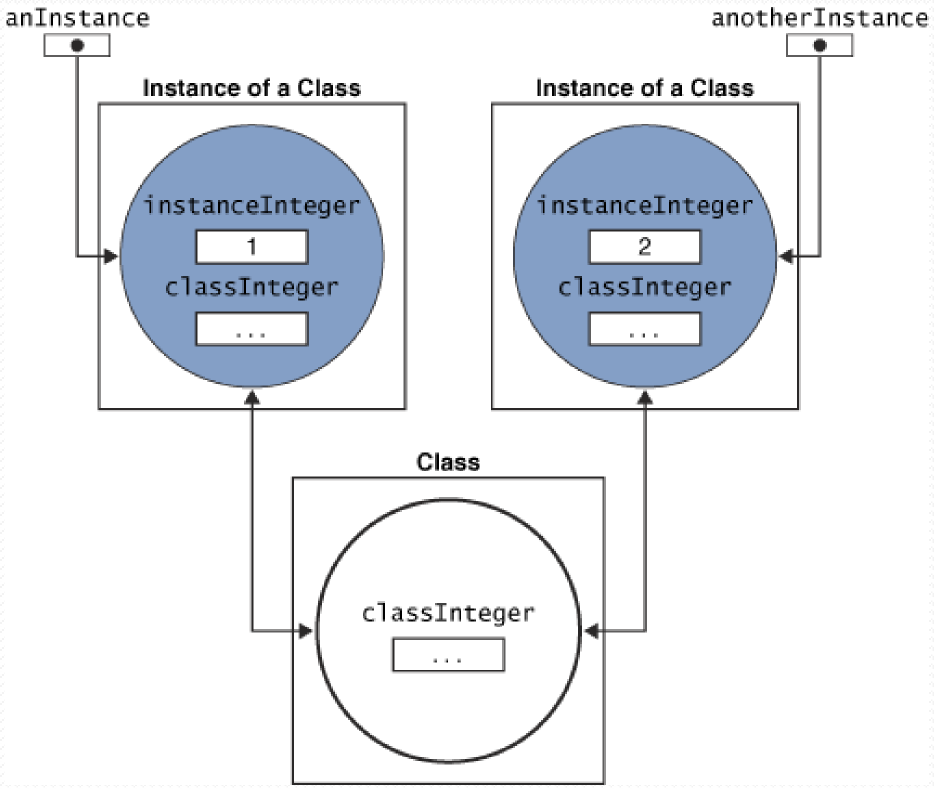
Self-review topics for next class

Book-Java Software Solutions, Foundations of Program Design 8th (2015)

Topics:

- 9.1 – creating subclasses
- 9.3 – class hierarchies
- 9.4 – visibility

Class Methods and Attributes (Static Members)



The Static Modifier

- We declare **static** methods and variables using the static modifier.
- It associates the method or variable with the class rather than with an object of that class.
- Static methods are sometimes called **class methods** and static variables are sometimes called **class variables**.
- Let's carefully consider the implications of each.

Static Members: Class Methods

- It is a method which **belongs to the class** and not to the object(instance).
- A static method can access only **static data**. It can not access non-static data (instance variables)
- A static method can call only other **static methods** and can not call a non-static method from it.

Static Members: Class Methods

- A static method can be **accessed directly** by the **class name** and doesn't need any object.

Syntax : ClassName.methodName(arguments)

- A common use for static methods is to access static attributes.
- A static method can **NOT** refer to "this" or "super" keywords in anyway.

Static Members: Class Methods

```
class Helper{
        public static int cube (int num)   {
            return num * num * num;
        }
}
```

Because it is declared as static, the method can be invoked as:

value = Helper.cube(5);

Class Methods Vs. Instance Methods

An **instance method** requires an object of its class to be created before it can be called, while **a static method** (class method) doesn't require object creation.

Class Methods Vs. Instance Methods

```
class Difference {
        static void display() {
            System.out.println("Programming is amazing."); }

       void show(){
           System.out.println("Java is awesome."); }

       public static void main(String[] args) {
       display(); //calling without object
       Difference t = new Difference();
       t.show(); //calling using object }
}
```

Static Members: Class Methods

- The reserve work **final** makes a method final, meaning that sub classes can not override this method.
- The compiler checks and gives an error if you try to override the method.
- When we want to restrict overriding, then make a method as a final.

Static Members: Class Attributes

- Static attributes **share the same** value for all the objects (or instances) of the class.
- When an attribute is declared with the keyword "**static**", its called a "class attribute".

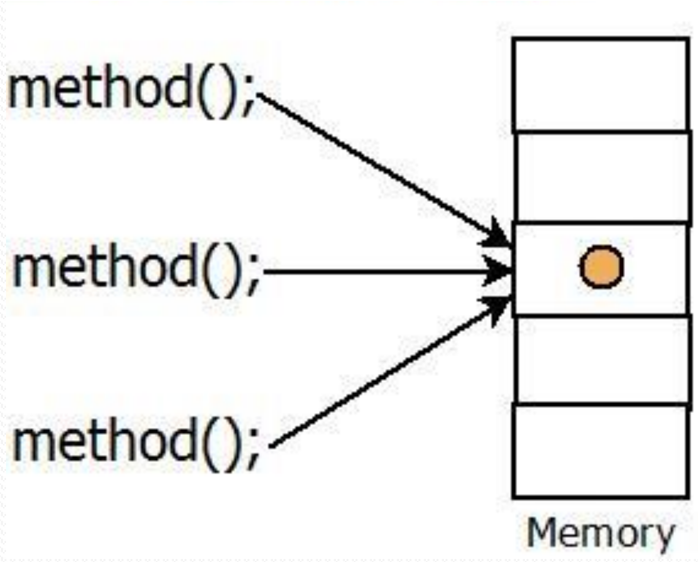
Syntax: static type attributeName;

- A static attribute can be accessed **directly** by the class name and doesn't need any object.

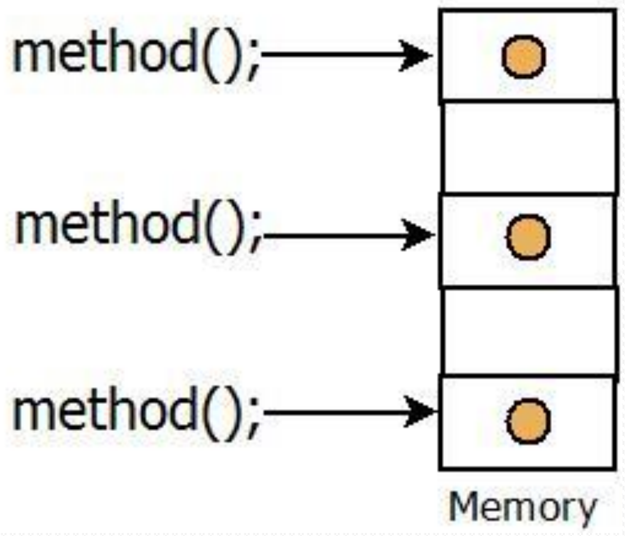
Syntax: ClassName.attributeName;

Static Members: Class Attributes

Class Attribute



Instance Attribute



Static Members: Class Attributes

Class attributes are initialized:

- When the class is loaded.
- Before any object of that class can be created.
- Before any static method of the class runs.

Static variables are initialized **only once**, at the start of the execution .

Static Members: Class Attributes

- Default values for declared and **uninitialized** static and non-static variables are same:
  - primitive integers(long, short etc): 0
  - primitive floating points(float, double): 0.0
  - boolean: false
  - object references: null

Static Members: Class Attributes

```
class VariableDemo {
    static int count=0;
    public void increment() { count++; }

    public static void main(String args[]) {
    VariableDemo obj1=new VariableDemo();
    VariableDemo obj2=new VariableDemo();
    obj1.increment();
    obj2.increment();
    System.out.println("Obj1: count is="+obj1.count);
    System.out.println("Obj2: count is="+obj2.count);
    }
}
```

What is the output?

Static Members

Not all combinations of instance and class variables and methods are allowed:

- Instance methods can access instance variables and instance methods directly.
- Instance methods can access class variables and class methods directly.
- Class methods can access class variables and class methods directly.
- Class methods ***cannot*** access instance variables or instance methods directly—they must use an object reference. Also, class methods cannot use the this keyword as there is no instance for this to refer to.

Discussion