

# Polymorphism

Dr. D. Sudheer

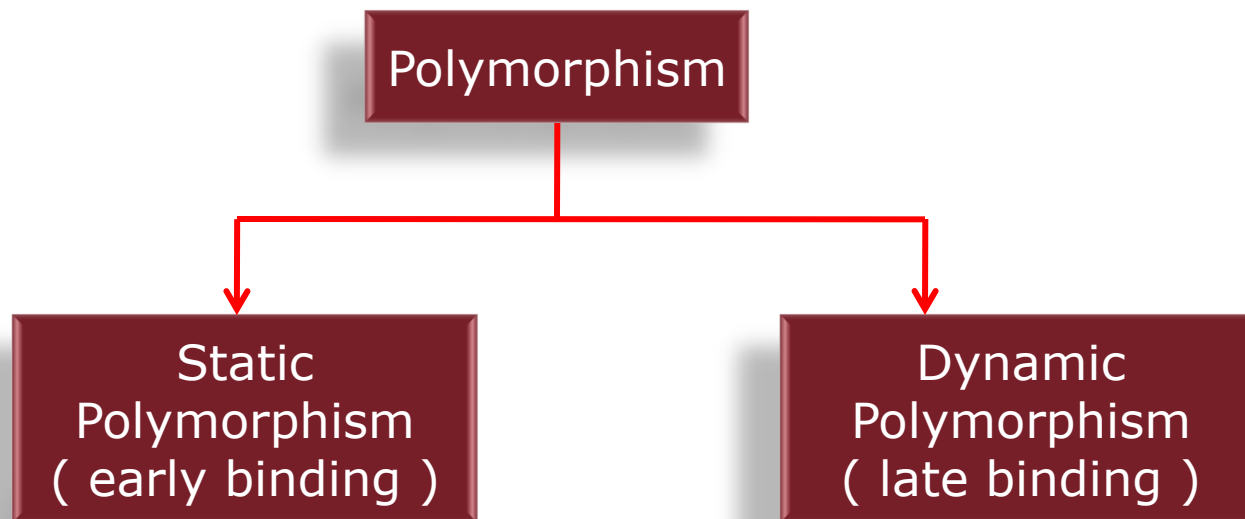
## SCOPE

## VIT-AP

# Polymorphism

Generally, Polymorphism refers to the ability to appear in many forms

## *Types of Polymorphism*



Generally, **Overloading** is an example of early binding and **Overriding** is an example of late binding. But, In java both are comes under late-binding.

# Polymorphism



## Polymorphism in a Java program

- The ability of a reference variable to change behavior according to what object instance it is holding.
- This allows multiple objects of different subclasses to be treated as objects of a single super class, while automatically electing the proper methods to apply to a particular object based on the subclass it belongs to

## Dynamic Polymorphism

- ❑ The polymorphism exhibited at run time is called dynamic polymorphism.
- ❑ This means when a method is called, the method call is bound to the method body at the time of running the program dynamically.
- ❑ In this case, Java compiler does not know which method is called at the time of compilation.
- ❑ Only JVM knows at runtime which method is to be executed.
- ❑ Hence, this is also called '**runtime polymorphism**'.

## *Resolving*

-  Method Signature is used to resolve Method Overloading at runtime
-  Dynamic method dispatch is used to resolve Method Overriding at runtime.

# Method Overloading

## Method Overloading

If a class has more than one methods having the same name but with different Signature.

### *Signature involves*

- (1) Number of parameters passed
- (2) Type of parameters and
- (3) Order of parameters

### *Note*

: Return type is not the part of the Signature.

## Implementing Method Overloading

<code>int add ( int num1 , int num2 )</code> <code>int add ( int num1 , int num2 , int num3 )</code>	✓
<code>double add ( double num1 , double num2 )</code> <code>int add ( int num1 , int num2 )</code>	✓
<code>double add ( int num1 , double num2 )</code> <code>double add ( double num1 , int num2 )</code>	✓
<code>int add ( double num1 , double num2 )</code> <code>double add ( double num1 , double num2 )</code>	✗

### *Resolving Overloaded Methods:*

Step 1: It checks for the exact match to bind it

Step 2: If exact match not found then, it tries for Automatic Conversion

Step 3: If Conversion also not possible then, it shows Compilation error.

## Method Overriding

- If a sub class has a method with the same name and with same signature that of super class is called Method Overriding.
- The created method of the subclass hides the method defined in the super-class.
- A subclass must override the abstract methods of a super-class.
- You cannot override the static and final methods of a super-class.

## *Dynamic Method Dispatch*

- ❑ It is the mechanism by which a call to an overridden method is resolved at run-time, rather than compile time.
- ❑ For this we have to create reference variable of super-class. It can refer to sub-class object.



## Dynamic Method Dispatch

```
class Base{
    void display() {
        System.out.println("Display Base");
    }
}
class Derived extends Base{
    void display() {                // Method overriding
        System.out.println("Display Derived");
    }
}
class PolyDemo{
    public static void main(String args[]) {
        Base ref;
        Base bobj=new Base();
        Derived dobj=new Derived();
        ref=bobj; ref.display();
        ref=dobj; ref.display();
    }
}
```

## Dynamic Method Dispatch

```
class Base{
    void display() {
        System.out.println("Display Base");
    }
}
class Derived extends Base{
    void display() {                // Method overriding
        System.out.println("Display Derived");
    }
}
class PolyDemo{
    public static void main(String args[]) {
        Base ref;
        Base bobj=new Base();
        Derived dobj=new Derived();
        ref=bobj; ref.display();
        ref=dobj; ref.display();
    }
}
```

No.	Method Overloading	Method Overriding
1)	Method overloading is code refinement. Same method is refined to perform different task.	Method overriding is code replacement. Sub class method overrides the super class method.
2)	Method overloading is performed <i>within class</i> .	Method overriding occurs <i>in two classes</i> that have IS-A (inheritance) relationship.
3)	In case of method overloading, <i>parameter must be different</i> .	In case of method overriding, <i>parameter must be same</i> .
4)	In java, method overloading can't be performed by changing return type of the method only. <i>Return type can be same or different</i> in method overloading. But you must have to change the parameter.	<i>Return type must be same or covariant</i> in method overriding.

In a banking system, various types of accounts (e.g., savings account, checking account, loan account) may exist, each with its own interest calculation logic. To implement runtime polymorphism:

Define a superclass `Account` with a method `calculateInterest()`.

Create subclasses `SavingsAccount`, `CheckingAccount`, and `LoanAccount`, each providing its own implementation

Imagine you are developing a system for a school that manages different types of staff members: Teachers and Administrative Staff. Both types of staff have some common attributes like name, age, and salary. However, they also have unique attributes and behaviors.

## Static Polymorphism

- ❑ The polymorphism exhibited at compile time is called static polymorphism.
- ❑ This means when a method is called, the method call is bound to the method body at the time of compiling the program.
- ❑ In this case, Java compiler knows which method is called at the time of compilation. Of course, JVM executes the method later.
- ❑ Hence, this is also called '**compile-time polymorphism**'.
- ❑ Achieving method overloading and method overriding by using **static**, **private** , and **final** members are the examples of static polymorphism

## **What is the difference between dynamic polymorphism and static polymorphism?**

Dynamic polymorphism is exhibited at runtime. Here java compiler does not understand which method is called at compilation time. Only JVM decides which method is called at runtime. Method overloading method overriding using instance methods are the examples of dynamic polymorphism.

Static polymorphism is exhibited at compile time. Here java compiler knows which method is called at compilation time. Method overriding with static methods and overloading with static, private final methods are examples of static polymorphism.

## final Keyword

The **final** keyword can be used with:

- A variable → A final variable is a constant.
- A method → You cannot override a final method.
- A class → You cannot subclass a final class.

- ❑ You can set a final variable once only, but that assignment can occur independently of the declaration; this is called **a blank final variable**.
- ❑ A blank final instance attribute must be set in every constructor.
- ❑ A blank final method variable must be set in the method body before being used.



*Constants are static final variables.*

```
static final int MAX_SIZE = 10;  
MAX_SIZE = 20; // compile time error
```

*final with method : ( prevents from Overriding )*

```
class Parent {  
    final void show() {  
        System.out.print("Parent Data");  
    }  
}  
  
class Child extends Parent {  
    void show() { // compile time error  
        System.out.print("Child Data");  
    }  
}
```

*final with class : ( prevents from inheriting )*

```
final class Parent {  
    . . .  
    . . .  
}  
  
class Child extends Parent { // compile time error  
    . . .  
    . . .  
}
```

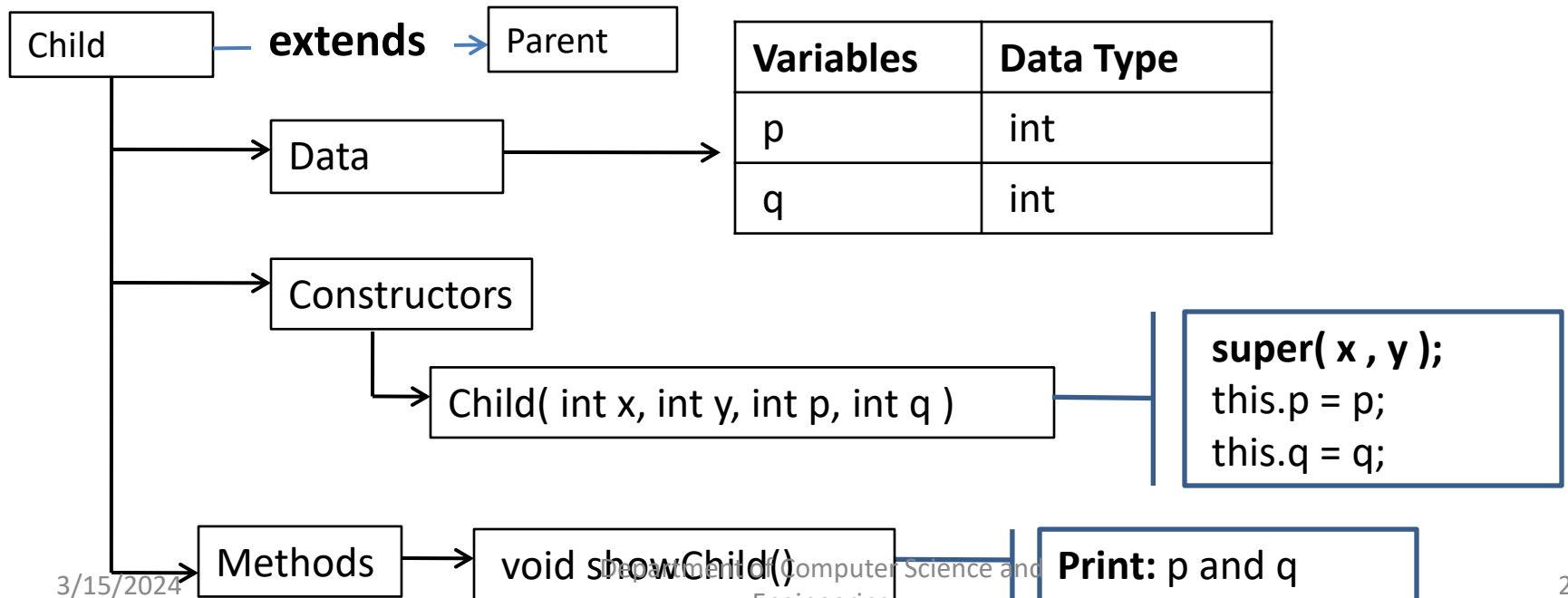
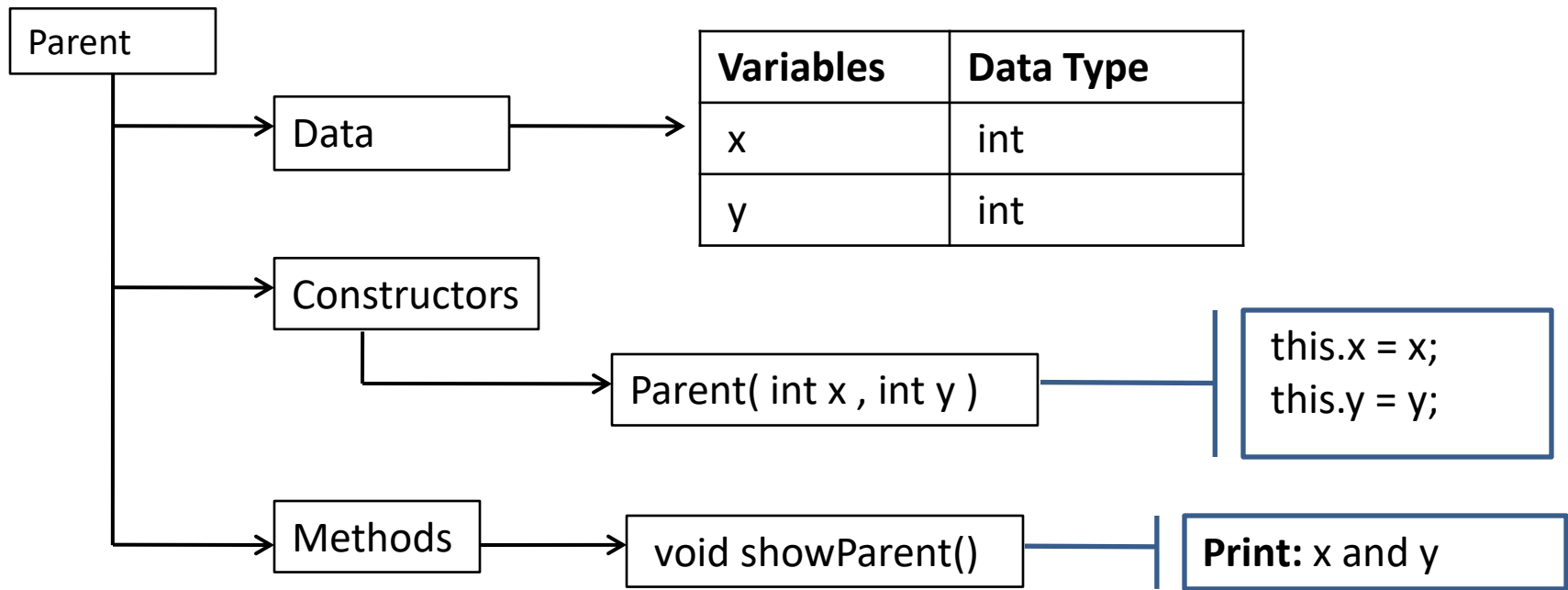
# Usage of “super” keyword

We use “**super**” keyword for 2 reasons:

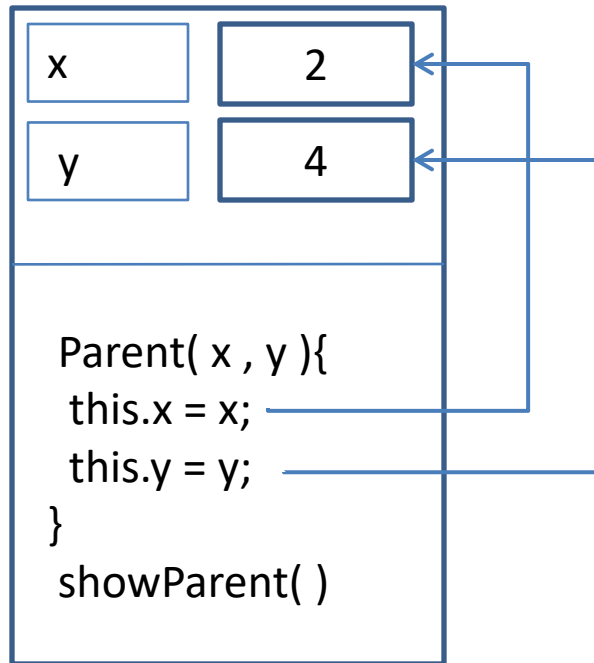
- ❑ To refer / invoke super class Constructors
- ❑ To call super class members in Overriding concept

### *Super Class Initialization*

- A subclass inherits all methods and variables from the superclass but a subclass does not inherit the constructor from the superclass.
- To invoke a parent constructor, you must place a call to **super** in the first line of the constructor.
- You can call a specific parent constructor by the arguments that you use in the call to super.
- If no **this** or **super** call is used in a constructor, then the compiler adds an implicit call to **super()** that calls the parent no argument constructor (default constructor). If the parent class defines constructors, but does not provide a no-argument constructor, then a compiler error message is issued.



# Memory



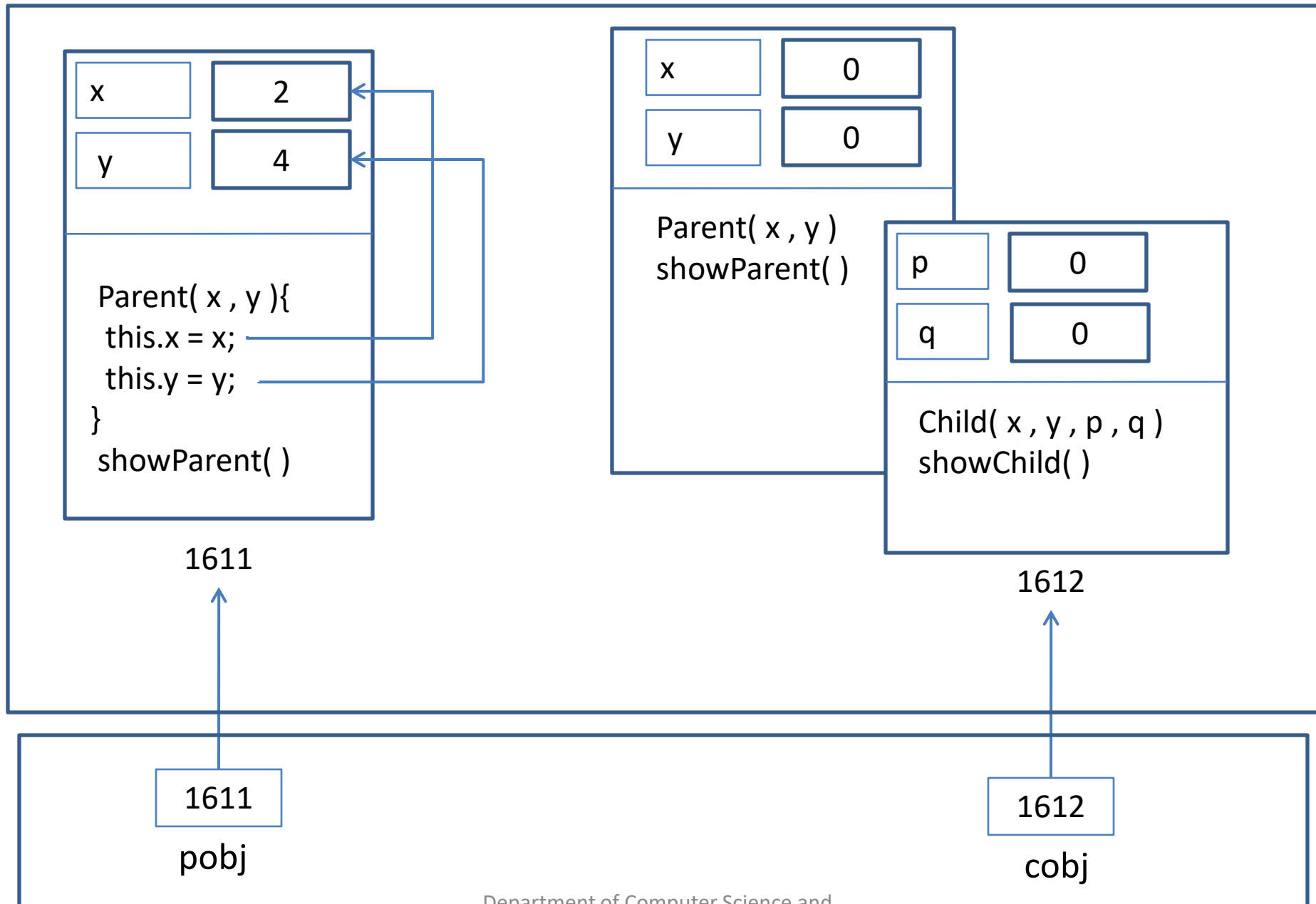
1611



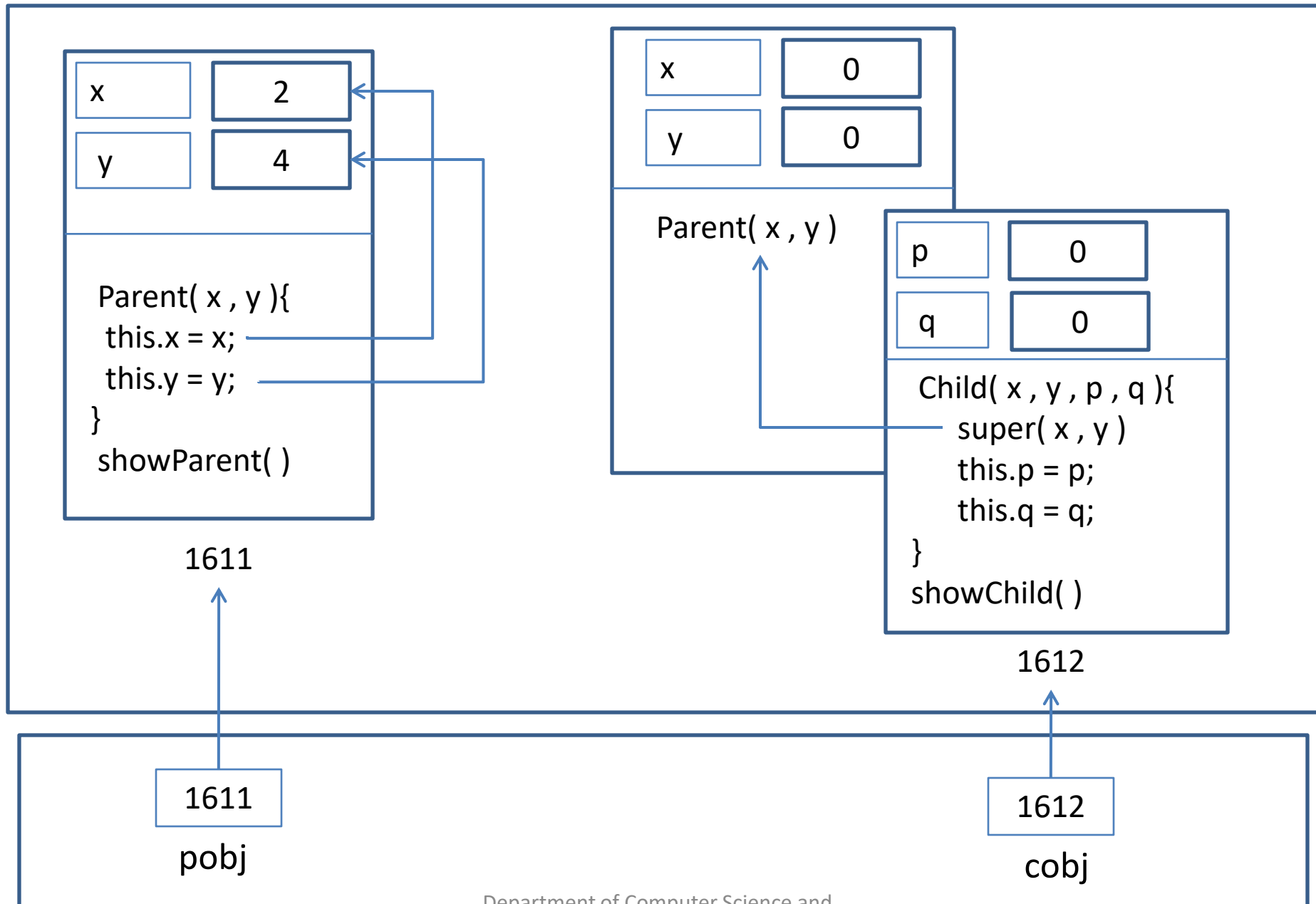
1611

pobj

# Memory

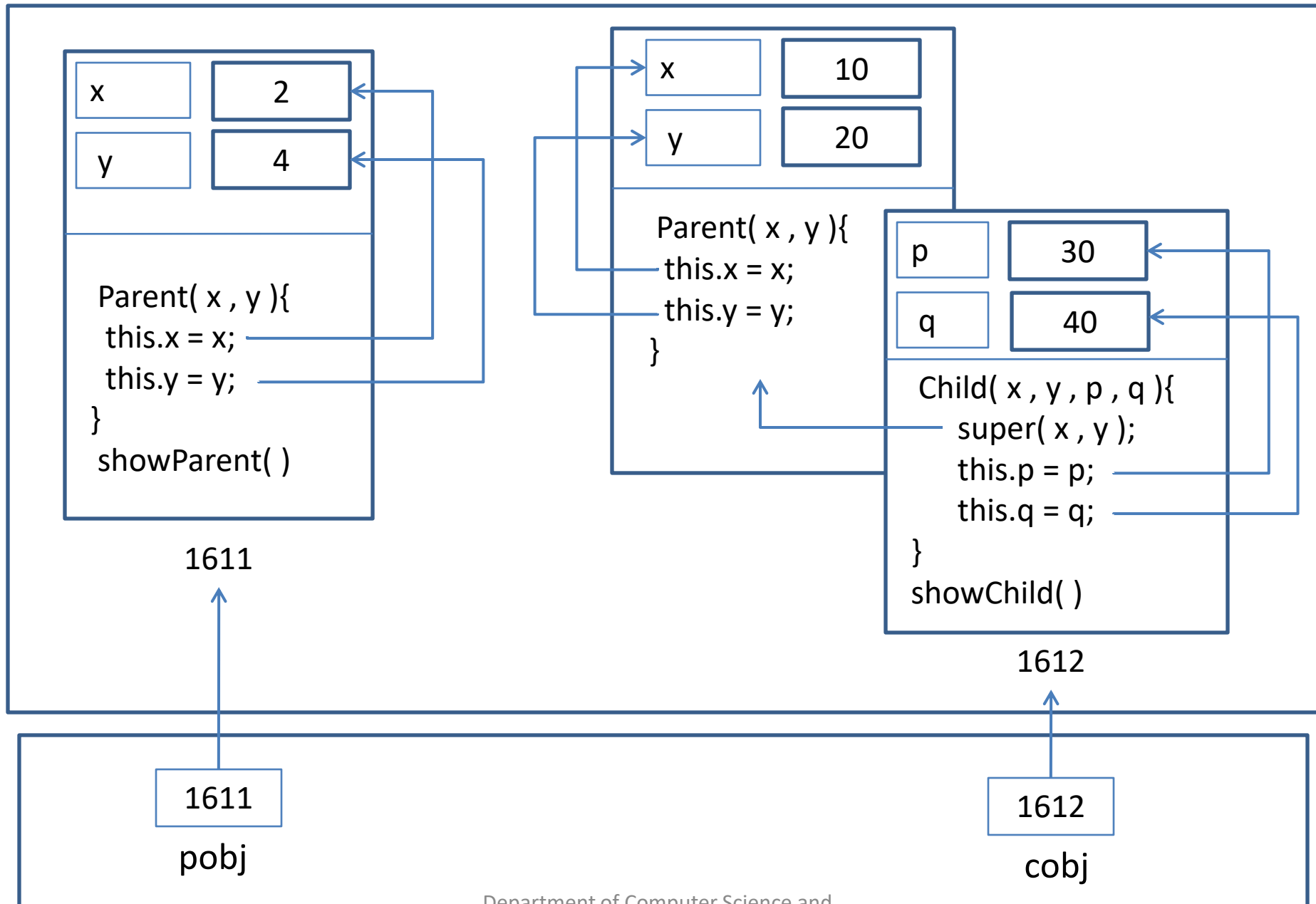


# Memory





# Memory



# Memory

