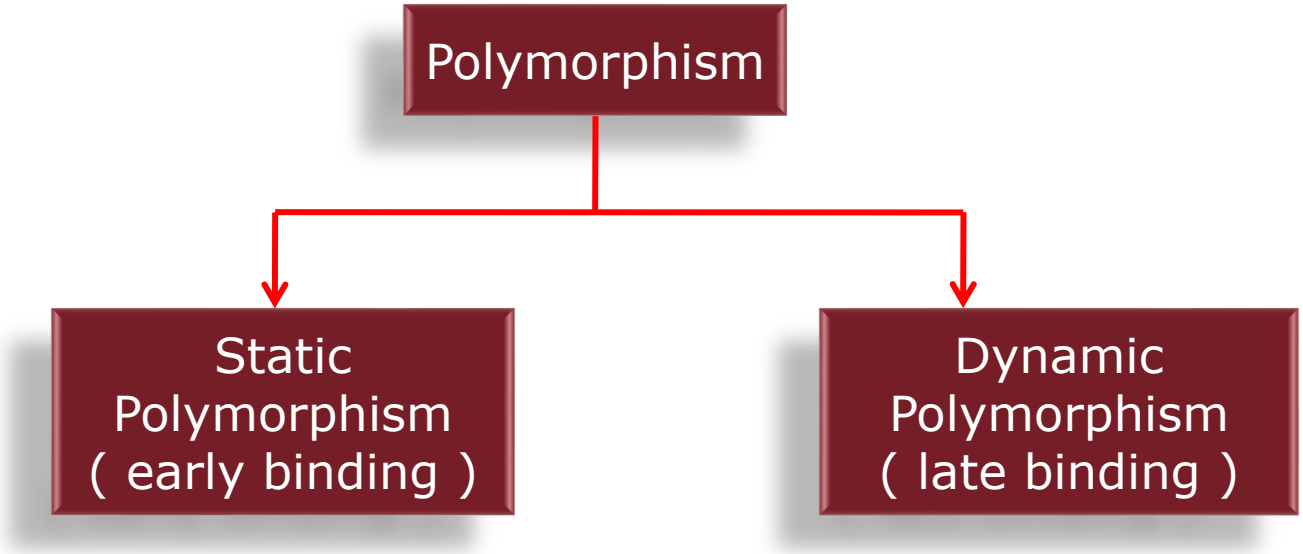
Polymorphism

Polymorphism

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

Types of Polymorphism



Generally, **Overloading** in 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

int add (int num1 , int num2) int add (int num1 , int num2 , int num3)	✓
double add (double num1 , double num2) int add (int num1 , int num2)	✓
double add (int num1 , double num2) double add (double num1 , int num2)	✓
int add (double num1 , double num2) double add (double num1 , double num2)	✗

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 *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

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

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

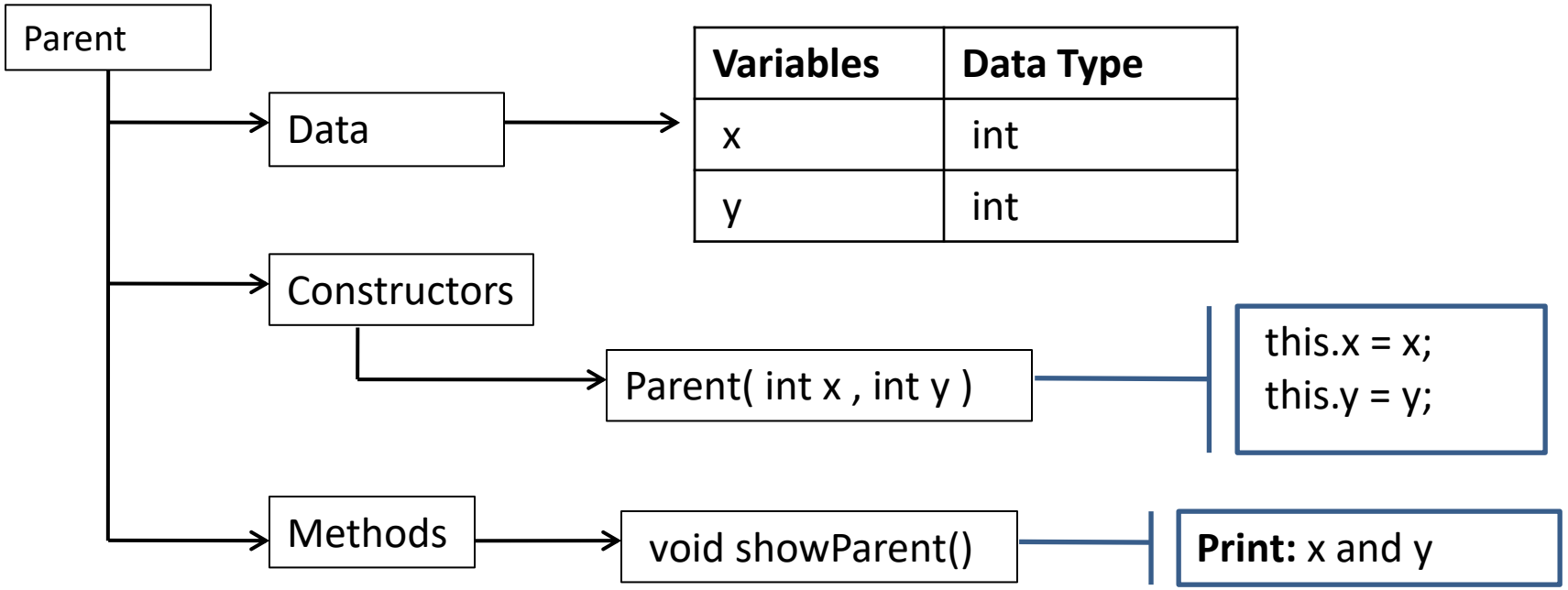
Super Class Initialization

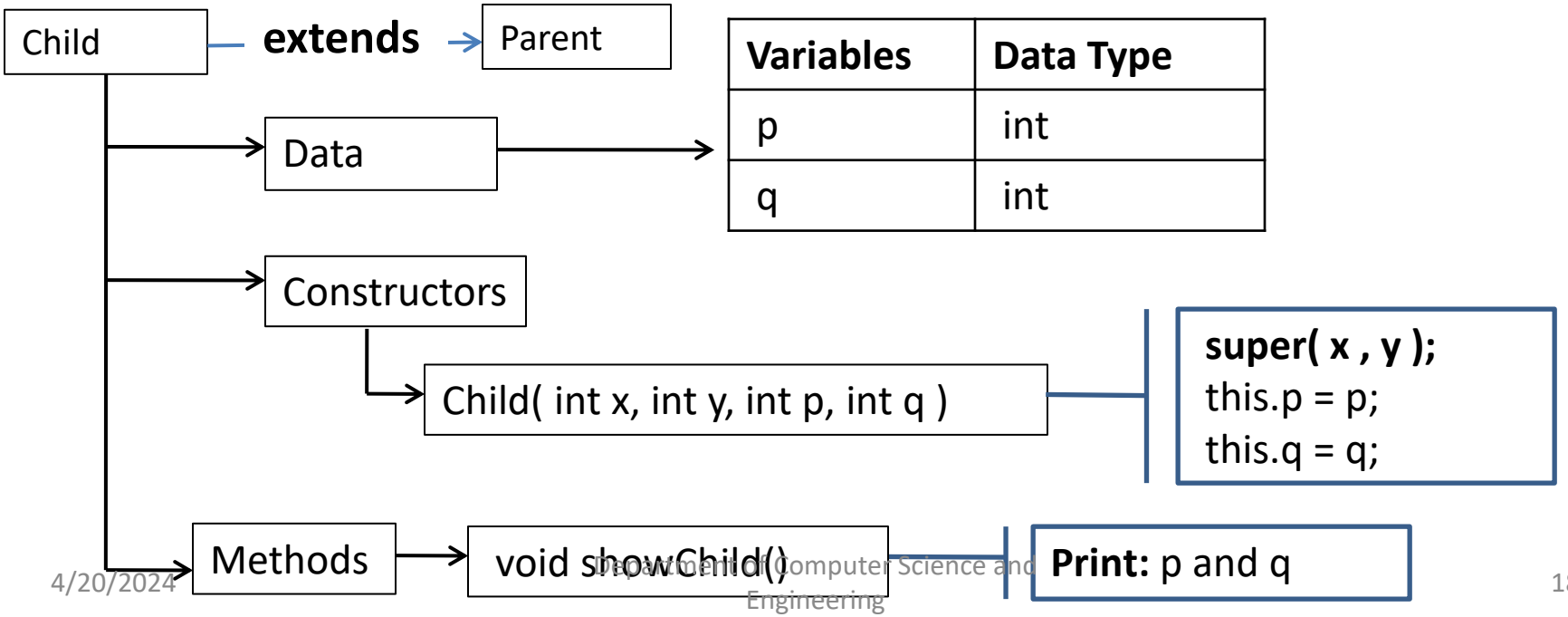
• A subclass inherits all methods and variables from the superclass but a subclass does not inherit the constructor from the superclass.

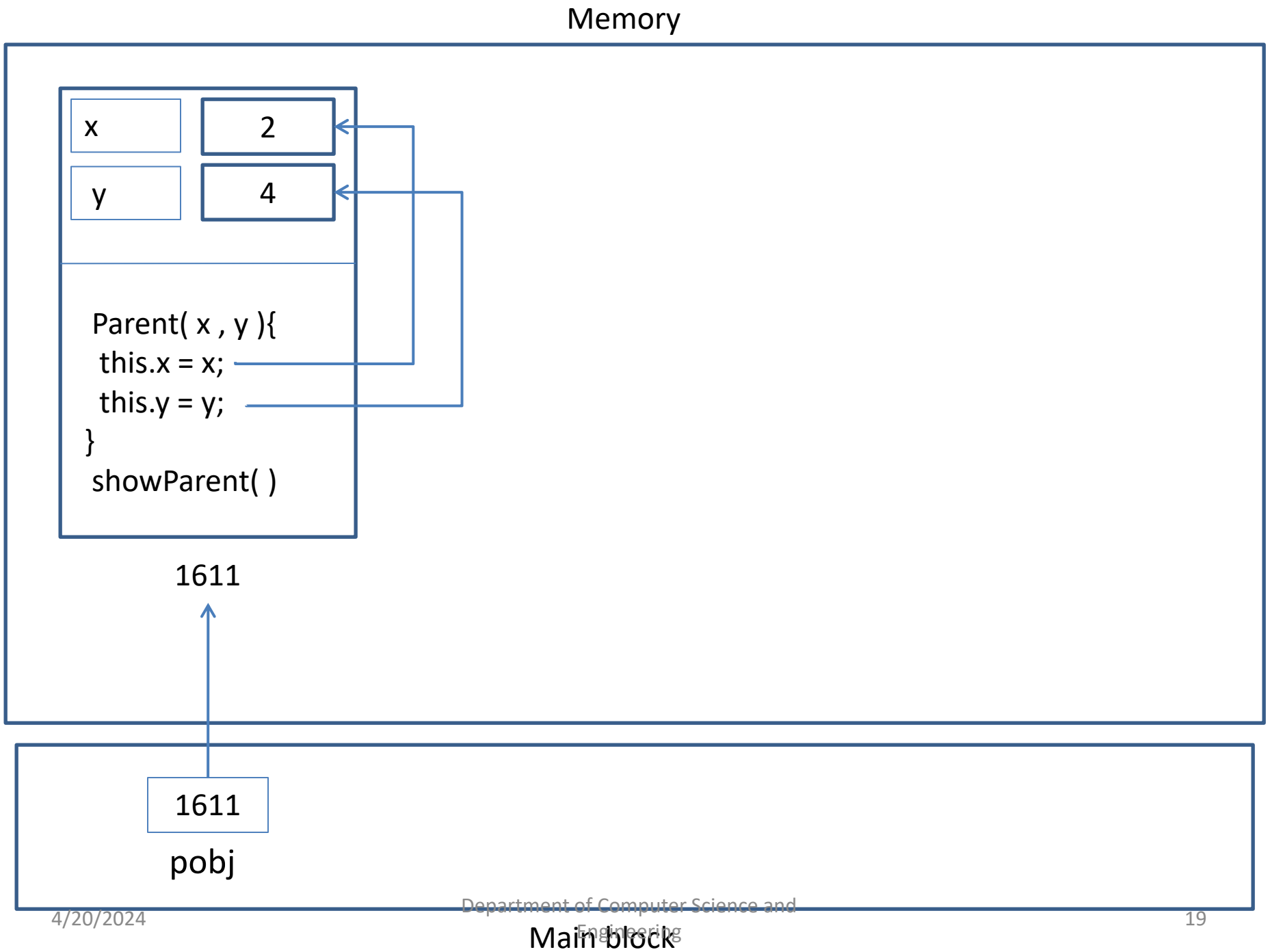
• To invoke a parent constructor, you must place a call to **super** in the first line of the constructor.

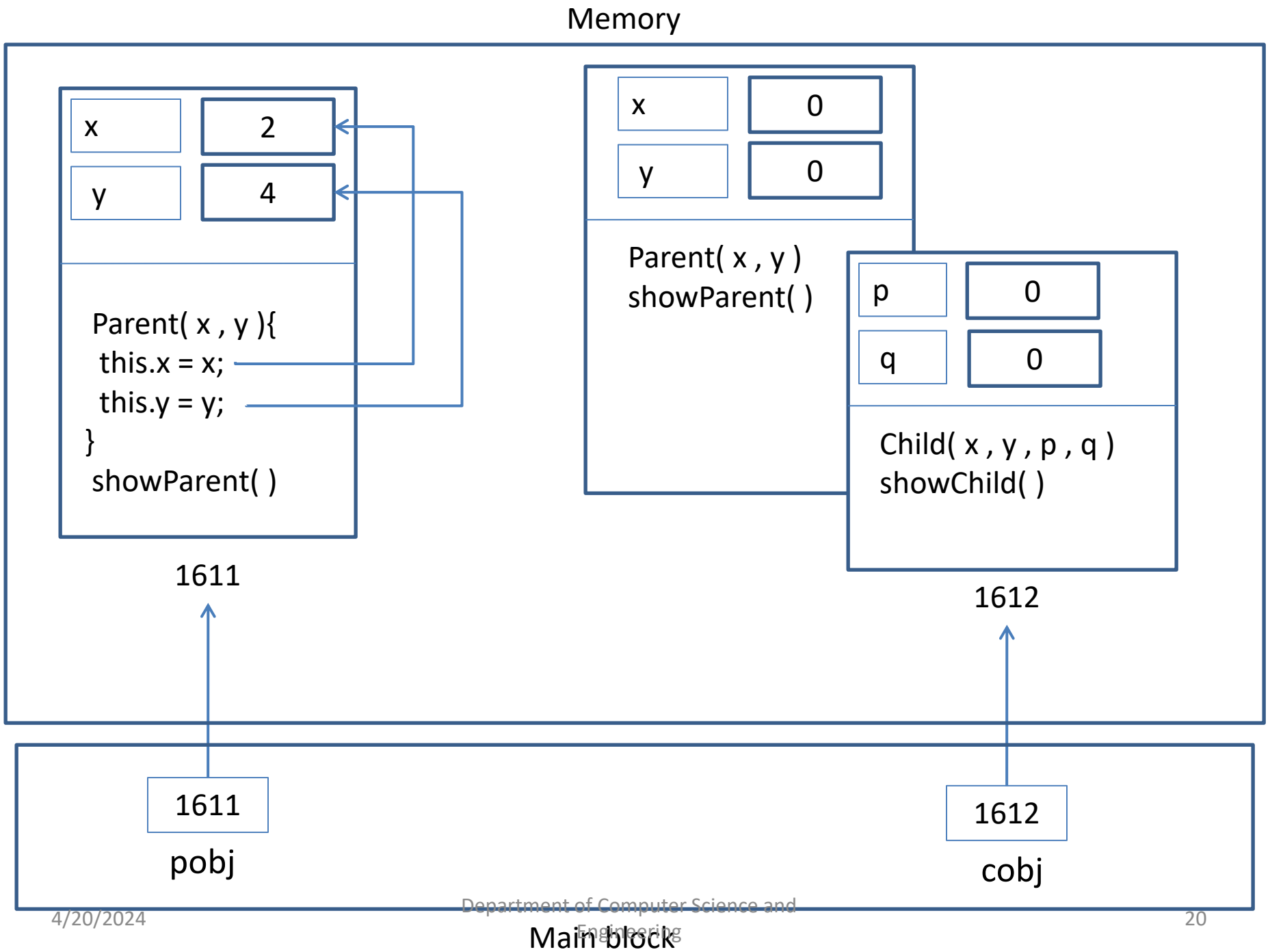
• You can call a specific parent constructor by the arguments that you use in the call to super.

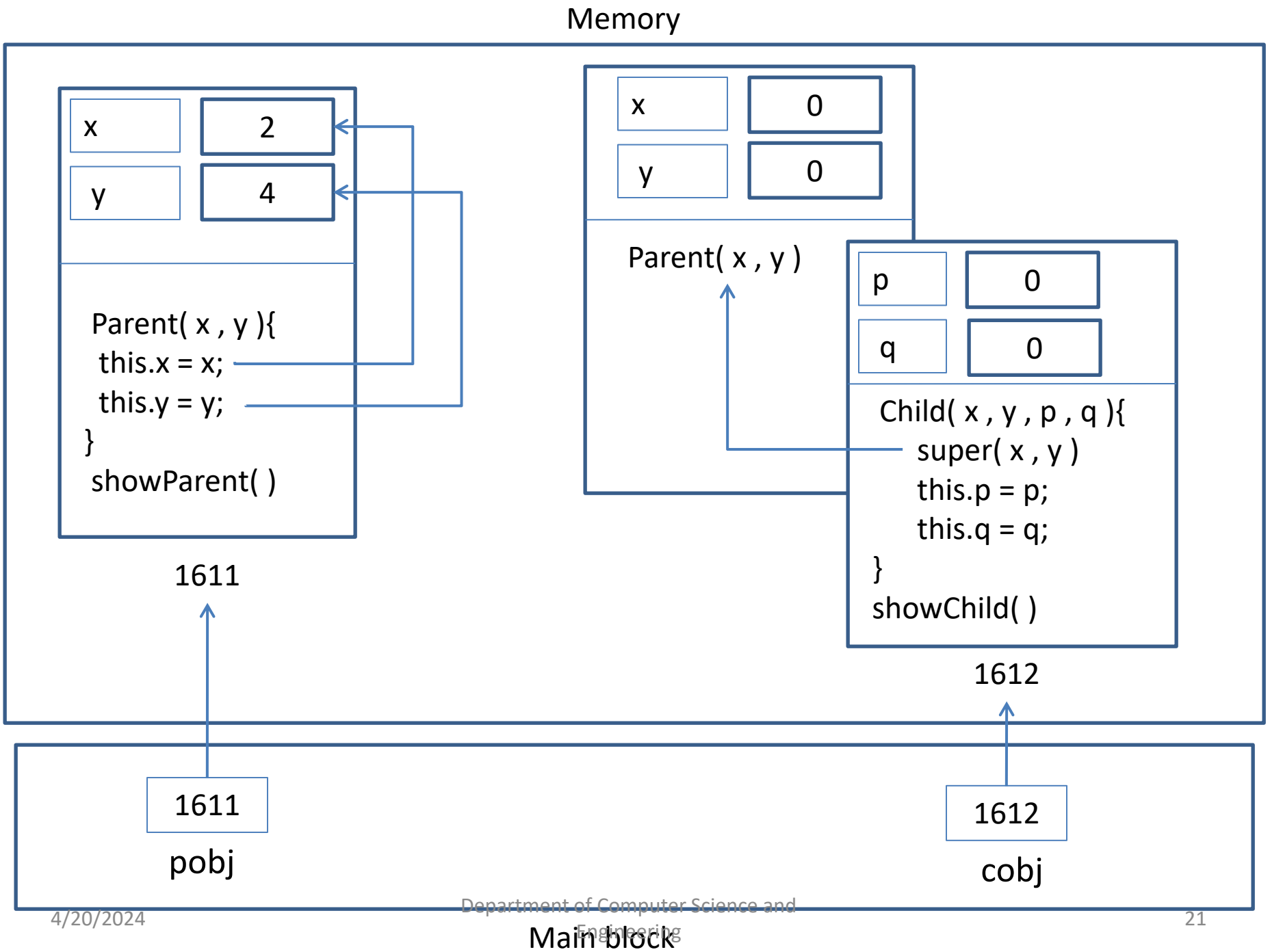
• If no **this** or **super** call is used in a constructor, then the compiler adds an implicit call to **super()** that calls the parent no argument constructor (default constructor). If the parent class defines constructors, but does not provide a no-argument constructor, then a compiler error message is issued.

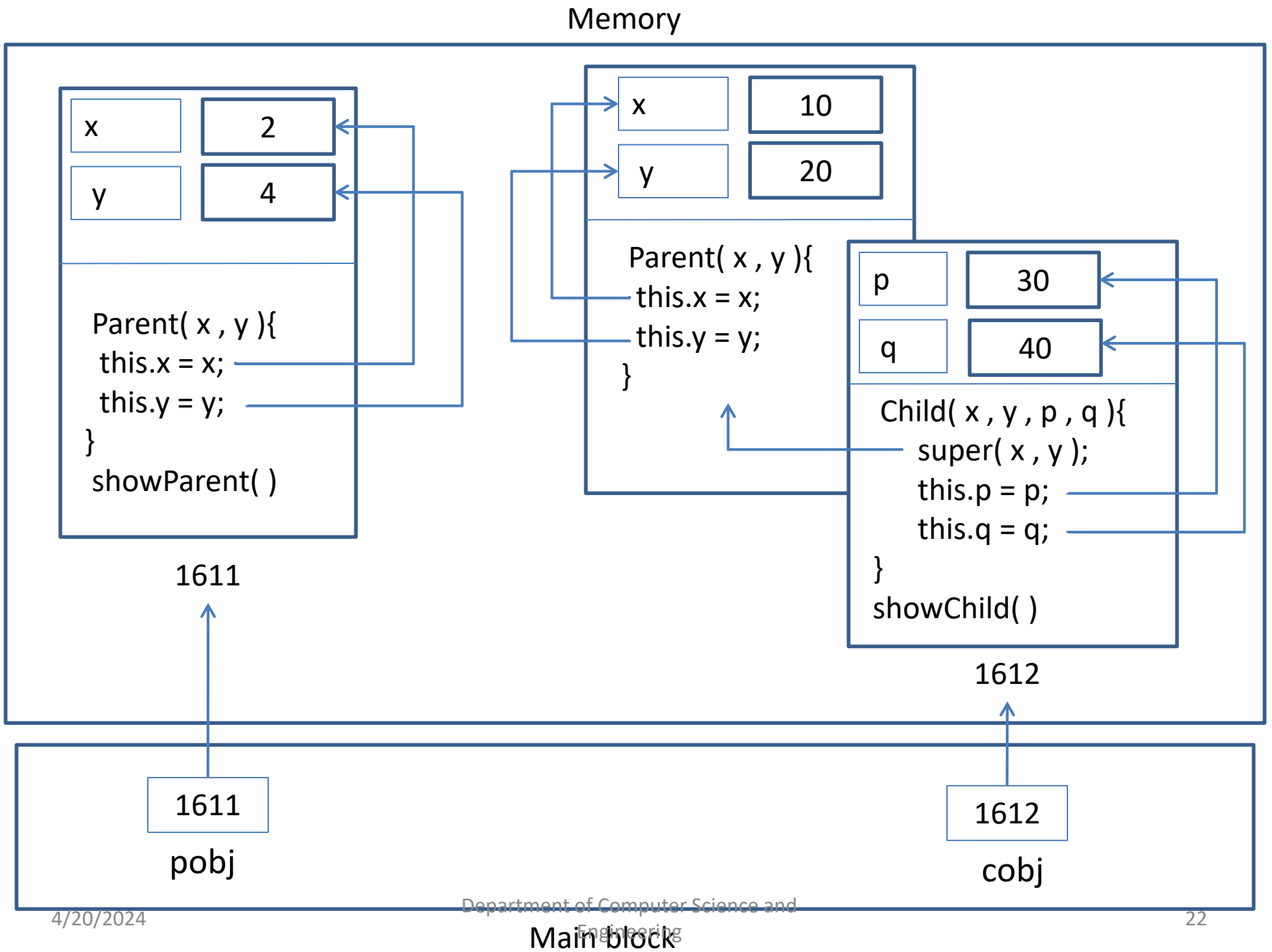

Parent
Data
Variables
Data Type
x
int
y
int
Constructors
Parent(int x , int y)
this.x = x;
this.y = y;
Methods
void showParent()
Print: x and y

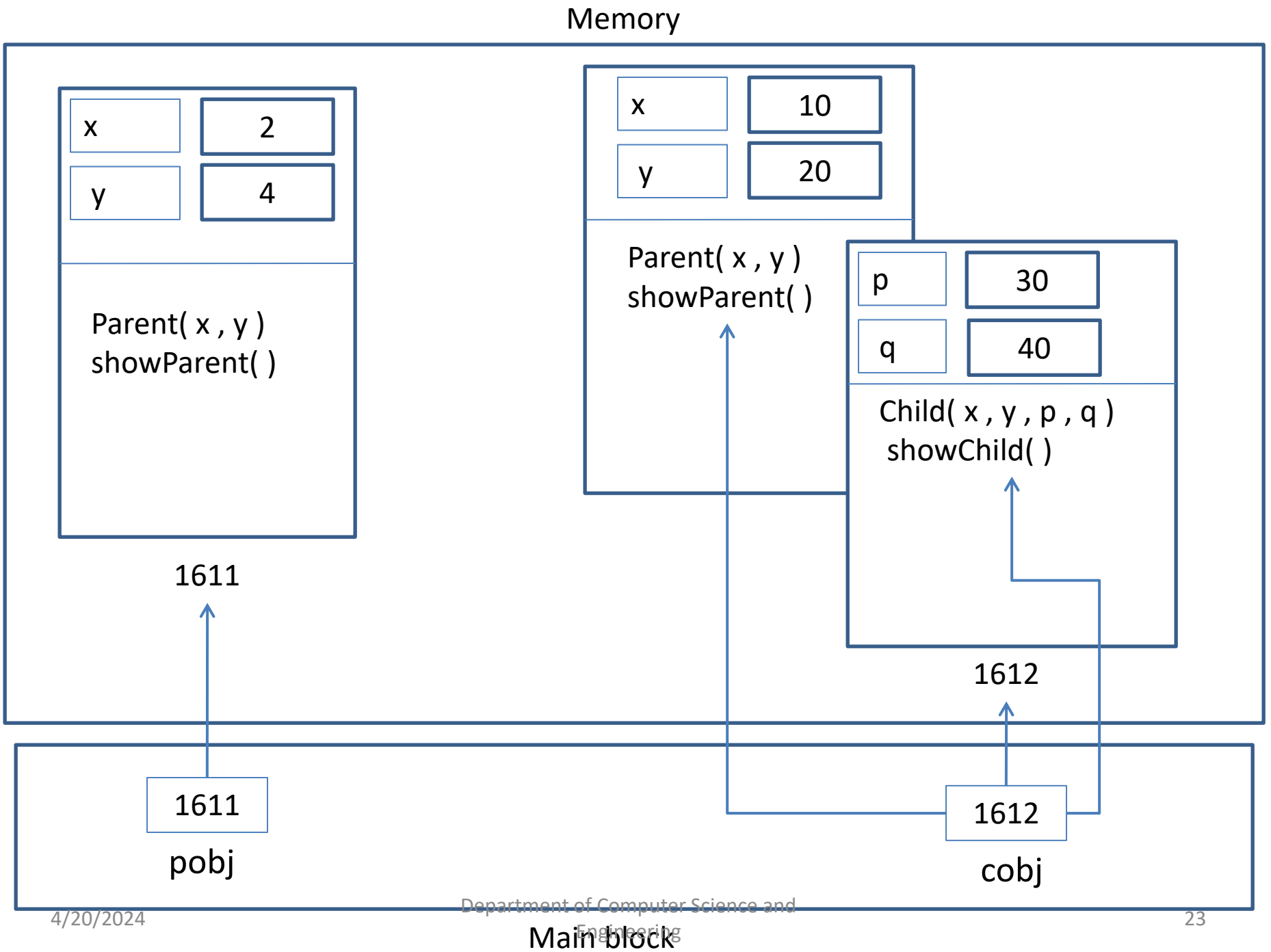

Child
extends
Parent
Data
Variables
Data Type
p
int
q
int
Constructors
Child(int x, int y, int p, int q)
super(x , y);
this.p = p;
this.q = q;
Methods
void showChild()
Print: p and q

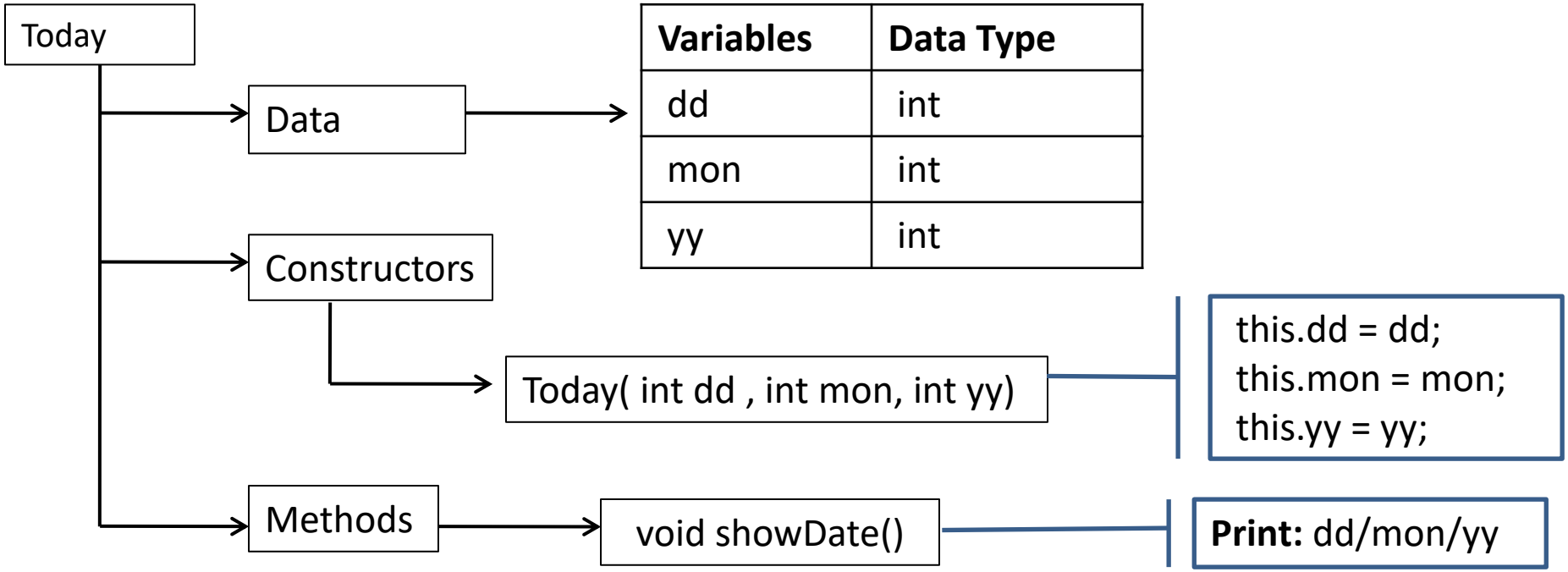

Memory
x
2
y
4
Parent(x , y){
this.x = x;
this.y = y;
}
showParent( )
1611
1611
pobj
Main block

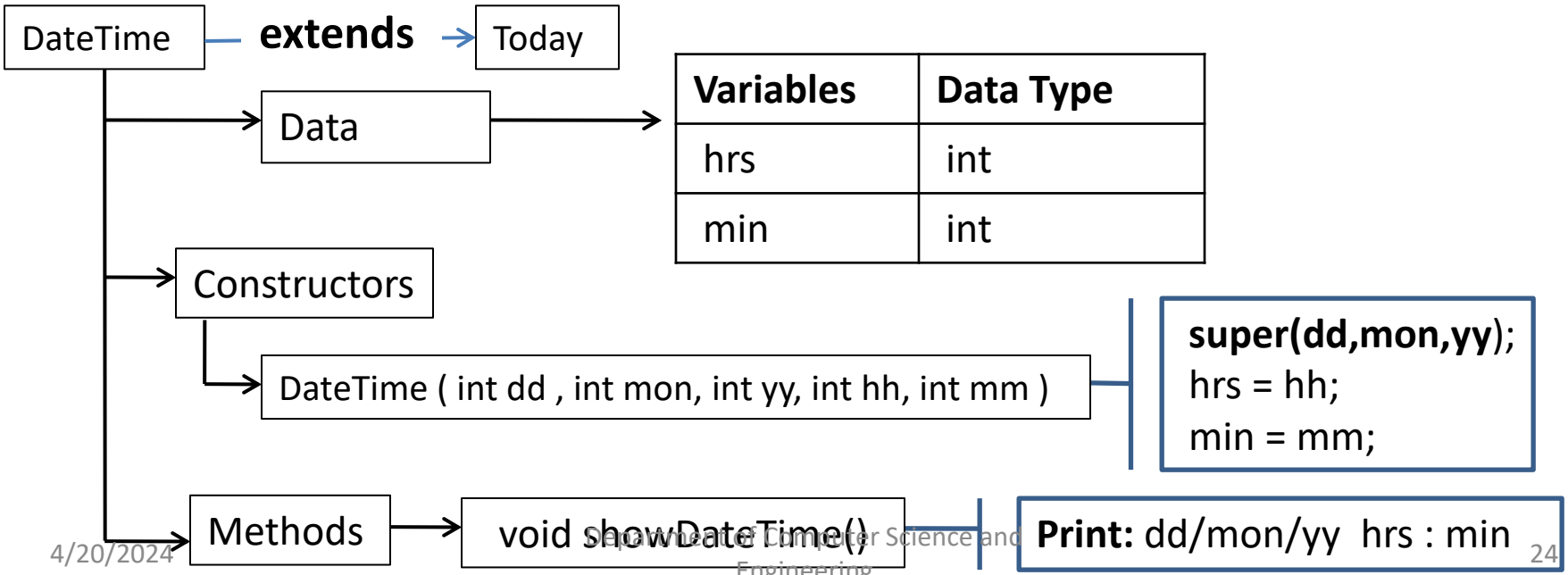

Memory
x
2
y
4
Parent(x , y){
this.x = x;
this.y = y;
}
showParent( )
1611
x
0
y
0
Parent(x , y)
showParent( )
p
0
q
0
Child(x , y , p , q)
showChild( )
1612
1611
pobj
1612
cobj
Main block


Memory
x
2
y
4
Parent(x , y){
this.x = x;
this.y = y;
}
showParent( )
1611
x
0
y
0
Parent(x , y)
p
0
q
0
Child(x , y , p , q){
super(x , y)
this.p = p;
this.q = q;
}
showChild( )
1612
1611
pobj
1612
cobj
Main block

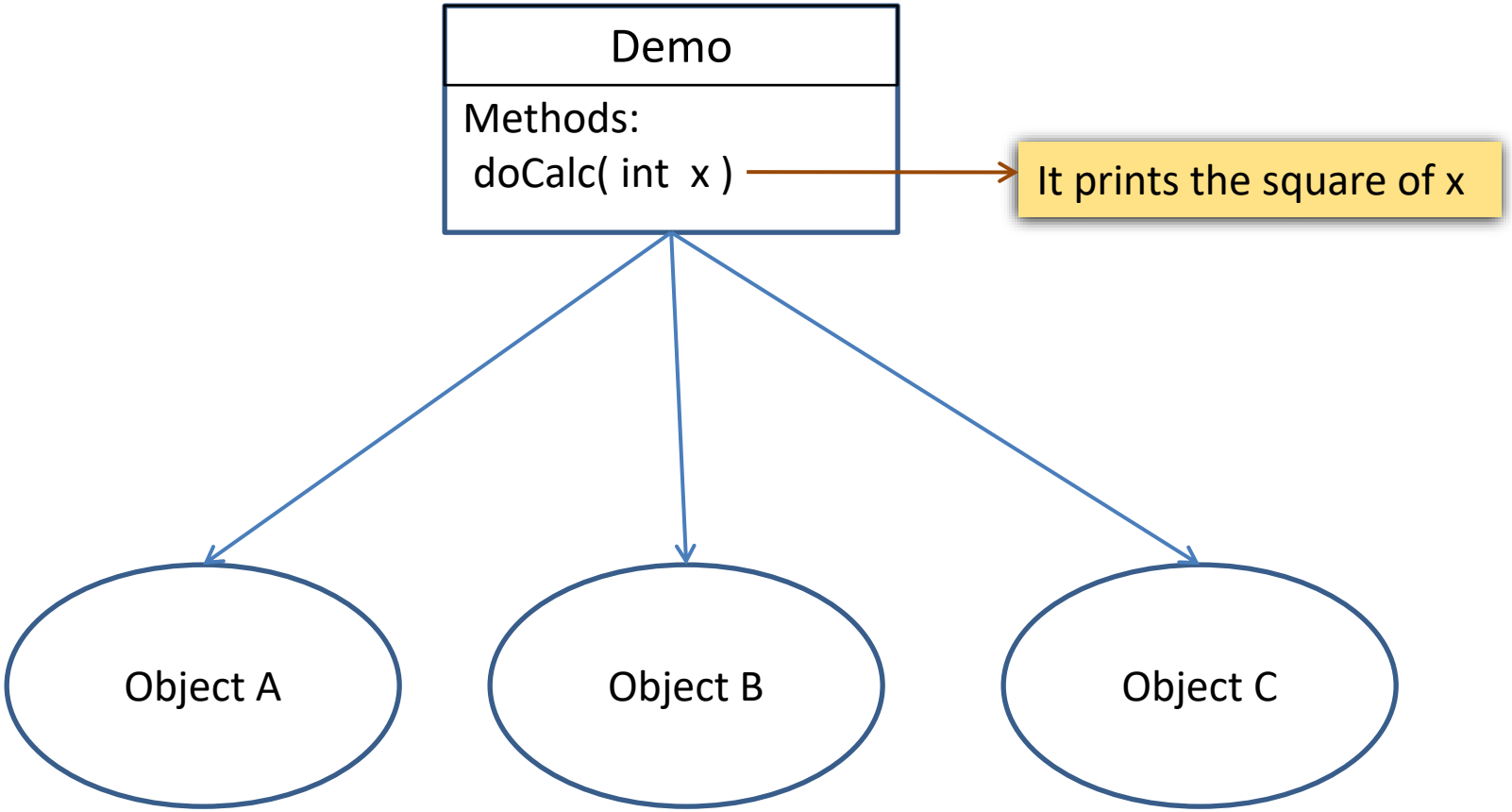

Memory
x
2
y
4
Parent(x , y){
this.x = x;
this.y = y;
}
showParent( )
1611
x
10
y
20
Parent(x , y){
this.x = x;
this.y = y;
}
p
30
q
40
Child(x , y , p , q){
super(x , y);
this.p = p;
this.q = q;
}
showChild( )
1612
1611
pobj
1612
cobj
Main block


Memory
x
2
y
4
Parent(x , y)
showParent( )
1611
x
10
y
20
Parent(x , y)
showParent( )
p
30
q
40
Child(x , y , p , q)
showChild( )
1612
1611
pobj
1612
cobj
Main block

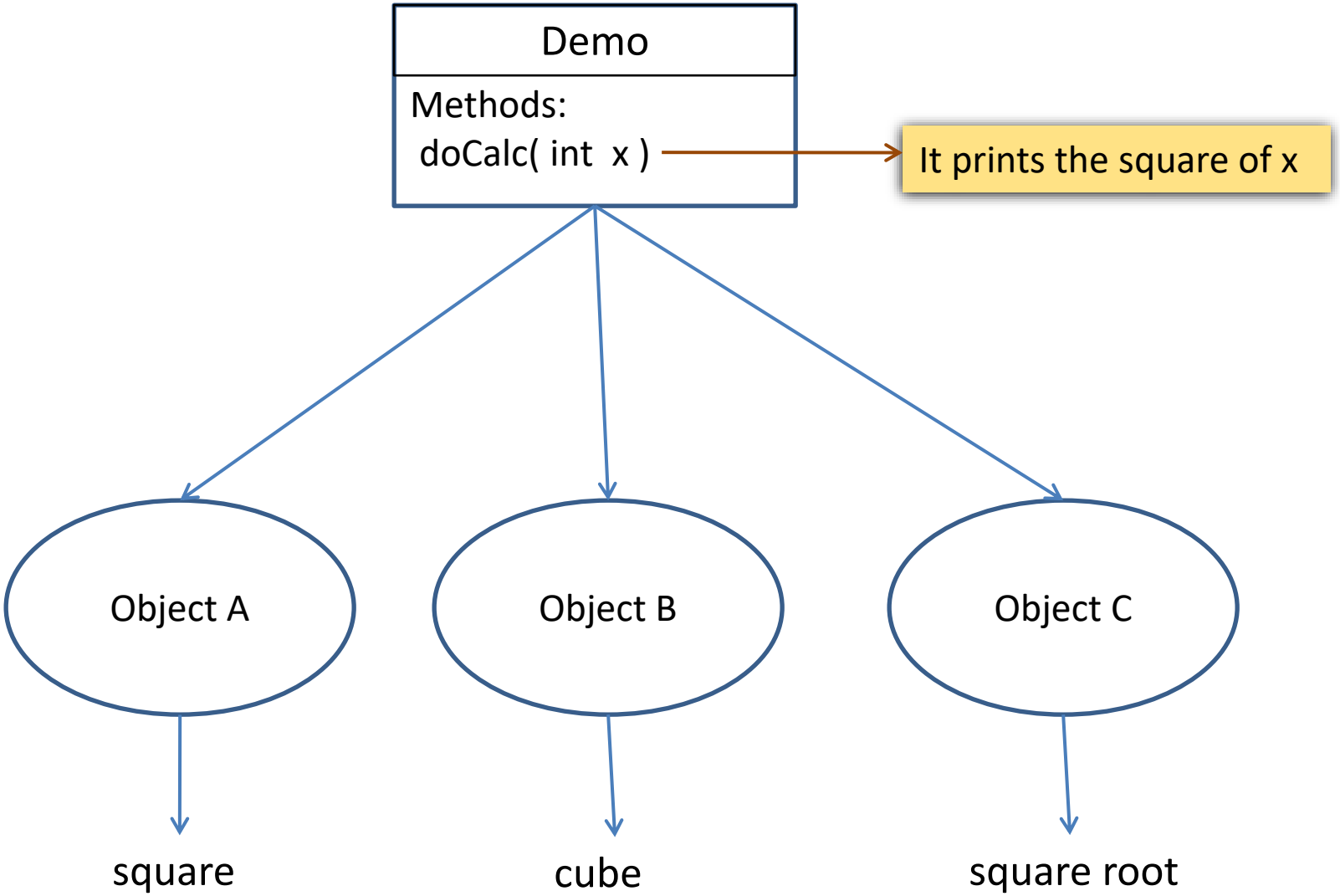

Today
Data
Variables
Data Type
dd
int
mon
int
yy
int
Constructors
Today(int dd , int mon, int yy)
this.dd = dd;
this.mon = mon;
this.yy = yy;
Methods
void showDate()
Print: dd/mon/yy


DateTime
extends
Today
Data
Variables
Data Type
hrs
int
min
int
Constructors
DateTime (int dd , int mon, int yy, int hh, int mm)
super(dd,mon,yy);
hrs = hh;
min = mm;
Methods
void showDateTime()
Print: dd/mon/yy hrs : min

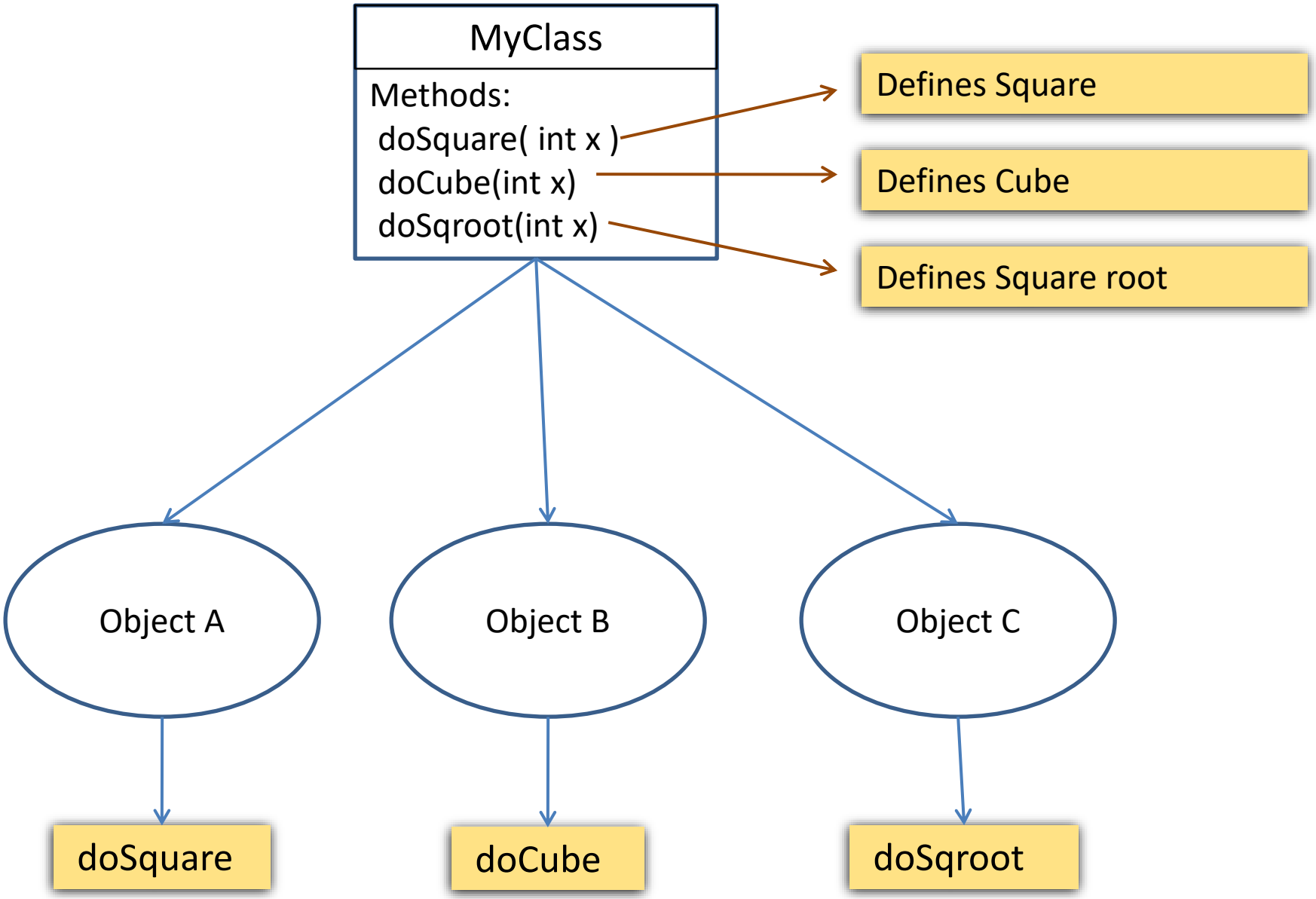
Abstract Classes



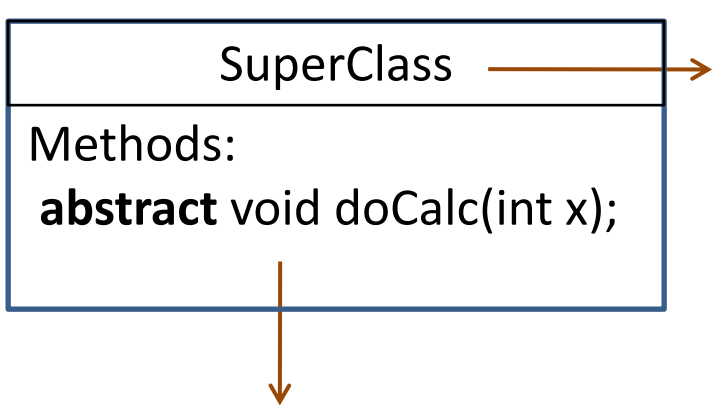
Here, A.doCalc( ) prints the square of given value, Similarly B.doCalc( ) and C.doCalc( ) also prints the square of the given value.



What ? Objects needs are different



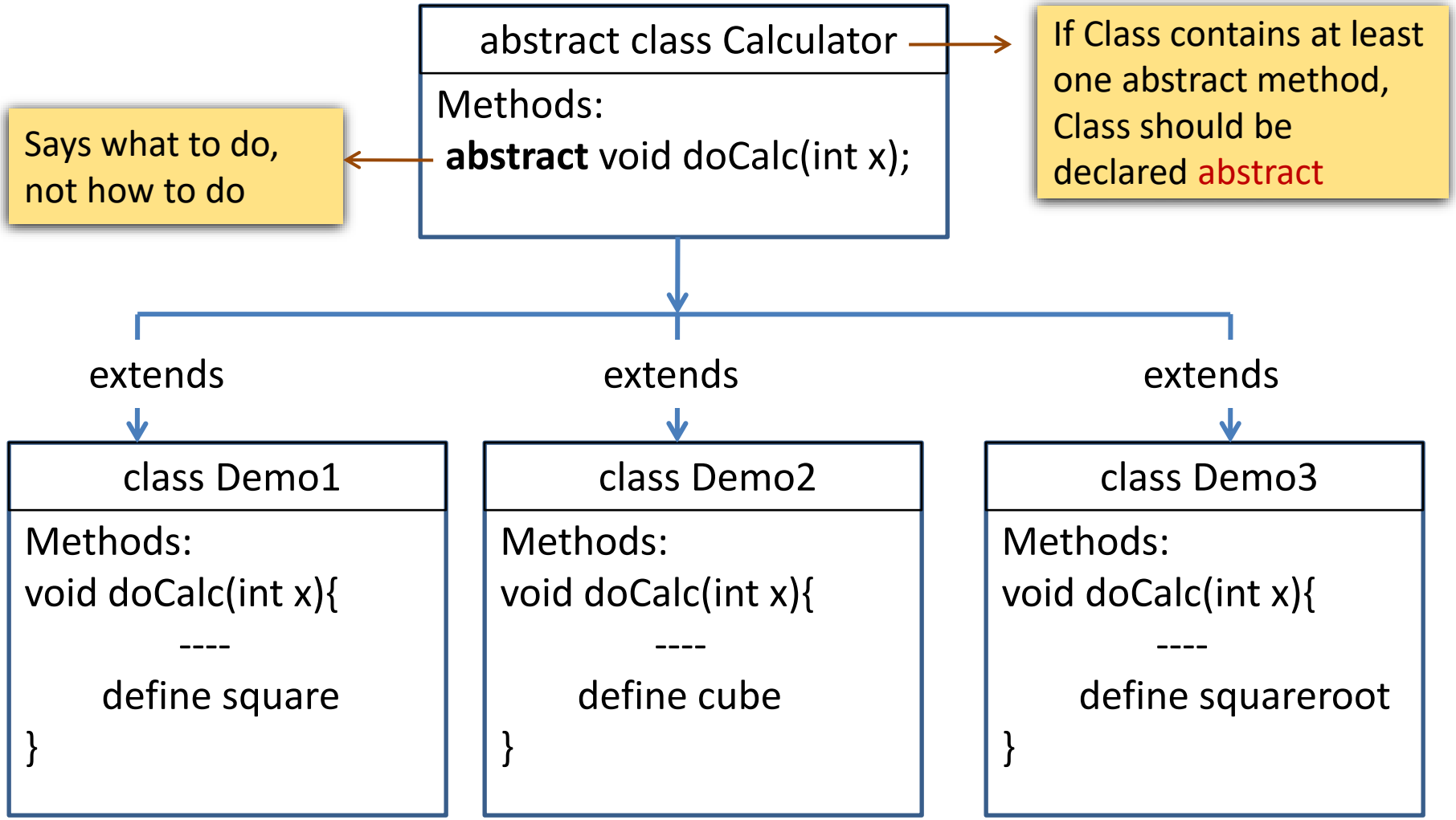
It is the responsibility of Class to define the behavior of objects



If Class contains at least one abstract method, Class should be declared abstract

Simply says what to do, not how to do

Now, Any class, which extends SuperClass, it is mandatory for that class to override abstract methods otherwise it generates compile-time error



Now, Any class, which extends SuperClass, it is mandatory for that class to override abstract methods otherwise it generates compile-time error

Abstract Methods

- An abstract method contains only the declaration for a method without any implementation details. It simply says what to do not how to do.

```
abstract void calcData ( double x , double y ) ;
```

Abstract Classes

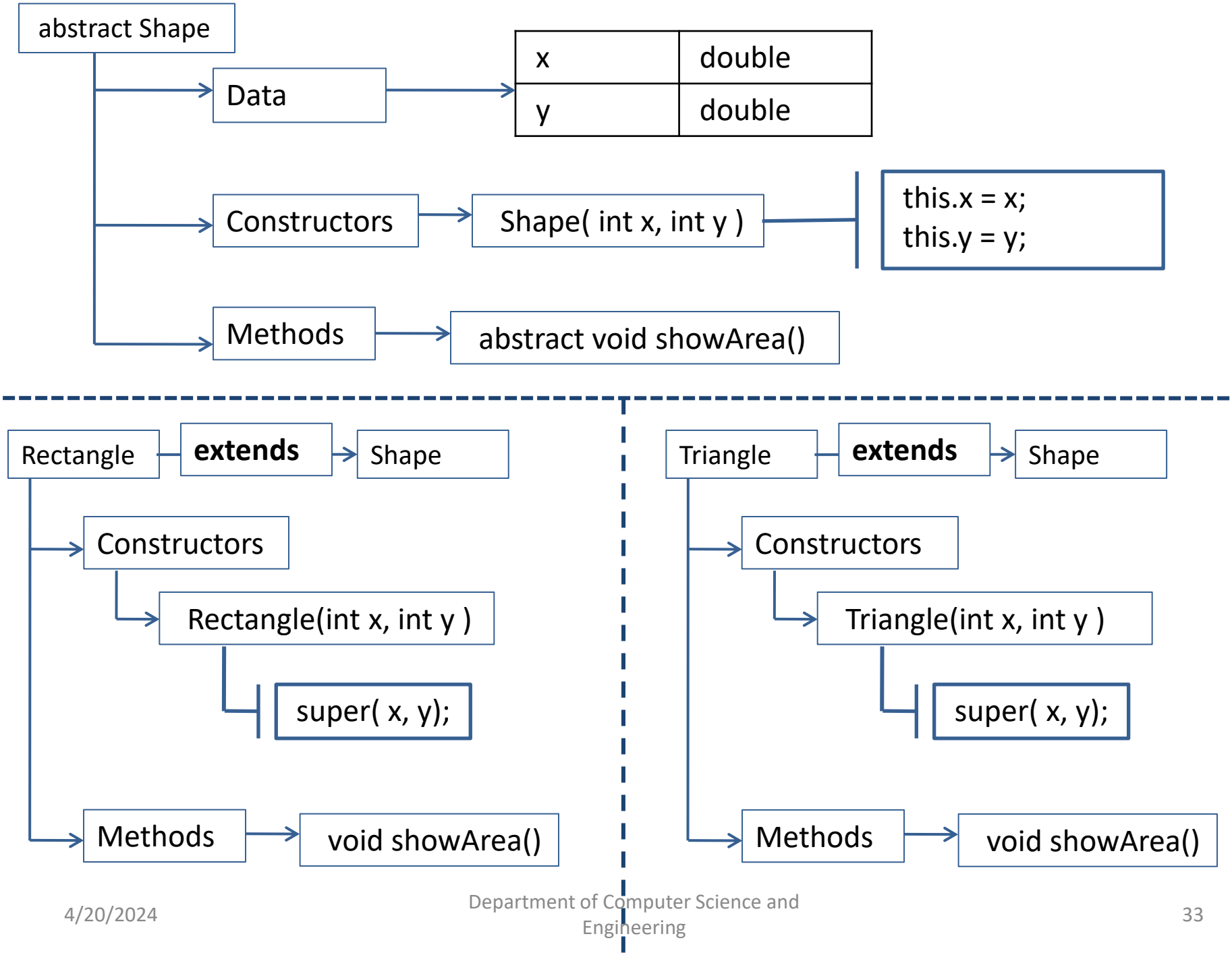
- An abstract class is a class that is declared abstract—it may or may not include abstract methods.
- If a class includes at least one abstract method, that class must be declared abstract.
- The non-abstract methods of an abstract class are **concrete** methods.

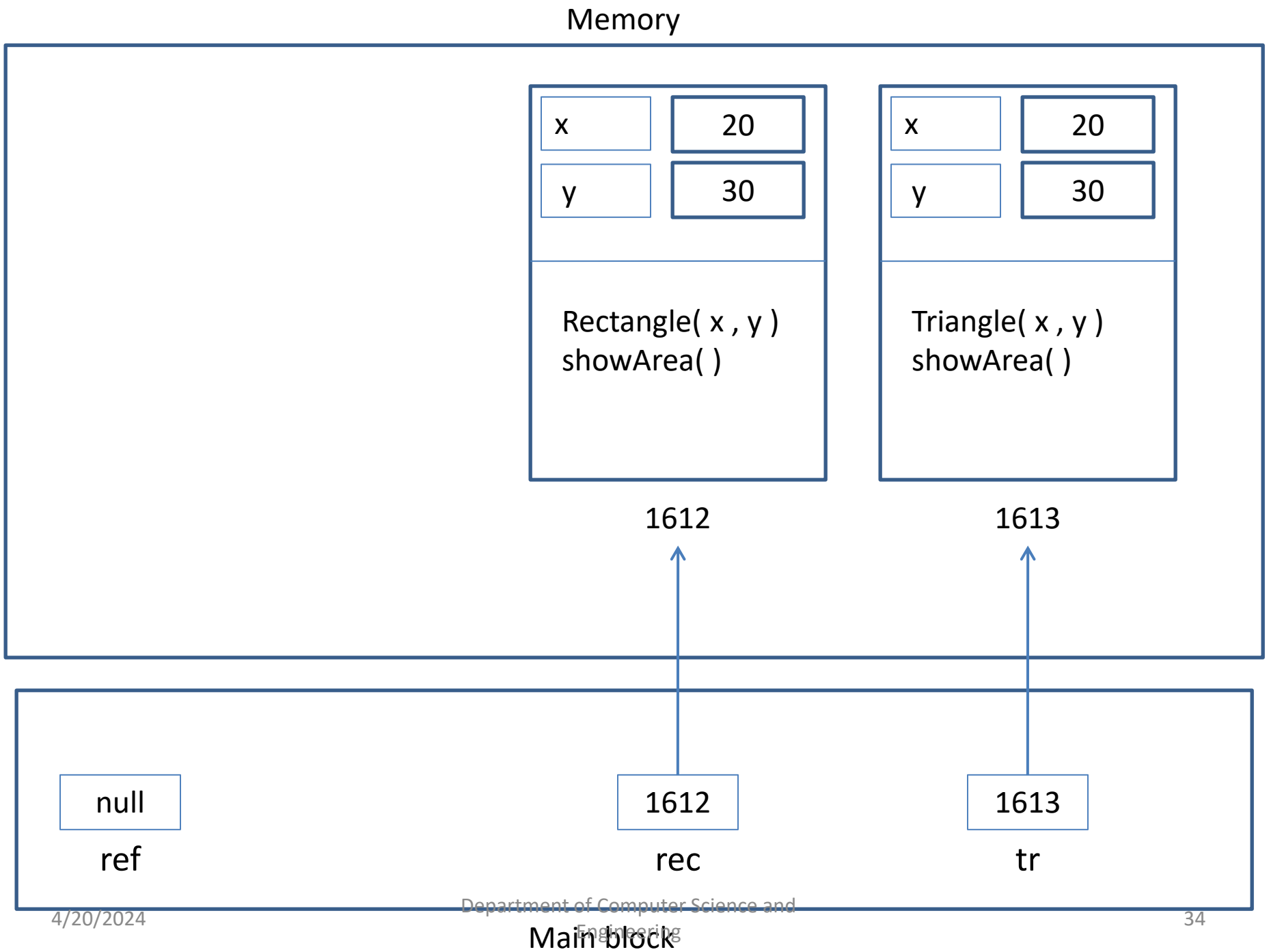
```
abstract  class MyClass {
    // declare abstract and non-abstract methods
 }
```

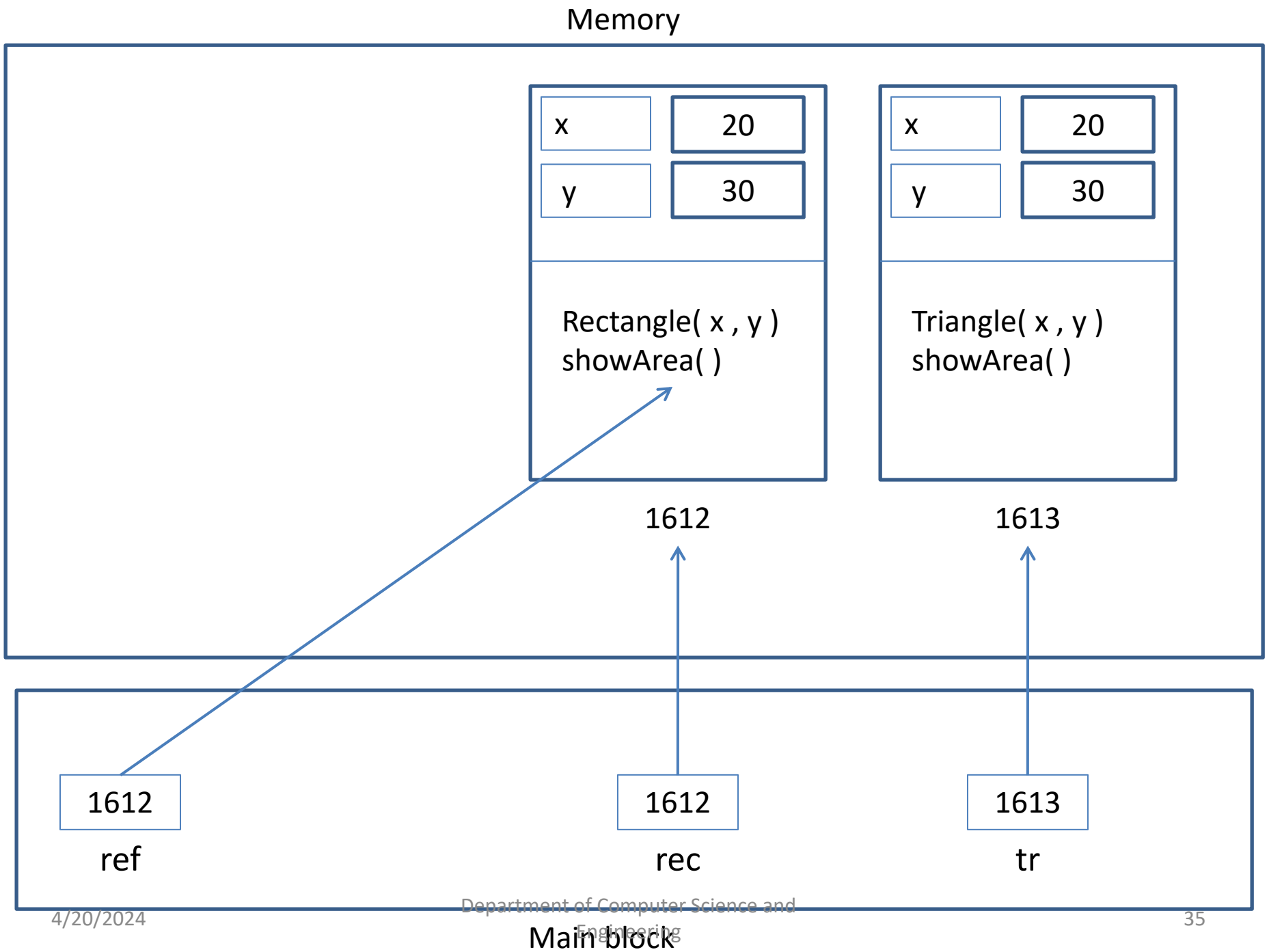
- Abstract classes cannot be instantiated

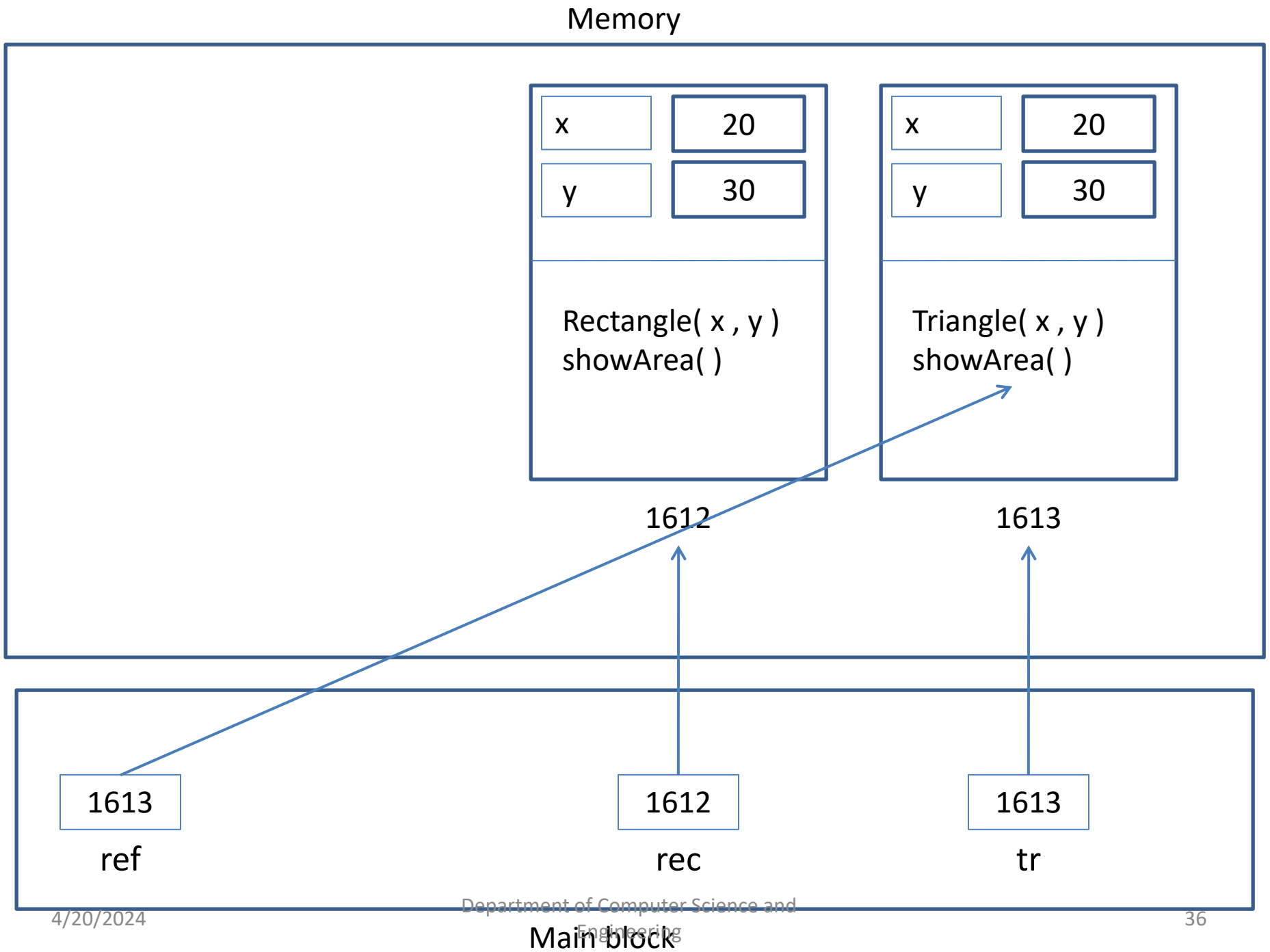
- When an abstract class is sub-classed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, the subclass must also be declared abstract. otherwise it generates compile-time error.

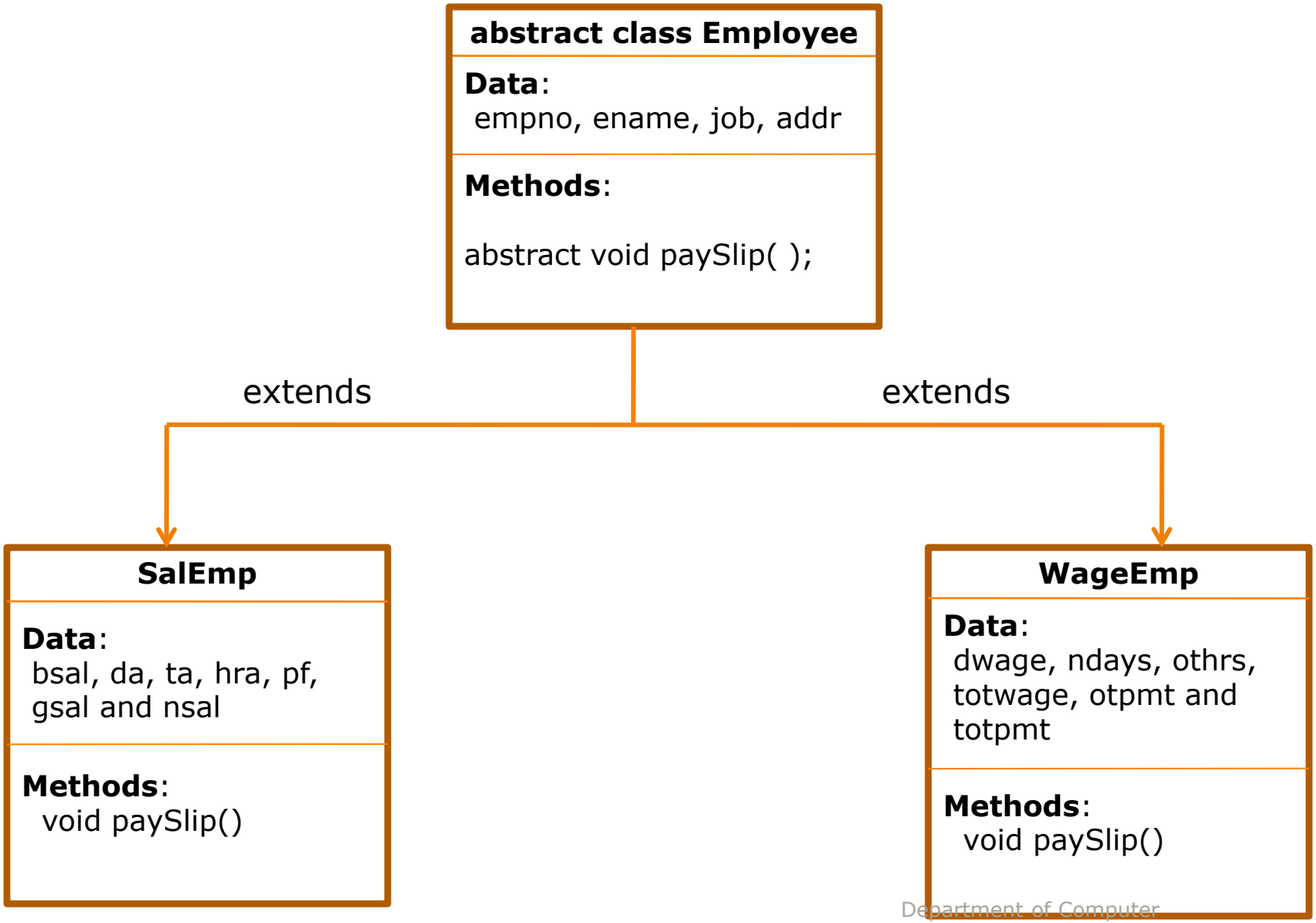
```
abstract  class Parent {

      . . . .

}


abstract class Child extends Parent {

      . . . .    //  if not implementing abstract methods

 }
```

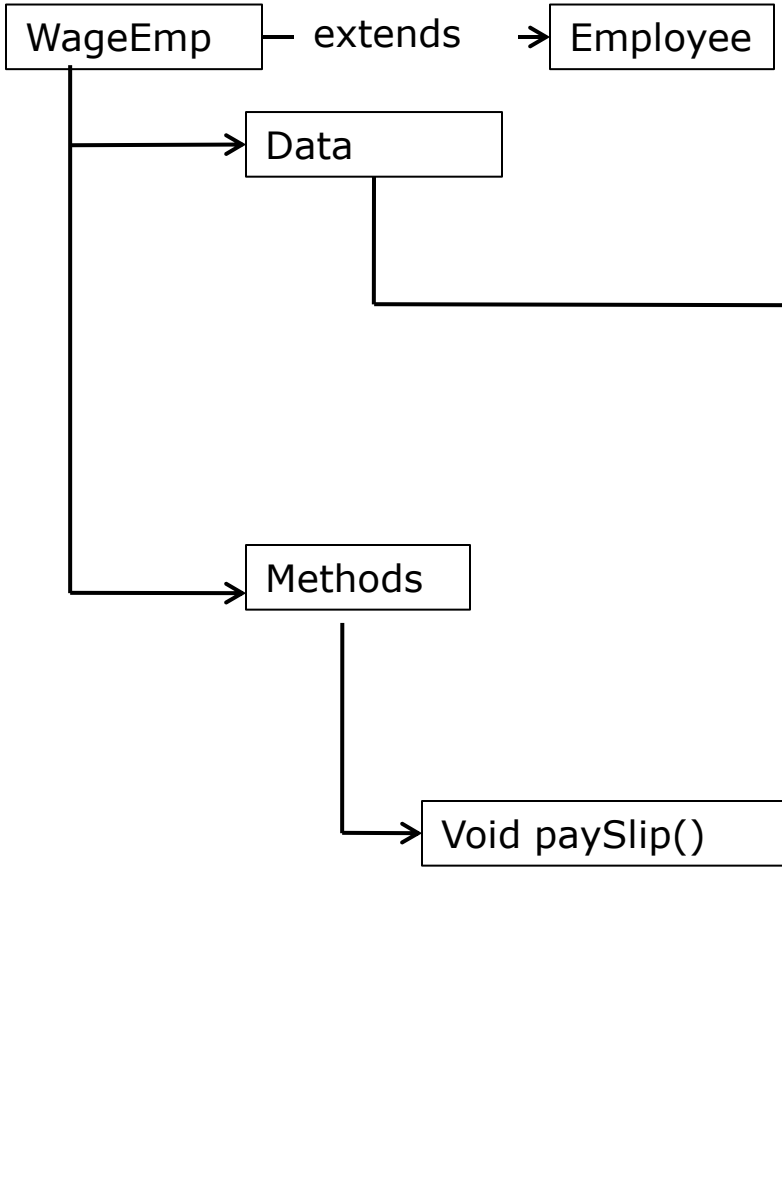

abstract Shape
Data
x
double
y
double
Constructors
Shape(int x, int y)
this.x = x;
this.y = y;
Methods
abstract void showArea()
Rectangle
extends
Shape
Constructors
Rectangle(int x, int y)
super(x, y);
Methods
void showArea()
Triangle
extends
Shape
Constructors
Triangle(int x, int y)
super(x, y);
Methods
void showArea()


Memory
x
20
y
30
Rectangle(x , y)
showArea( )
x
20
y
30
Triangle(x , y)
showArea( )
1612
1613
null
1612
1613
ref
rec
tr
Main block


Memory
x
20
y
30
Rectangle(x , y)
showArea( )
x
20
y
30
Triangle(x , y)
showArea( )
1612
1613
1612
1612
1613
ref
rec
tr
Main block


Memory
x
20
y
30
Rectangle(x , y)
showArea( )
x
20
y
30
Triangle(x , y)
showArea( )
1612
1613
1613
1612
1613
ref
rec
tr
Main block


abstract class Employee
Data:
empno, ename, job, addr
Methods:
abstract void paySlip( );
extends
extends
SalEmp
Data:
bsal, da, ta, hra, pf,
gsal and nsal
Methods:
void paySlip()
WageEmp
Data:
dwage, ndays, othrs,
totwage, otpmt and
totpmt
Methods:
void paySlip()



Variables	Data Type
dwage	double
ndays (no of days)	int
othrs (overtime hrs)	double
totwage	double
otpmt (overtime pmt)	double
totpmt (total pmt)	double

Read:
dwage, ndays and othrs

totwage = dwage x ndays
otpmt = ((dwage / 8) x 2) x othrs
totpmt = totwage + otpmt

Print: totwage, otpmt, and totpmt

Object class:

- There is a class with name Object in java.lang package.
- It is the super class of all classes in java.
- Every class in java is a direct or indirect sub class of the Object class.
- To convert an object to String, compare objects .
- It can store any reference of any object.
- Object class reference can store any reference of any object.
-

Methods of Object class:

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

Cloning objects:

- The **object cloning** is a way to create exact copy of an object.
- The clone() method of Object class is used to clone an object.
- The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create.
- If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

String handling:

- It plays a very important role. Most of the data that transmits on internet will be in the form of String (group of characters).
- In java it is not a character array terminated by NULL (‘\0’) operator like C and C++. Its an object of String class.
- Represented by the java.lang.String class
- String characteristics
  - Reference type
  - Immutable
  - final
- Each character is represented by java.lang.Char (char)
  - Uses UTF-16 encoding

Creating Strings

- Using a String literal
- Using a Constructor
- Calling toString( ) on another object.

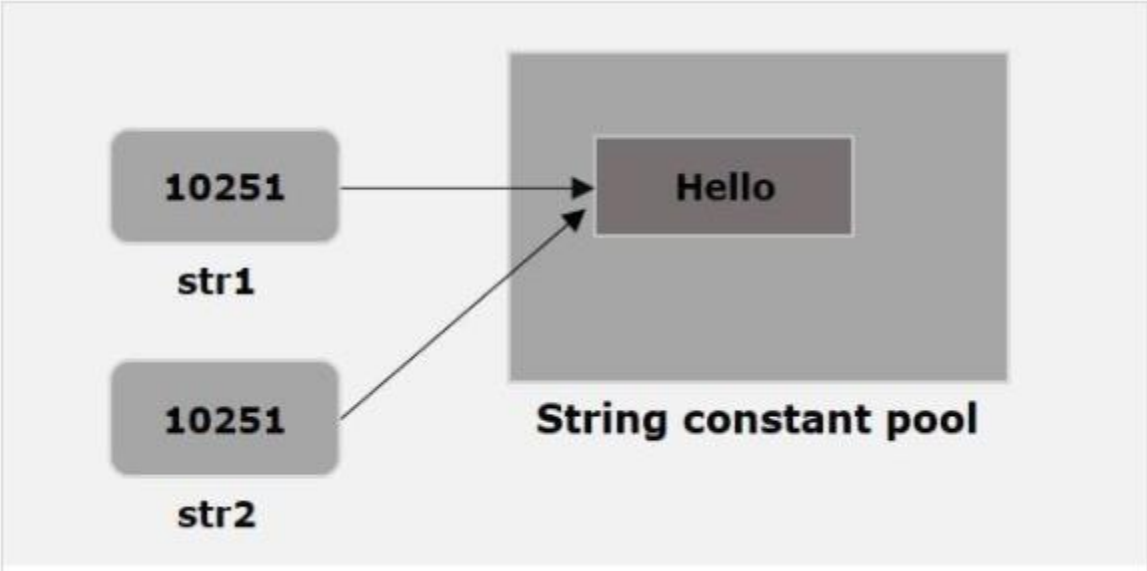
```
String str ;
str   = "Hello";                (or)
String  str  = "Hello";
```

```
String str = new String( "Hello" );
```

```
char arr [ ] = { 'C','O','M','P','U','T','E','R' };
String str = new String( arr );
String str = new String( arr , 2 , 4);
```

String constant pool

- JVM creates a String object with the given value in a separate block of memory known as String constant pool.



String Length

- Accessible through the length ( ) method, which returns the number of characters contained in the String object.

```
public class StringDemo {
    public static void main ( String args[ ] ) {
        String str = “VITAP” ;
        int  len = str.length();
        System.out.println( "String Length is : " + len );
    }
}
```

String Concatenation

- Java Strings can be concatenated (joined) using the **+** and **+=** operators to create new Strings.
- Every time an operation modifies a String object, a new read-only String object is created.
- Using **concat ( )** method

```
String language = " Java ";
String course = "Introduction to " + language ;
course += " Programming ";
System.out.println( course );      // Introduction to Java Programming
```

```
String  str1 = " Hello ";
String  str2 = " World ";
String  str3 = str1.concat(str2);
System.out.println(str3) ;          //  Hello World
```

Comparing Strings

- Strings are compared to determine equality and for sorting
- Java provides variety of methods to compare String objects
- Use of **= =** operator only tests whether two String object references are same or not

Method	Description
int compareTo (String)	Compares two strings lexicographically and returns int value (0 , >0 , <0)
int compareToIgnoreCase (String)	Compares two strings, ignoring case differences.
boolean equals (String)	Compares two strings and returns true or false.
boolean equalsIgnoreCase (String)	Compares two strings, ignoring case differences.

String Comparison Methods

Method	Description
boolean startsWith (String prefix)	Tests whether the current string starts with specified prefix or not
boolean endsWith (String suffix)	Tests whether the current string ends with specified suffix or not
int indexOf (String)	Returns the index of the first occurrence of the specified string.
int lastIndexOf (String)	Returns the index of the last occurrence of the specified string, searching backward.

String Manipulation Methods

Method	Description
String toLowerCase ( ) String toUpperCase ( )	Transforms the String into either upper or lower case
String replace (char old , char new)	Replaces old character to new character
String substring (int beginIndex) String substring (int beginIndex, int endIndex)	Returns to the index of the string to end.
String [ ] split (String regex)	Splits the current string in to string array.
String trim ( )	Returns the string, with leading and trailing whitespace omitted.
char charAt (int index)	Returns the character at the specified index.

String Manipulation Methods

Usage	Prints
String str = "Hello world!";	
System.out.println(str.toLowerCase());	hello world!
System.out.println(str.toUpperCase());	HELLO WORLD!
str.replace("Hello", "Good morning‟);	Good morning world!
System.out.println(str.substring(0, 5));	Hello

String list = "1,2,3,4,5";
String [ ] listItems = list.split(',');
for (String item : listItems) {
System.out.println(item);
}
	1
	2
	3
	4
	5

StringBuilder or StringBuffer

- String is immutable.
- StringBuilder and StringBuffer is mutable.
- StringBuilder is not always more efficient than string.
- These classes provide methods which can modify the content of the objects directly.
- Some methods insert(), delete() and reverse() which are not available in String class.
- StringBuffer class will take more execution time than the StringBuilder.

StringBuilder Methods

Method	Description
StringBuilder append (String)	It is used to append the specified string with this string.
void insert (int index , String)	It is used to insert the specified string with this string at the specified position.
void reverse ( )	It is used to reverse the string.
void delete (int start , int end)	It is used to delete the string from specified startIndex and endIndex.
void setCharAt (int index , char ch)	Replace char at index position
String toString ( )	The toString() method returns the String representation of the object.

The main difference between the StringBuffer and StringBuilder is that StringBuilder methods are not thread safe(not Synchronized).

Type casting

- Converting from one data type to another data type is called type casting.
- If the two types are compatible, then Java will perform the conversion automatically.
- For example, it is always possible to assign an **int value to a long variable.**
- There is no automatic conversion defined from **double to byte.**

Java's Automatic Conversions:

✓ The two types are compatible.
✓ The destination type is larger than the source type.

When these two conditions are met, a ***widening*** *conversion takes place.*

byte -> short -> char -> int -> long -> float -> double

❑ However, there are no automatic conversions from the numeric types to **char or boolean. Also, char and boolean are not compatible with each other.**

Casting Incompatible Types

- For example, what if you want to assign an **int value to a byte variable?**
- This conversion will not be performed automatically, because a **byte is smaller** than an **int.**
- This kind of conversion is sometimes called a ***narrowing*** *Conversion.*

double -> float -> long -> int -> char -> short -> byte

- b = (byte) a; // (byte) is casting operator.

Using Nested / Inner Class

If you declared a class with in the another class then it is Nested / Inner class

- Nested class can be static
- Nested class can be non-static i.e., Inner Class

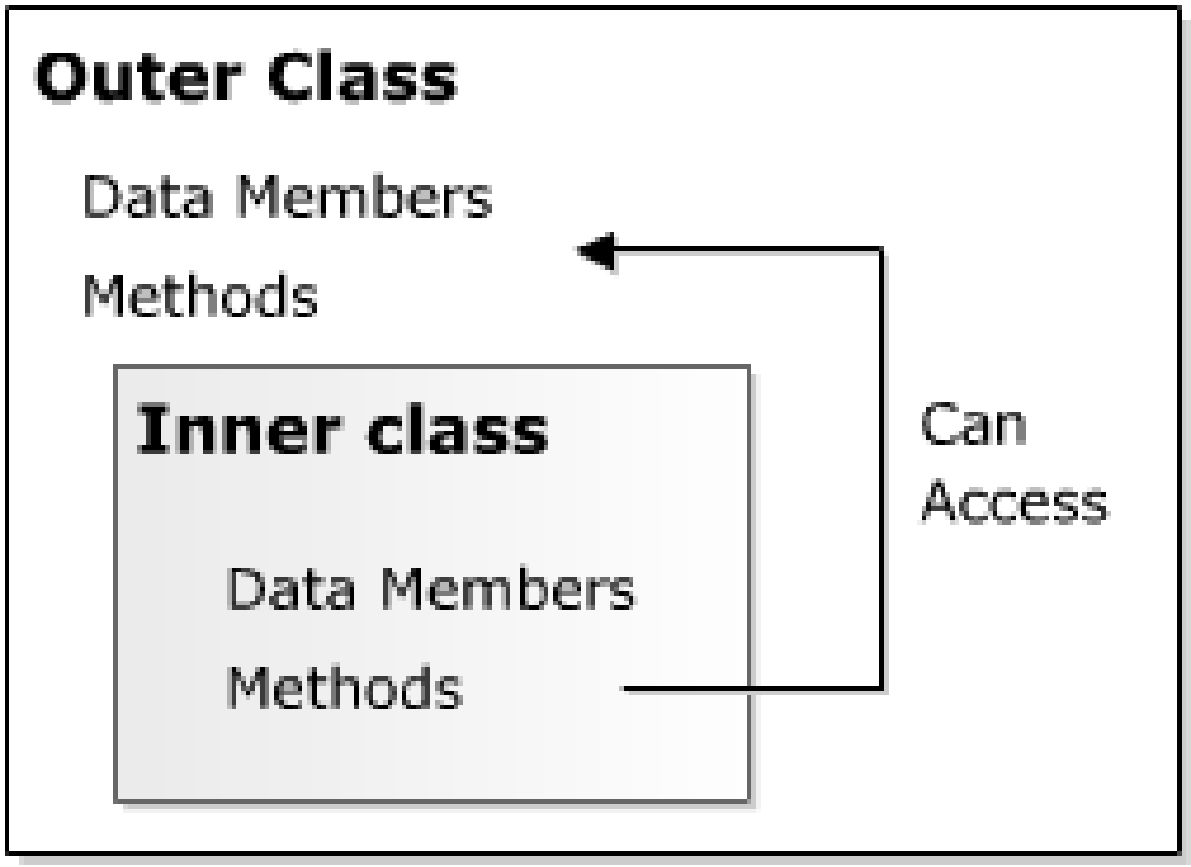
Inner class can access the outer class members directly including private members.

For Nested class to access the outer class members, you need to create an Outer class object.

The relation between inner and outer class is 'has-a' (Composition).

Can Class be declared static?

Generally, A Class cannot be declared static. However, if it is a Nested class then it can be declared static.

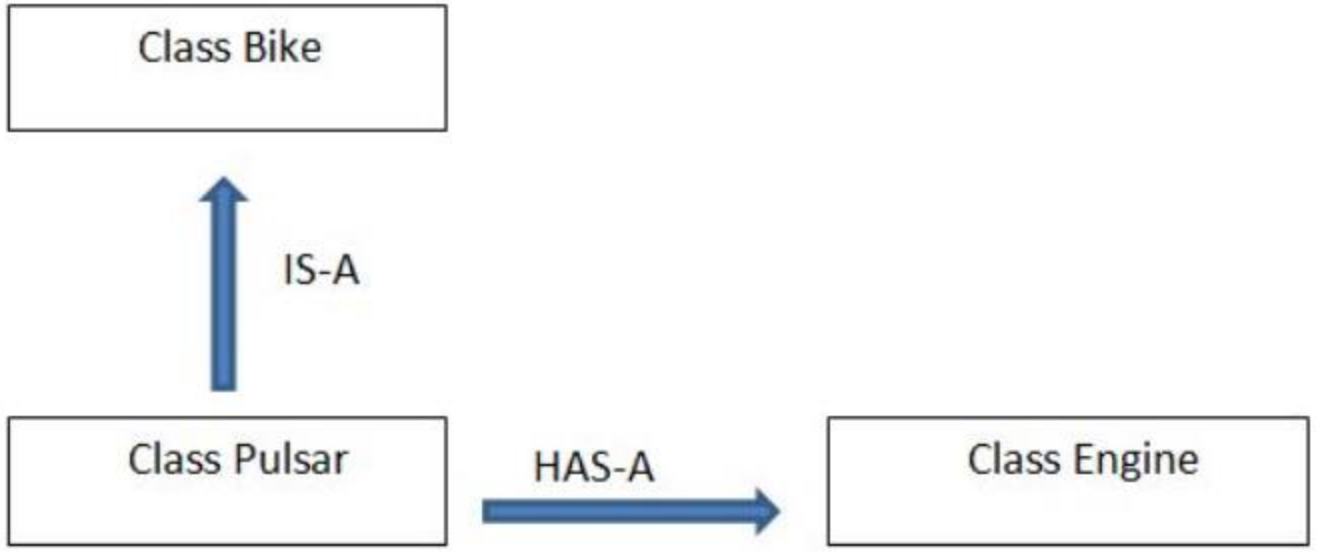

Outer Class
Data Members
Methods
Inner class
Data Members
Methods
Can
Access

Forms of Inheritance

- One of the main purposes is substitutability.
- The substitutability means that when a child class acquires properties from its parent class, the object of the parent class may be substituted with the child class object.
- The following are the different forms of inheritance in java.

❖Specialization
❖Specification
❖Construction
❖Extension
❖Limitation
❖Combination

IS-A & Has-A relation



An Is-A relationship is also known as inheritance and a Has-A relationship is also known as composition in Java.

Specialization

It is the most ideal form of inheritance. The subclass is a special case of the parent class. It holds the principle of substitutability.

Specification

The parent class just specifies which methods should be available to the child class but doesn't implement them.

Construction

The child class may change the behavior defined by the parent class (overriding).

Extension

The child class may add its new properties.

Limitation

The subclass restricts the inherited behavior.

Combination

The subclass inherits properties from multiple parent classes.
(Not supported by java)

Benefits and Costs of Inheritance

Benefits

- Inheritance helps in code reuse. The child class may use the code defined in the parent class without re-writing it.

- Inheritance can save time and effort as the main code need not be written again.

- Inheritance provides a clear model structure which is easy to understand.

- An inheritance leads to less development and maintenance costs.

- With inheritance, we will be able to override the methods of the base class so that the meaningful implementation of the base class method can be designed in the derived class.

- In inheritance base class can decide to keep some data private so that it cannot be altered by the derived class.

Costs of Inheritance

- Inheritance decreases the execution speed due to the increased time and effort it takes, the program to jump through all the levels of overloaded classes.
- Inheritance makes the two classes (base and inherited class) get tightly coupled. This means one cannot be used independently of each other.
- The changes made in the parent class will affect the behavior of child class too.
- The overuse of inheritance makes the program more complex

Expressions

- Every expressions consists of at least one operator and an operand.
- Operand can be either a literal, variable or a method invocation.
- Expression evaluation in Java is based upon the following concepts:
- Type promotion rules
- Operator precedence
- Associativity rules

Promotion rules

✓ First, all **byte, short, and char values are promoted to int.**
✓ If one operand is a **long, the whole expression is promoted to long.**
✓ If one operand is a **float, the entire expression is promoted to float.**
✓If any of the operands are **double, the result is double.**

Java Operators Precedence and Associativity

- The following table describes the allowable operators, their precedence, and associativity.

Category (by precedence)	Operator(s)	Associativity
Unary	+ - ! ~ ++x --x (T)x	right
Multiplicative	* / %	left
Additive	+ -	left
Shift	<< >>	left
Relational	< > <= >= is as	left
Equality	== !=	left
Logical AND	&	left
Logical XOR	^	left
Logical OR	|	left
Conditional AND	&&	left
Conditional OR	||	left
Null Coalescing	??	left
Ternary	?:	right
Assignment	= *= /= %= += -= <<= >>= &= ^= |= =>	right

Garbage collection:

Garbage collection in Java is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

Unreachable objects: An object is said to be unreachable if it doesn't contain any reference to it. Also, note that objects which are part of the island of isolation are also unreachable.

There are generally four ways to make an object eligible for garbage collection.

1. Nullifying the reference variable
2. Re-assigning the reference variable
3. An object created inside the method
4. Island of Isolation

Finalization:

Just before destroying an object, Garbage Collector
calls finalize() method on the object to perform cleanup activities.
Once finalize() method completes, Garbage Collector destroys that object.
finalize() method is present in Object class with the following prototype.

```
protected void finalize() throws Throwable
```

Ways for requesting JVM to run Garbage Collector

Once we make an object eligible for garbage collection, it may not destroy immediately by the garbage collector. Whenever JVM runs the Garbage Collector program, then only the object will be destroyed. But when JVM runs Garbage Collector, we can not expect.

1. Using System.gc() method: System class contain static method gc() for requesting JVM to run Garbage Collector.
2. Using Runtime.getRuntime().gc() method: Runtime class allows the application to interface with the JVM in which the application is running. Hence by using its gc() method, we can request JVM to run Garbage Collector.

Overview of Interface:

- An Interface contain a set of Abstract methods and Static data members
- Interface is known as a prototype for a class.
- Methods defined in an Interface are only abstract methods. (from jdk 1.8 default, static and private methods also allowed).
- We use the keyword "implements" to inherit interface
- You can implement multiple interfaces in a single class.
- Interfaces cannot be instantiated

```
interface < interface-name > {
        static final data members ;
         public return-type methods (parameters);
  }
```

```
interface A {
        . . .
 }

class B  implements  A {
        . . .   override the methods of A interface
 }
```

```
interface A {              interface B {
    . . .                      . . .
}                          }

class C implements A,B{
        . . .  override both A and B methods
}
```

```
interface A {
        . . .
}
interface B extends A {
        . . .
}
class C implements B {
        . . .      override both A and B methods
 }
```

class	→ extends	→	class
class	→ implements	→	interface
interface	→ extends	→	interface

- When a class implements interface, the class usually provides implementations for all of the methods of that interface. However, if it does not, the class must also be declared abstract. otherwise it generates compile-time error.

```
interface A {
    . . .
    . . .
}

abstract class B  implements  A {
   . . .
    . . .  in case of not overriding
}
```

- Class extending Super class and implementing Interface

```
interface A {
    . . .
    . . .
}
class  Sample {
    . . .
    . . .
 }
class  Demo  extends  Sample  implements  A {
    . . .
    . . .
}
```

Abstract classes	Interfaces
An abstract class is written when there are some common features shared by all the objects.	An interface is written when all the features are implemented differently in different objects.
An abstract class contains instance methods and variables.	An interface contain only constants and abstract methods. (jdk 1.8 version onwards static private and default methods also allowed)
Abstract classes have constructors.	Interfaces can not have constructors.