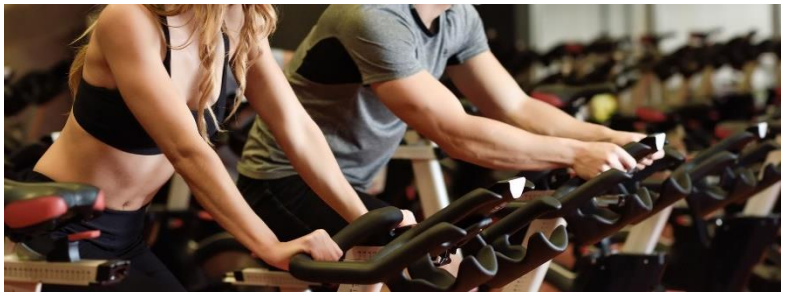
IECS273/274 [1111 3670/3671]
物件導向設計與實習

06#1 OOP Fundamentals Inheritance



01 Inheritance

Inheritance

- Object-oriented programming (OOP) **inheritance** improves **software reusability**.
- ***Inheritance*** *enables you to define a general class (i.e., a superclass, parent class) and later extend it to more specialized classes (i.e., subclasses, derived classes, child classes).*
- **Inheritance** means that a very general form of a class can be defined and compiled. More specialized versions of that class may be defined by starting with the already defined class and adding more specialized instance variables and methods. The specialized classes are said to *inherit* the methods and instance variables of the previously defined general class.
- The sharing of attributes and operations among classes based on a hierarchical **relationship**.

Relationships

Inheritance

- ✓ New classes created from existing classes
- ✓ Derived class is a class that inherits data members and member functions from a previously defined base class
- ✓ **Single Inheritance**
  - Class inherits from one base class
- ✓ **Multiple Inheritance**
  - Class inherits from multiple base classes
- ✓ Three types of inheritance:
  - **public**: Derived objects are accessible by the base class objects
  - **private**: Derived objects are inaccessible by the base class
  - **protected**: Derived classes and friends can access protected members of the base class
- ✓ **Is-a relationship (generalization)**

Relationships

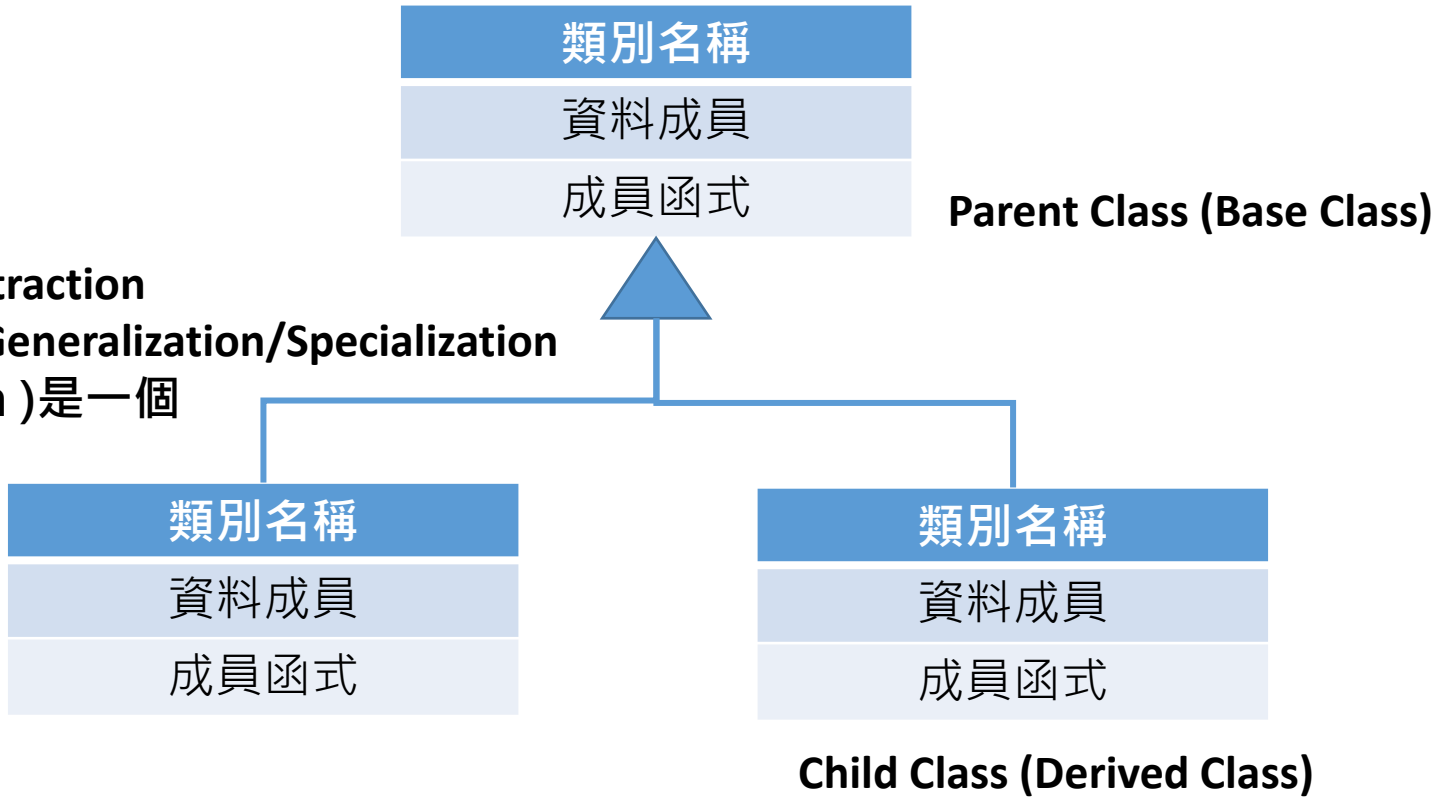
- An **association** is a bi-directional connection between classes

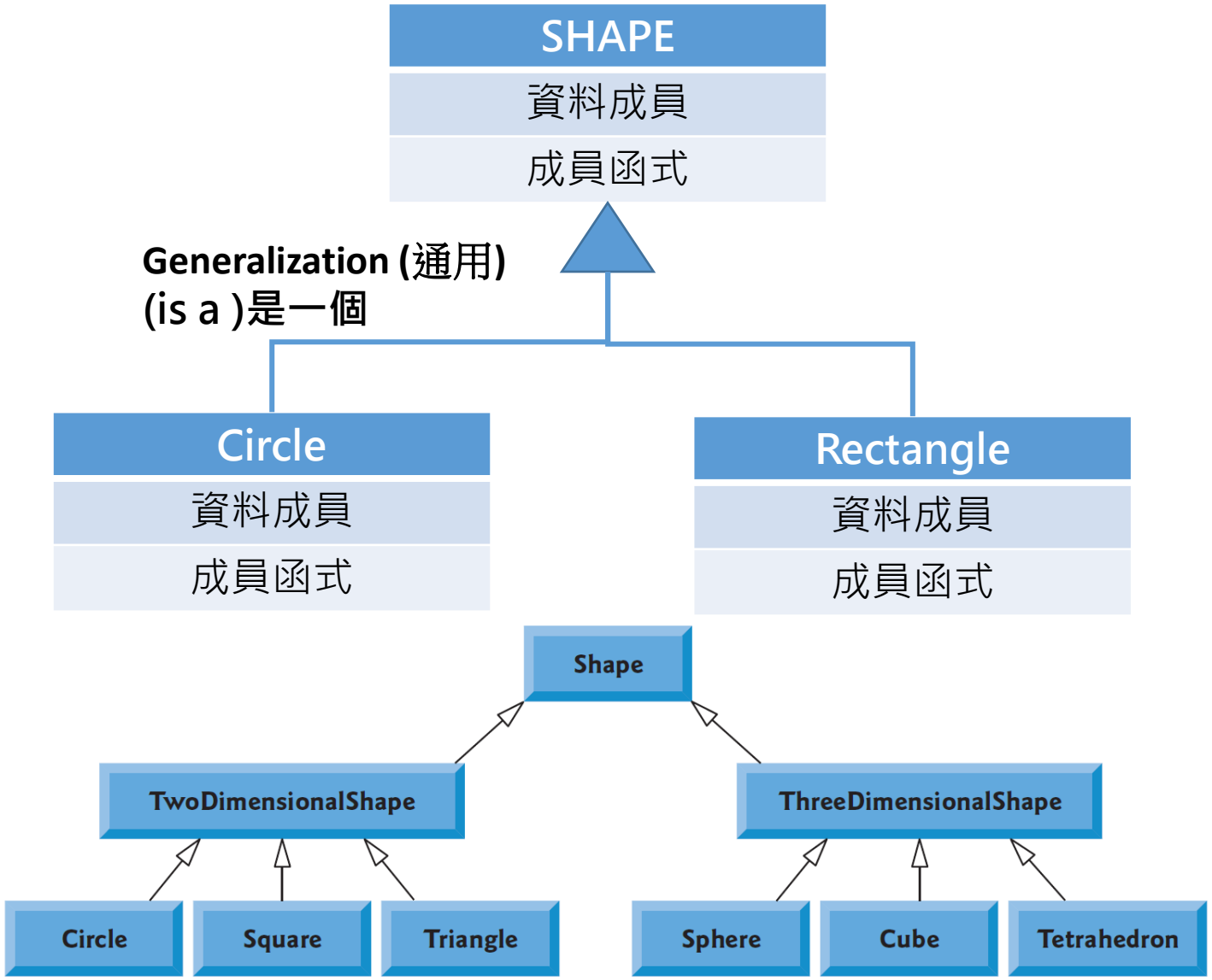
- An **aggregation** is a stronger form of association where the relationship is between a whole and its parts

- A **composition** is a stronger form of aggregation where the part is contained in at most one whole and the whole is responsible for the creation of its parts

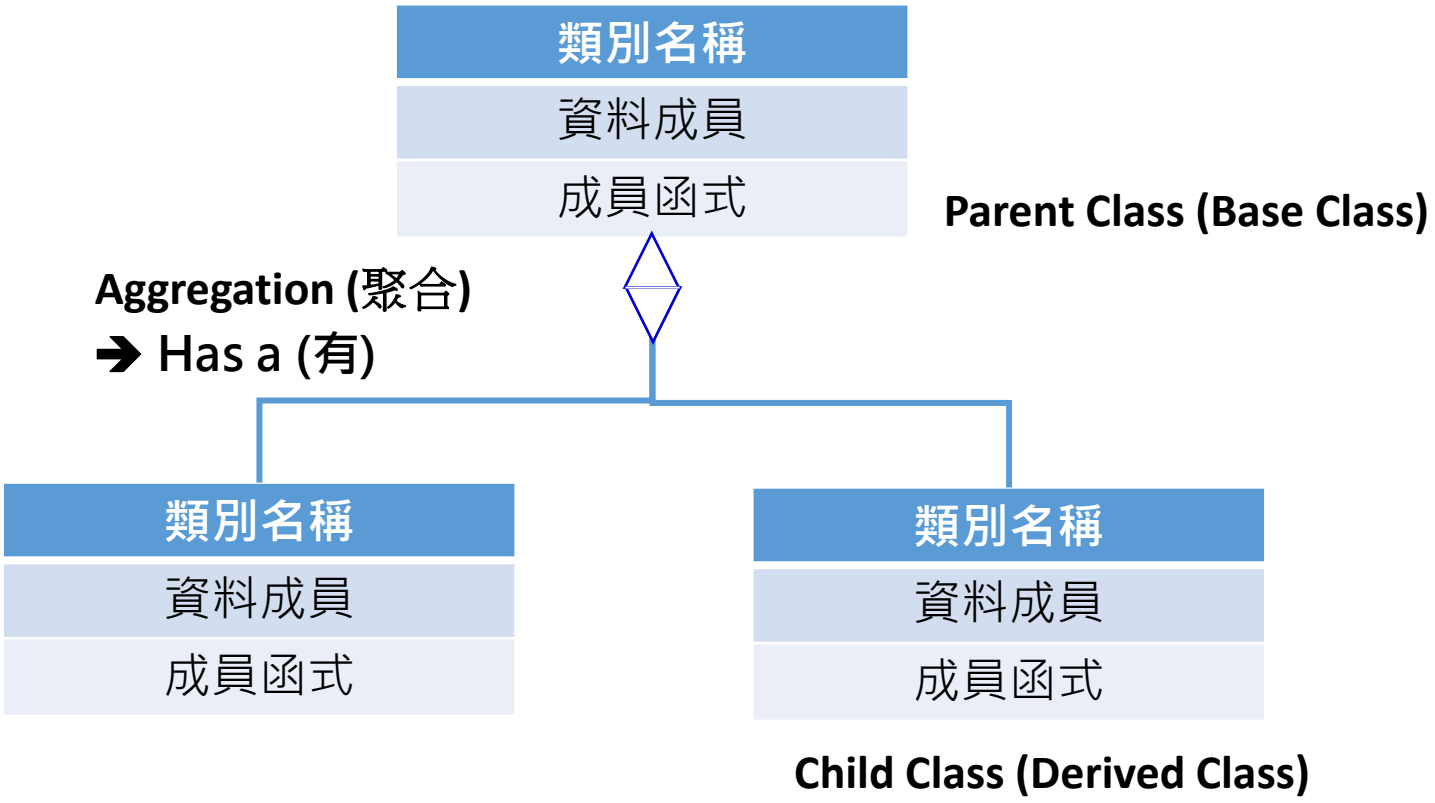
- A **dependency** is a weaker form of relationship showing a relationship between a client and a supplier where the client does not have semantic knowledge of the supplier

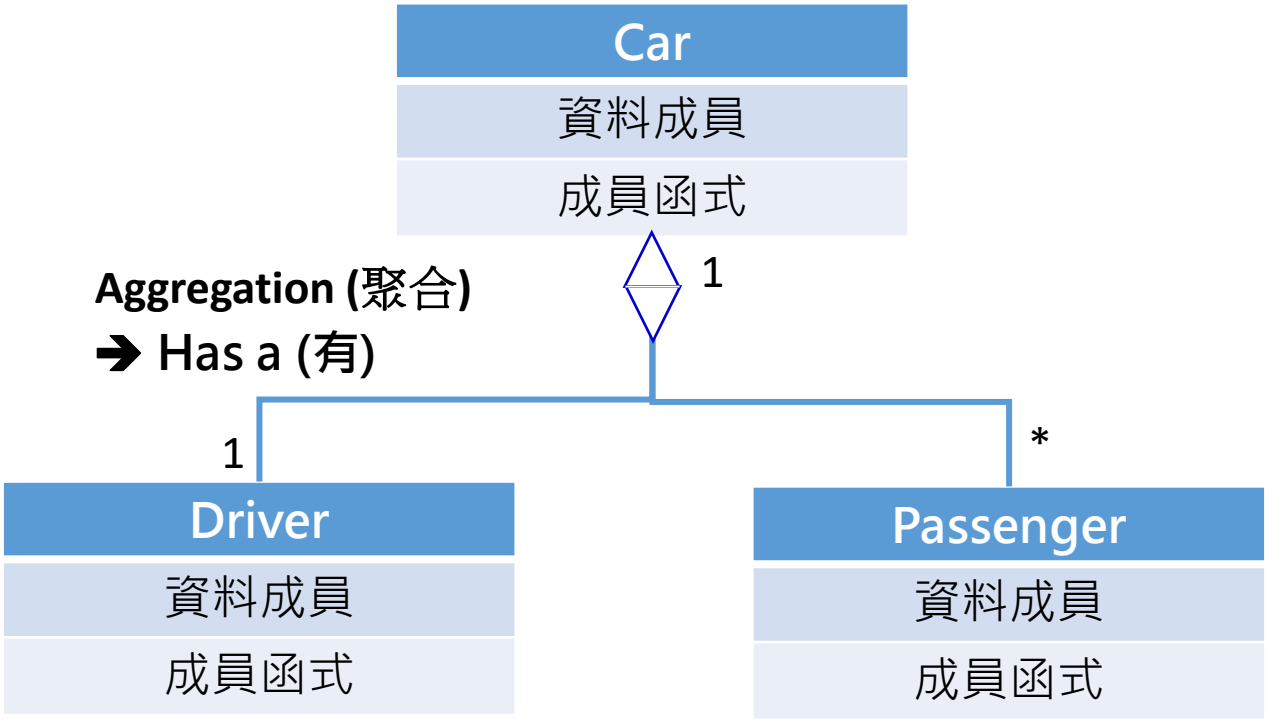
① Generalization




SHAPE
資料成員
成員函式
Generalization (通用)
(is a)是一個
Circle
資料成員
成員函式
Rectangle
資料成員
成員函式
Shape
TwoDimensionalShape
ThreeDimensionalShape
Circle
Square
Triangle
Sphere
Cube
Tetrahedron

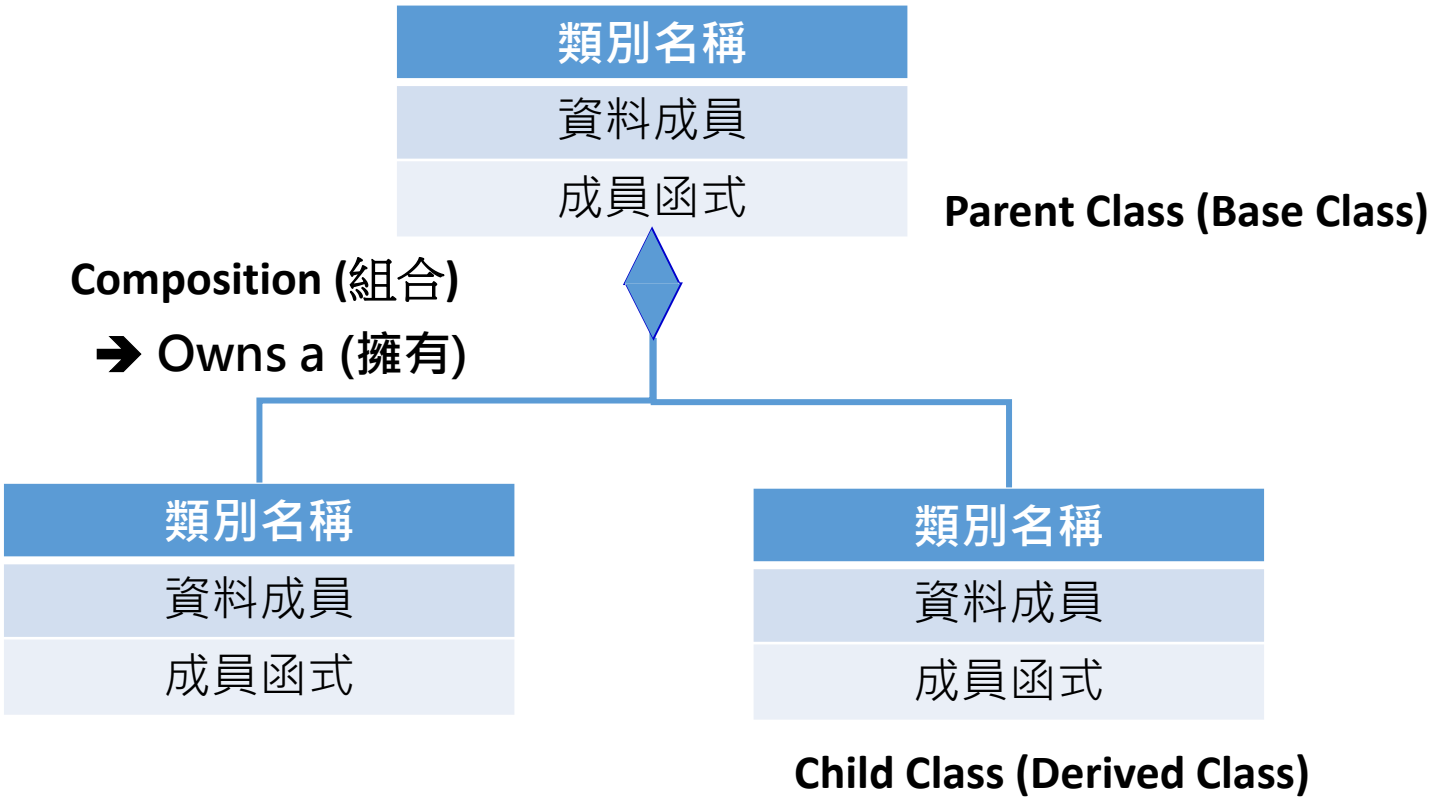
② Aggregation

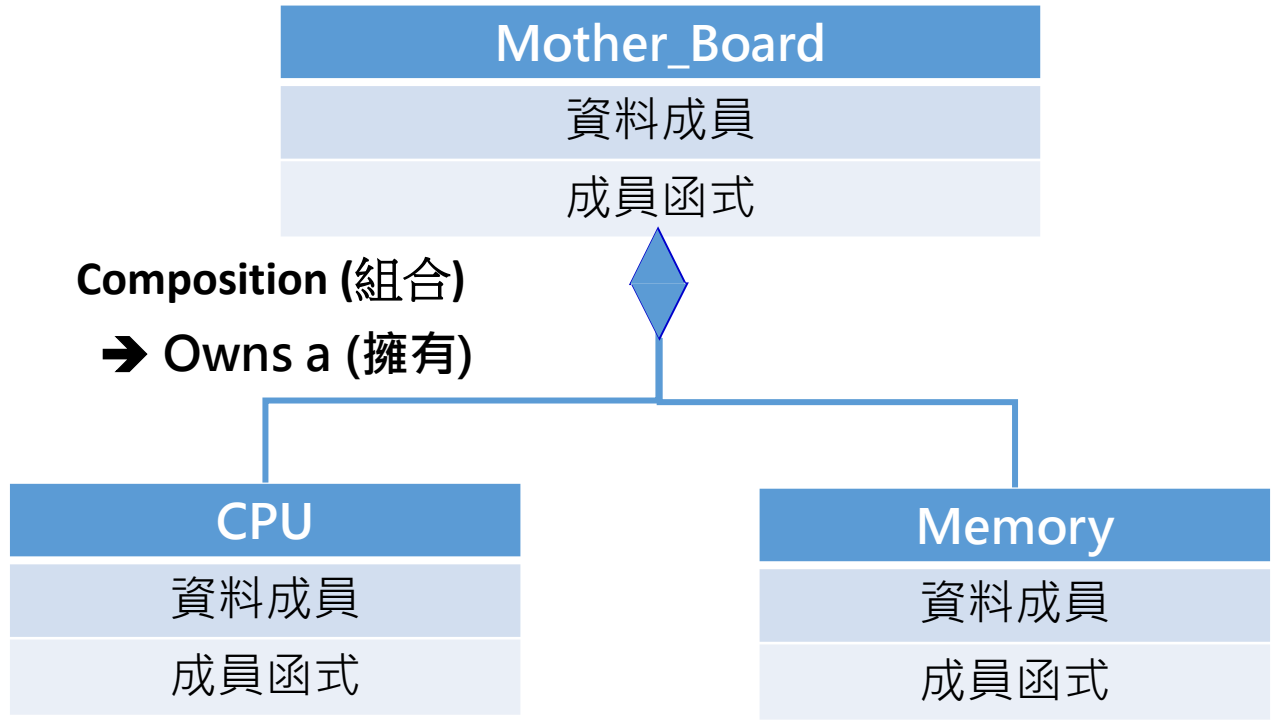




- 父類別並不負責新增或消滅子類別
  - 指標變數自己的生命週期
  - 當物件Car被消滅時，物件Driver與Passenger仍然可以存在

③ Composition





- 父類別負責產生和消滅子類別
  - 變數與擁有者生命週期相同
  - 當物件Mother_Board被消滅，CPU與Monitor也被消滅

Access Mode - public

➢以public屬性繼承

```
public class 子類別名稱 extends 父類別名稱{
        public:
        資料成員 / 成員函式 .....
        protected:
        資料成員 / 成員函式 .....
        private:
        資料成員 / 成員函式.....
};
```

父類別內的	將成為子類別的
public區域的成員	public區域的成員
protected區域的成員	protected區域的成員
private區域的成員	無法直接使用

Access Mode - protected

➢以protected屬性繼承

```
protected class 子類別名稱 extends 父類別名稱{
        public:
        資料成員 / 成員函式 .....
        protected:
        資料成員 / 成員函式 .....
        private:
        資料成員 / 成員函式.....
};
```

父類別內的	將成為子類別的
public區域的成員	protected區域的成員
protected區域的成員	protected區域的成員
private區域的成員	無法直接使用

Access Mode - protected

➢以private屬性繼承

```
private class 子類別名稱 extends 父類別名稱{
        public:
        資料成員 / 成員函式 .....
        protected:
        資料成員 / 成員函式 .....
        private:
        資料成員 / 成員函式.....
};
```

父類別內的	將成為子類別的
public區域的成員	private區域的成員
protected區域的成員	private區域的成員
private區域的成員	無法直接使用

02

Inheritance in Java

Characteristics of Java Inheritance

- **Inheritance** is a procedure for a **subclass** (*child class*) to obtain all the characteristics of a **superclass** (*parent class*), including *data members and member functions*.
- The Java programming language uses the keyword **extends** to indicate inheritance。Syntax:

  class SuperClass {…}

  class SubClass **extends** SuperClass {…}
- Every class in Java is descended from the **java.lang.Object class**. If no inheritance is specified when a class is defined, the superclass of the class is **Object** by default.

Characteristics of Java Inheritance

- When creating a SubClass object, the Java system creates memory for all SuperClass data members.
- Overriding method: Subclasses can override superclass methods.
  - If the subclass wants to call the overloaded method of the superclass, add the keyword **super**. Example: **super**.methodName.
- Use the **instanceof** keyword to test whether an object is an instance of a class. E
  - (obj **instanceof** Cls) returns **true** if obj is an instance of Cls; otherwise, returns **false**.
- **Explicit casting**: Convert superclass reference to a subclass reference (downcasting).
  - Can only be done when superclass reference **actually referring to a subclass object**.

The Class Object

- In Java, every class is a descendent of the class **Object**
  - Every class has Object as its ancestor
  - Every object of every class is of type Object, as well as being of the type of its own class
- If a class is defined that is not explicitly a derived class of another class, it is still automatically a derived class of the class Object
- Having an Object class enables methods to be written with a parameter of type Object
  - A parameter of type Object can be replaced by an object of any class
  - For example, some library methods accept an argument of type Object so they can be used with an argument that is an object of any class
- The class Object has some methods that every Java class inherits
  - For example, the equals and toString methods
- Every object inherits these methods from some ancestor class
  - Either the class Object itself, or a class that itself inherited these methods (ultimately) from the class Object

Methods of Object

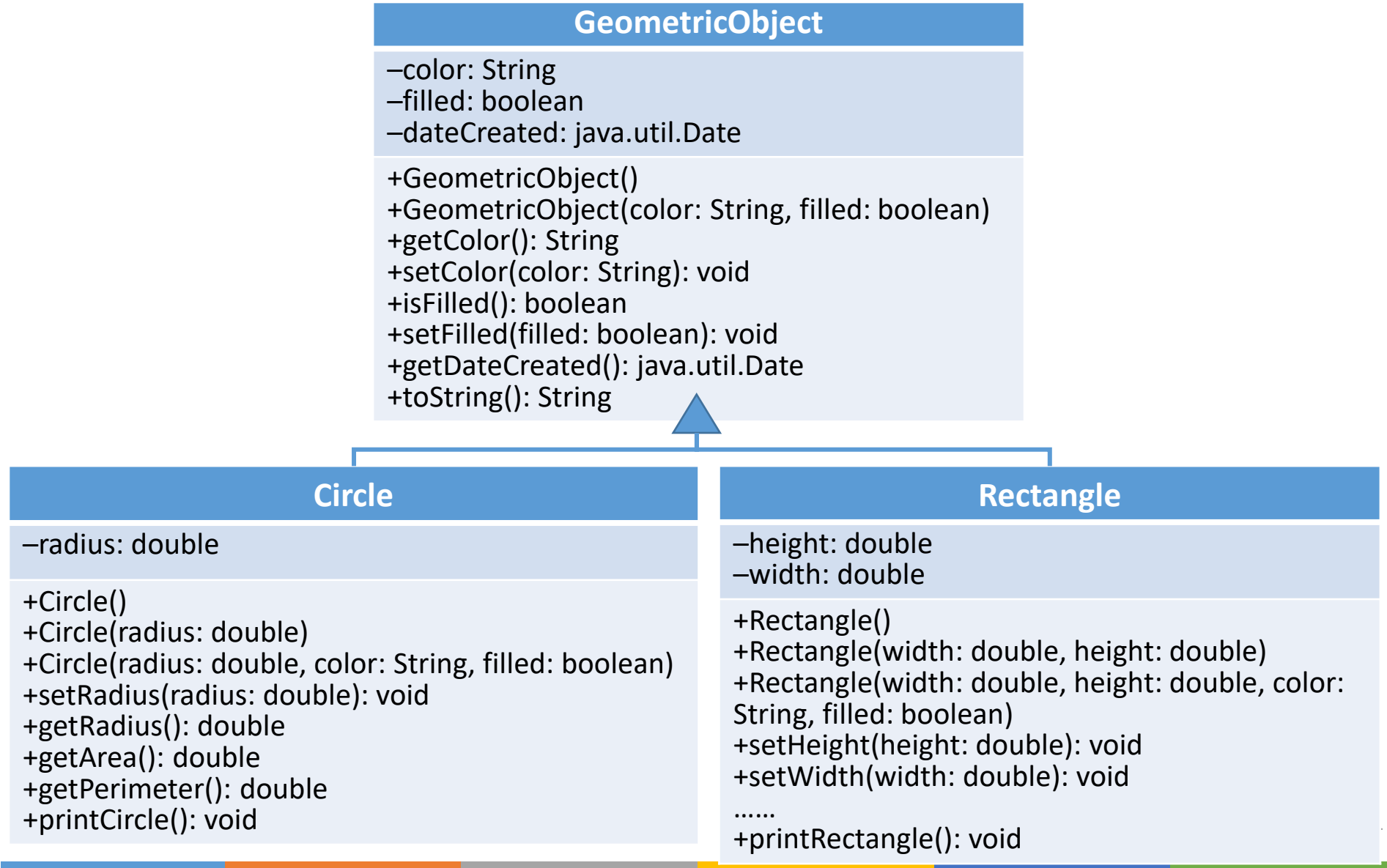
- The **Object** class, in the java.lang package, sits at the top of the class hierarchy tree. Every class is a descendant, direct or indirect, of the Object class. Every class you use or write inherits the instance methods of Object.
- The methods inherited from Object that are discussed in this section are:
  - protected Object clone() throws CloneNotSupportedException
    Creates and returns a copy of this object.
  - public boolean **equals**(Object obj)
    Indicates whether some other object is "equal to" this one.
  - protected void finalize() throws Throwable
    Called by the garbage collector on an object when garbage collection determines that there are no more references to the object
  - public final Class getClass()
    Returns the runtime class of an object.
  - public int hashCode()
    Returns a hash code value for the object.
  - public String **toString**()
    Returns a string representation of the object.

Methods of Object

- The notify, notifyAll, and wait methods of Object all play a part in synchronizing the activities of independently running threads in a program, which is discussed in a later lesson and won't be covered here. There are five of these methods:
  - public final void notify()
  - public final void notifyAll()
  - public final void wait()
  - public final void wait(long timeout)
  - public final void wait(long timeout, int nanos)

Examples

- The GeometricObject class is the superclass for Circle and Rectangle.



GeometricObject.java

```
public class GeometricObject {
        private String color = "white";
        private boolean filled;
        private java.util.Date dateCreated;
        /** Construct a default geometric object */
        public GeometricObject() {
                dateCreated = new java.util.Date();
        }
        /** Construct a geometric object with the specified color and filled value */
        public GeometricObject(String color, boolean filled) {
                dateCreated = new java.util.Date();
                this.color = color;
                this.filled = filled;
        }
        /** Return color */
        public String getColor() {
                return color;
        }
        /** Set a new color */
        public void setColor(String color) {
                this.color = color;
        }
```

Refer to Y. D. Liang, Listing 11.1, *Introduction to Java Programming and Data Structures (11th Edition),* Pearson.

GeometricObject.java

```
        /** Return filled. Since filled is boolean, its getter method is named isFilled */
        public boolean isFilled() {
                return filled;
        }
        /** Set a new filled */
        public void setFilled(boolean filled) {
                this.filled = filled;
        }
        /** Get dateCreated */
        public java.util.Date getDateCreated() {
                return dateCreated;
        }
        /** Return a string representation of this object */
        public String toString() {
                return "created on " + dateCreated + "\ncolor: " + color +
                        " and filled: " + filled;
        }
}
```

CircleObject.java

```
public class Circle extends GeometricObject {
        private double radius;
        public Circle() {
        }
        public Circle(double radius) {
                this.radius = radius;
        }
        public Circle(double radius,String color, boolean filled) {
                this.radius = radius;
                setColor(color);
                setFilled(filled);
        }
        /** Return radius */
        /** Set a new radius */
        /** Return area */
        /** Return diameter */
        /** Return perimeter */
        /** Print the circle info */
        public void printCircle() {
                System.out.println("The circle is created " + getDateCreated() +
                        " and the radius is " + radius);
        }
}
```

Refer to Y. D. Liang, Listing 11.2, *Introduction to Java Programming and Data Structures (11th Edition),* Pearson.

TestCircle.java

```
public class TestCircle {
        public static void main(String[] args) {
                Circle circle = new Circle(1);
                System.out.println("A circle " + circle.toString());
                System.out.println("The color is " + circle.getColor());
                System.out.println("The radius is " + circle.getRadius());
                System.out.println("The area is " + circle.getArea());
        }
}
```

TestCircle.class

```
A circle created on Thu Feb 10 19:54:25 EST 2011
color: white and filled: false
The color is white
The radius is 1.0
The area is 3.141592653589793
```

Refer to Y. D. Liang, Listing 11.4, *Introduction to Java Programming and Data Structures (11th Edition),* Pearson.

Access Modifier

Access Modifier	Class	Package	Subclass	World
public	V	V	V	V
protected	V	V	V	X
default (package access)	V	V	X	X
private	V	X	X	X

- The **protected** modifier provides very weak protection compared to the **private** modifier
  - *It allows direct access for any programmer who defines a suitable derived class.*
  - *Therefore, instance variables should normally not be marked protected.*

Using the super Keyword

- The keyword **super** refers to the superclass and can be used to invoke the superclass's methods and constructors.
  - The syntax to call a superclass's constructor is:

    super(); or
    super(arguments);

```
public Circle(double radius, String color, boolean filled) {
        super(color, filled);
        this.radius = radius;
}
```

- The keyword super can also be used to reference a method other than the constructor in the superclass.
  - The syntax is

    super.method(arguments);

Overriding a Method Definition

- The definition of an inherited method can be changed in the definition of a derived class so that it has a meaning in the derived class that is different from what it is in the base class. This is called **overriding** the definition of the inherited method.
  - *To override a method, the method must be defined in the subclass using the same signature as in its superclass.* When a method is **overridden**, the new method definition given in the derived class has **the exact same number and types of parameters** as in the base class.
  - *Overloading means to define multiple methods with the same name but different signatures. Overriding means to provide a new implementation for a method in the subclass.* When a method in a derived class has a **different signature** from the method in the base class, that is **overloading.**

https://notfalse.net/59/override
https://notfalse.net/58/overload

Overriding a Method Definition

- the modifier **final**
  - If the modifier **final** is placed before the definition of a ***method***, then that method may not be redefined in a derived class.
  - It the modifier **final** is placed before the definition of a ***class***, then that class may not be used as a base class to derive other classes.

```
import java.util.Date;
public class Employee {
        protected String name;
        protected Date hireDate;
        public Employee(){}
        public Employee(String theName, Date theDate){
                name = theName;
                hireDate = theDate;
        }
        public Date getHireDate(){
                return hireDate;
        }
        final public String getName(){
                return name;
        }
}
```

Overriding a Method Definition

- Changing the **Return Type** of an Overridden Method

For example, suppose one class definition includes the following details:

```
public class SuperClass
{
        ...
        public Employee getSomeone(int someKey)
        ...
```

In this case, the following details would be allowed in a derived class:

```
public class SubClass extends SuperClass
{
        ...
        public PartTimeEmployee getSomeone(int someKey)
        ...
```

When the method definition for getSomeone is overridden in SubClass, the returned type is changed from Employee to PartTimeEmployee.

Overriding a Method Definition

- Changing the **Access Permission** of an Overridden Method

You can change the access permission of an overridden method from **private** in the superclass to **public** in the subclass.

superclass:

***private* void** doSomething()

subclass:

***public* void** doSomething()

You cannot change permissions to make them **more restricted** in the subclass. You can change private to public, but you cannot change public to private.

Point.java

```
package shapeInheritance;

public class Point {                          // Will be the superclass
  protected int x, y; // Coordinates of the point.   // protected permissions, directly accessible by subclasses

  // Point constructor without arguments.
  public Point() {
    setPoint(0, 0);
  }

  // Point constructor with two integers.
  public Point(int a, int b) {
    setPoint(a, b);
  }

  // Set the coordinate [x, y] of the point.
  public void setPoint(int a, int b) {
    x = a;
    y = b;
  }
```

// two constructors

Point.java

```
  // Get x coordinate.
  public int getX() { return x; }

  // Get y coordinate.
  public int getY() { return y; }

  // Convert to the output string of point [x, y].
  public String toString() {                    // The public method of the superclass,  will be overridden
    return "[" + x +", " + y + "]";
  }
}
```

Circle.java

```
package shapeInheritance;

// Circle is a class inherits from class Point.
public class Circle extends Point{
   protected double radius;

   // Constructor without argument.
   public Circle() {
      // Implicitly call superclass constructor here.
      setRadius(0.0); // Set radius to 0.
   }

   public Circle(double r, int a, int b) {
      super(a, b); // Explicit call to the superclass constructor.
      setRadius(r); // Set radius.
   }

   // Set the value to radius.
   public void setRadius(double r) {
      radius = (r>=0.0 ? r : 0.0);
   }
```

// Use extends to define subclasses,
// Point is a superclass, and Circle is a subclass

// Two constructors

Circle.java

```
   // Get radius of the circle.
  public double getRadius() { return radius; }

  // Calculate area of the circle.
  public double area() { return Math.PI * radius * radius; }

  // Convert to the output string of circle of center [x, y] and radius.
  // Override method in the superclass.
  public String toString() {
    return "Center = [" + x +", " + y + "]; Radius = " + radius;
  }
}
```

// Public methods of subclasses,
// override methods of superclasses

ShapeInheritance.java

```
package shapeInheritance;

// Test of inheritance of points and circles.
public class ShapeInheritance {

   public static void main(String[] args) {
      Point p; // A point object.
      Point pointRef; // Reference to a point.
      Circle c; // A circle object.
      Circle circleRef; // Reference to a circle.
      String output; // Output string.

      p = new Point(10, 20); // Create a point of [10, 20].
      c = new Circle(5.6, 50, 50); // Create a circle of radius 5.5 and center [50, 50].

      // Output point p and circle c.
      System.out.print("Point p: " + p.toString() + "\n" + "Circle c: " + c.toString() + "\n\n");

      pointRef = c; // Assign circle to point reference.
      System.out.print("Circle c via point reference: " + pointRef.toString() + "\n\n");
```

// Create a superclass object
// Create a subclass object

// Superclass variable,
// but refer to subclass object

ShapeInheritance.java

```
    // Down casting (cast a superclass reference to a subclass data type.
    circleRef = (Circle) pointRef;          // downcasting, convert superclass reference to subclass reference
    System.out.print("Circle c via circle reference: " + circleRef.toString() + "\n");

    // Print area of the circle.
    System.out.printf("Area of circle c via circle reference: %5.2f\n\n", circleRef.area());

    if (p instanceof Circle){ System.out.println("Instance of a circle via a point."); } // instance test, p is not an
    else { System.out.println("Point is not an instance of a circle.\n"                  // object of subclass Circle
                              + "Cannot refere circle for a point."); }
  }
}
```

ShapeInheritance.class

```
Point p: [10, 20]
Circle c: Center = [50, 50]; Radius = 5.6

Circle c via point referecne: Center = [50, 50]; Radius = 5.6

Circle c via circle reference: Center = [50, 50]; Radius = 5.6
Area of circle c via circle reference: 98.52

Point is not an instance of a circle.
Cannot refere circle for a point.
```

03 Inner Class in Java

Inner Class in Java

■ The classes we've defined so far have been at the top level of the file.

However, a class can also be defined within another class, called an **inner class**.

Syntax:

```
public class ⟨OuterClass⟩ {
    …
    class ⟨InnerClass⟩ {
      …
    }
}
```

- The purpose of inner classes is to **hide a type**, and only **objects of ⟨InnerClass⟩ can be declared and used within ⟨OuterClass⟩**; however, program code outside ⟨OuterClass⟩ cannot access inner classes at all.
- Generally speaking, inner classes are **"*small*" classes**, that is, their code is very simple, and the classes are used infrequently.。

Inner Class in Java

- The inner class is classified into:
  - **Member inner class**: a class exists as a member of inside a class.
    - ✓ For code outside the outer class, only the object of the outer class can be created, and then the member inner class object can be created.
    - ✓ Member inner class object can be created in member method of outer class.
    - ✓ Cannot have **static** members, but can have **static** constants.

OuterClassMember.java

```
package innerClassMember;

public class OuterClass {
  int a = 10;
  static int b = 20;

  public class InnerClass {
    int a = 100;
    static final int b = 200;
    static final int c = 300;

    public void show() {
      System.out.println("Outer A = " + ++OuterClass.this.a);
      System.out.println("Inner A = " + ++a);
      System.out.println("Outer B = " + ++OuterClass.b);
      System.out.println("Inner B = " + b);
      System.out.println("Inner C = " + c);
    }
  }
```

top-level outer class

non-static variable

static variable

member inner class

A non-static variable with the same name as a non-static member of an outer class

static constant, with the same name as the outer class static member

static constant

inner class method

print the value of non-static variable a of the outer class

print the value of non-static variable a of the inner class

print value b of the outer class static variable

prints the value of the inner class static constant b

print the value of the inner class static constant c

OuterClassMember.java

```
  public void show() {
    InnerClass innerObj = new InnerClass();

    System.out.println("Outer A = " + ++a);
    System.out.println("Inner A = " + ++innerObj.a);
    System.out.println("Outer B = " + ++b);
    System.out.println("Inner B = " + innerObj.b);
    System.out.println("Inner C = " + innerObj.c);
  }
}
```

outer class method

declare and create inner class objects

print the value of non-static variable a of the outer class

print the value of non-static variable a of the inner class

print the value of static variable b the outer class

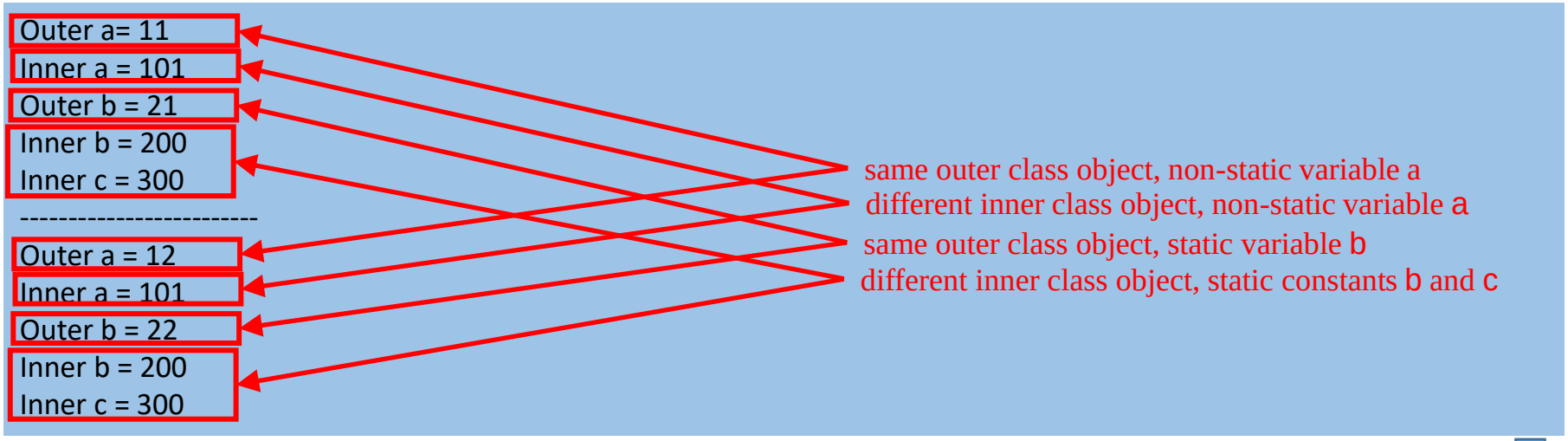
prints the value of the inner class static constant b

prints the value of the inner class static constant c

InnerClassMemberTest.java

```
package innerClassMember;

public class InnerClassTest {

   public static void main(String[] args) {
     OuterClass outerObj = new OuterClass();                    create outer class object
     OuterClass.InnerClass innerObj = outerObj.new InnerClass();   create inner class object from outer class object

     innerObj.show();                                           call inner class method.
      System.out.println("------------------------");
     outerObj.show();                                           call outer class method.

   }
}
```

InnerClassMemberTest.class



Inner Class in Java

- **Static inner class**: a class exists as a **static** member inside a class。
  - ✓ Objects of static inner classes can be *created directly* without creating objects of outer classes.
  - ✓ Members and methods of static inner classes must be **static**.
  - ✓ However, when the static inner class refers to the variables and methods of the outer class, it can only refer to the *outer static variables* and *static methods*, and not the non-static ones.

OuterClassStatic.java

```
package innerClassStatic;

public class OuterClassStatic {
  int a = 10;
  static int b = 20;

  public static class InnerClass {
    static int a = 100;
    static final int b = 200;
    static final int c = 300;

    public static void show() {
      // System.out.println("Outer a = " + ++OuterClassStatic.this.a);
      System.out.println("Inner a = " + ++a);
      System.out.println("Outer b = " + ++OuterClassStatic.b);
      System.out.println("Inner b = " + b);
      System.out.println("Inner c = " + c);
    }
  }
```

top-level outer class
non-static variable
static variable

static inner class
inner class static variable, same name as outer class non-static member
inner class static constant, same name as outer class static member
inner class static constant

inner class static method, must be static,
only static members can be accessed
cannot print non-static variable a of the outer class
print inner class static variable a
print outer class static variable b
print inner class static constant b
print inner class static constant c

OuterClassStatic.java

```
public void show() {
    InnerClass innerObj = new InnerClass();

    System.out.println("Outer a = " + ++a);
    System.out.println("Inner a = " + ++innerObj.a);
    System.out.println("Outer b = " + ++b);
    System.out.println("Inner b = " + innerObj.b);
    System.out.println("Inner c = " + innerObj.c);
  }
}
```

outer class method

declare and create inner class objects

print outer class non-static variable a

print inner class static variable a

print the outer class static variable b

print inner class static constant b

print inner class static constant c

InnerClassStaticTest.java

```
package innerClassStatic;

public class InnerClassStaticTest {
   public static void main(String[] args) {
      OuterClassStatic outerObj = new OuterClassStatic();
      OuterClassStatic.InnerClass innerObj = new OuterClassStatic.InnerClass();

      innerObj.show();
       System.out.println("------------------------");
      outerObj.show();
   }
}
```

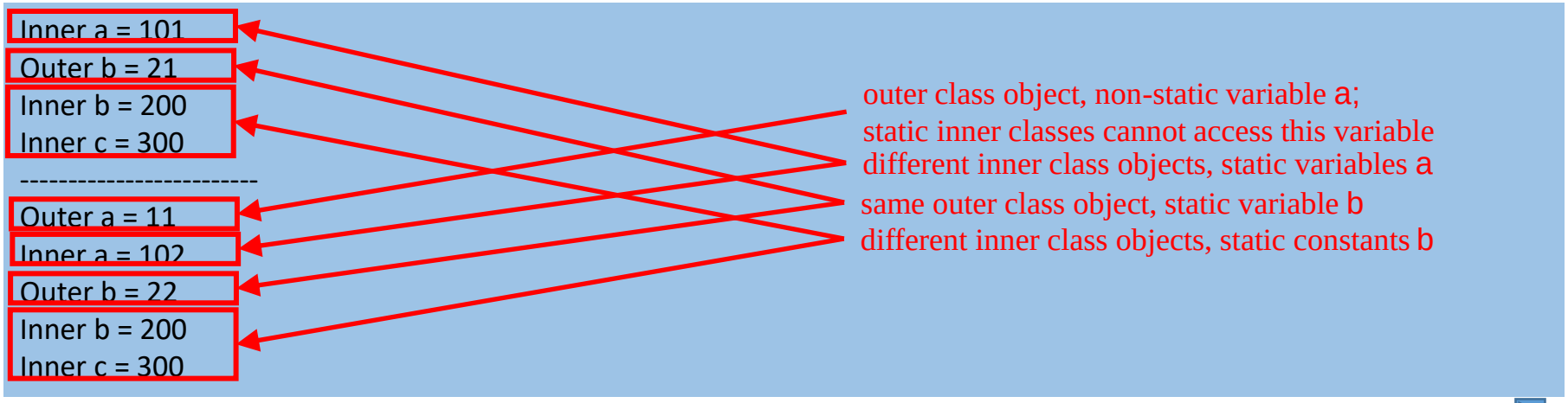
create external class object

create inner class objects without going through outer class objects

call inner class method

call outer class method

InnerClassStaticTest.class



Inner Class in Java

- **Local inner class**: a class that exists **inside a method**.
  - ✓ A local inner class object can be created only in a registered method, and the method of the local inner class can be triggered through the object in the registered method; once execution of the registered method terminates, the life cycle of the local inner class object ends.
  - ✓ If a local inner class references a variable of a registered method, the variable must be a **final** modified constant.
  - ✓ The registered method of the local inner class is referenced through the outer class object.

OuterClassLocal.java

```
package innerClassLocal;

public class OuterClassLocal {
  int a = 10;
  static int b = 20;

  public void foo() {
    int a = 100;
    final int b = 200;

    class InnerClass {
      int a = 1000;

      void show() {
        System.out.println("Outer a = " + ++OuterClassLocal.this.a);
        System.out.println("Inner a = " + ++a);
        System.out.println("Outer b = " + ++OuterClassLocal.b);
        System.out.println("Foo   b = " + b);
      }
    }
```

top-level outer class
non-static variable
static variable

outer class method
non-static variable with the same name as a non-static member of outer class
static constant, with the same name as the outer class static member

local inner class
inner class non-static variable a

inner class method
print outer class non-static variable a
print inner class non-static variable a
print the outer class static constant b
print registerd method non-static constant b

OuterClassLocal.java

```
    InnerClass innerObj = new InnerClass();
    innerObj.show();
     System.out.println("-------------------------");
    System.out.println("Outer a = " + ++OuterClassLocal.this.a);
    System.out.println("Foo   a = " + ++a);
    System.out.println("Outer b = " + ++OuterClassLocal.b);
    System.out.println("Foo   b = " + b);
  }
}
```

create inner class object

call local inner class method

print the outer class non-static variable a

print registered method static variable a

print the outer class static constant b

print registered method non-static constant b

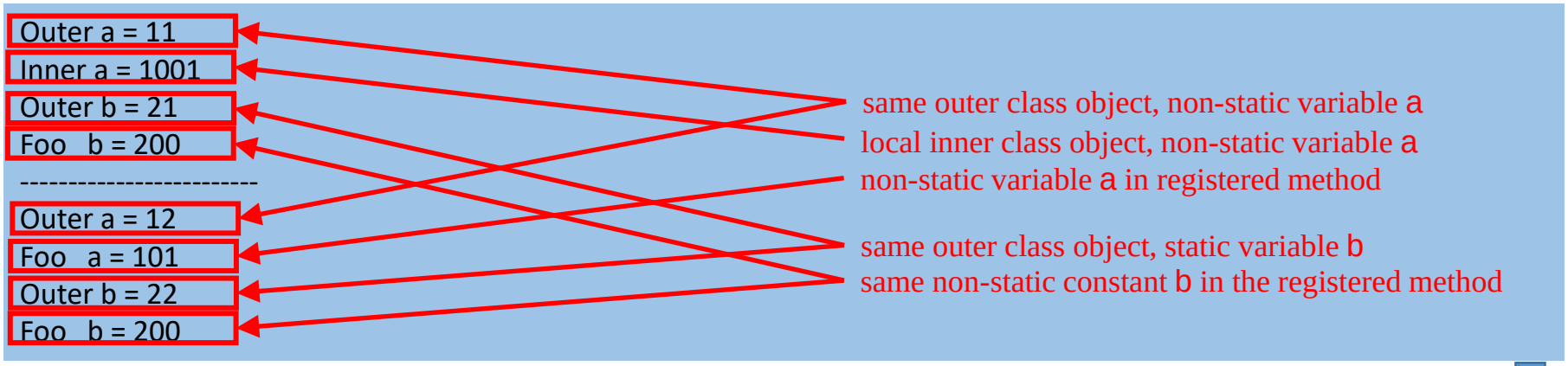
InnerClassStaticTest.java

```
package innerClassLocal;

public class InnerClassLocalTest {

   public static void main(String[] args) {
      OuterClassLocal obj = new OuterClassLocal();
      obj.foo();
   }
}
```

create external class object

call registered method in outer class

InnerClassLocalTest.class



Inner Class in Java

- **Anonymous inner class**: a class exists inside a class, but has ***no class name***.
  - ✓ Class definitions are *merged with* object creation.
  - ✓ A regular class can extend a class and implement any number of interfaces simultaneously. But anonymous Inner class can extend a class or can implement an interface but not both at a time.

Inter.java

```
package innerClassAnonymous;

public interface Inter {
   public abstract void show();
}
```

interface definition, an abstract method show()

ClassNonAnonymous.java

```
package innerClassAnonymous;

public class ClassNonAnonymous implements Inter {
    public void show( ) {
      System.out.println("From object obj1 of a named class.");
    }
}
```

define a named class to implement the interface Inter

call the method show() with the object obj1

InnerClassAnonymous.java

```
package innerClassAnonymous;

public class InnerClassAnonymous {

   public static void main(String[] args) {
      Inter obj2 = new Inter() {
         public void show () {
            System.out.println("From object obj2 of the anonymous class.");
         }
      };     must have semicolon ';'

      ClassNonAnonymous obj1 = new ClassNonAnonymous();
      obj1.show();   call method of the named class

      obj2.show()    call method of an anonymous class
   }
}
```

create an object of anonymous class obj2;
define inner anonymous inner class to implement interface Inter

create an object of named class obj1

ClassNonAnonymous.class

From object obj1 of a named class. named class method output
From object obj2 of the anonymous class. inner anonymous class method output

04 Programming Practice

Practice 1：Hexadecimal Big Integer

BigInteger provides analogues to all of Java‘s primitive integer operators, and all relevant methods from java.lang.Math.

(https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html)

Write a BigHexInteger class to inherit the BigInteger class and handle primitive operations of the hexadecimal number.

For your convenience, you don’t need to write many codes for this class. You can use the super to invoke the methods of BigInteger class.

In this class, the following methods you must implemented:

- Constructor

  public BigHexInteger(String value);

- Overloading method

  public BigHexInteger add(BigHexInteger val);

- Overriding method

  public String toString();

Practice 1：Hexadecimal Big Integer

The following two methods are alternative.

- Overloading method

  public BigHexInteger add(BigInteger val);

- Overriding method

  public BigInteger add(BigInteger val);

You can add any methods in the BigHexInteger class if necessary.

Write a Java to test the add operation of BigIneger and BigHexInteger objects.

This program declares `BigInteger[]` array to refer the objects and calculates the sum of the objects by BigInteregr and BigHexInteger respectively.

The BigInteger[] contains six objects at least. The BigInteger and BigHexInteger objects should be included. The minus number must be taken in consideration also.

Practice 1：Hexadecimal Big Integer

```
bi[0] =121 (BigInteger)
bi[1] =-12 (BigHexInteger)
bi[2] =-23 (BigInteger)
bi[3] =FF (BigHexInteger)
bi[4] =45 (BigInteger)
bi[5] =3A (BigHexInteger)


Sum of BigInteger = 438
Sum of BigHexInteger = 1B6
```