

# Object Oriented Programming

LESSON 05

Delegation and Inheritance  
(P2)



# Outline

---

1. Overriding method and attributes
2. Orders of constructors
3. Default constructor
4. Visibilities
5. Final methods and classes
6. Sealed Classes
7. Record Classes



# Overview

---

In this chapter, you are going to learn about

- Know how to modify method of super class
- Know how to create and use constructors
- Know how to use default constructor
- Know visibilities and its usage in Java
- Know how to define and use final methods and classes

# Learning content

---



1. Overriding method and attributes
  - Creating override methods
  - Overriding method with reuse
  - Specialty of overriding methods
2. Orders of constructors
  - Reusing constructors
  - Constructors chain
  - Example
3. Default constructor
  - Introduction Example
  - Masking default constructor
  - Masking attributes
4. Visibilities
  - Visibilities List in JAVA

- Attributes and Methods visibility
  - Class visibility
5. Final methods and classes
    - Final attributes and methods
    - Final methods
    - Final classes
  6. Sealed classes
    - Sealed Class Constraints
    - Sealed Interfaces
  7. Record classes
    - Record class declaration
    - Record class object
    - Explicit Declaration of Members

# Pre-Test



Question	Possible answers	Correct Answer	Question Feedback
1. Which task that you can let others do for you?	a. Reading book b. Have breakfast c. Taking exam	a. Reading book	Having breakfast must be done yourself. Taking exam must be done yourself
2. Which class can be transformed to another class?	a. Cow b. Butterfly c. Caterpillar	c. Caterpillar	When caterpillar growth into full form, it will transform to butterfly
3. Which one is smallest?	a. Cockroach b. Candle c. Cock	a. Cockroach	Cockroach is smallest.



# 1. Overriding method and attributes

---

- A sub-class can **add variables and methods** to expand on inherited methods and attributes of super-class
  - A sub-class can **override methods that is inherited** and give specific implementation for itself.
- ➔ Overriding a method
- When a class defined a method with **name, return type and arguments identical** to the method of super-class
  - When an overridden method is called on an object of this class, it is **new definition** and not the method of super-class **that is executed**.

# 1.1. Creating override methods

---

```
public class A {  
    public void hello() {  
        System.out.println("Hello");  
    }  
  
    public void display() {  
        System.out.println(" I am A");  
    }  
}
```

```
public class B extends A {  
    public void display() {  
        System.out.println(" I am B");  
    }  
}
```

```
A a = new A();
```

```
B b = new B();
```

```
a.hello();    --> Hello
```

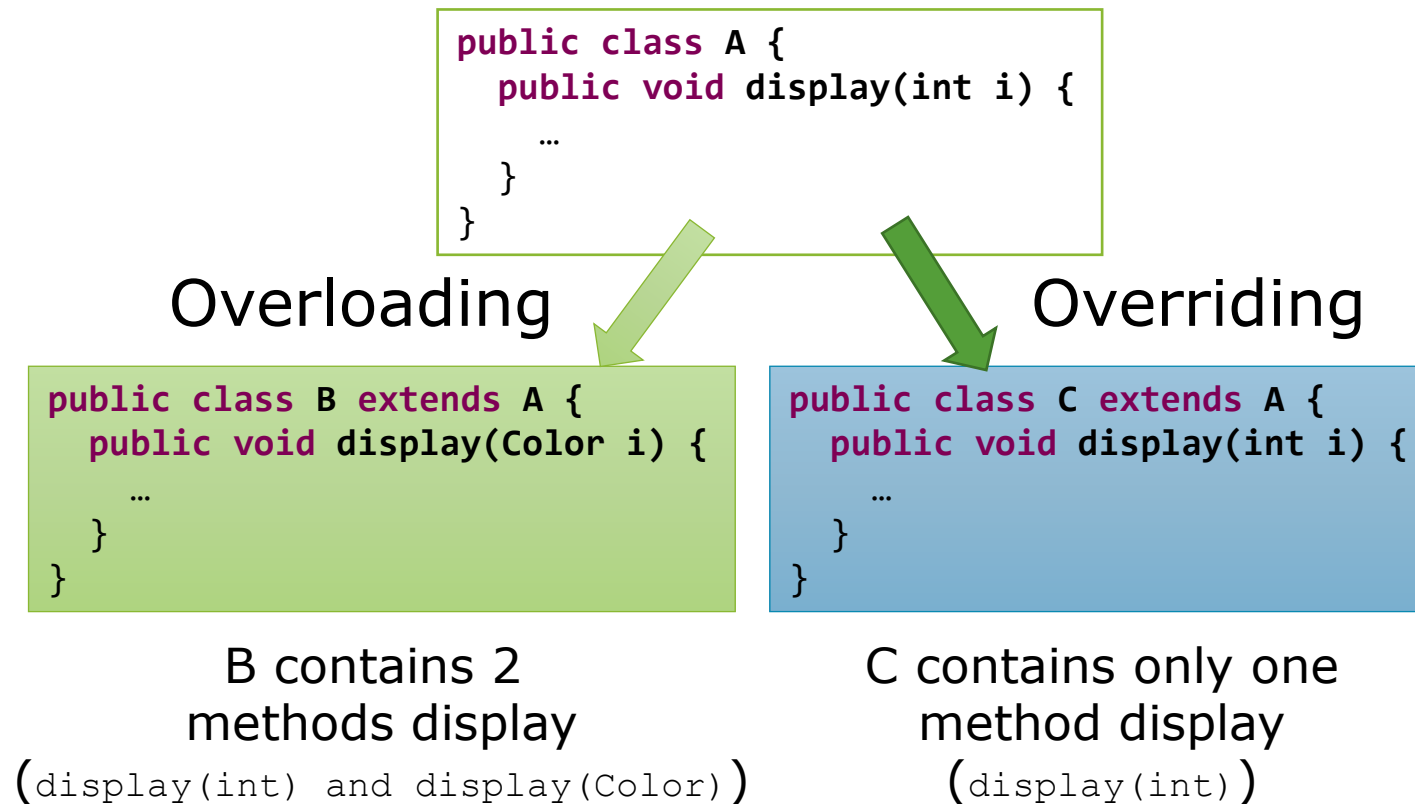
```
a.display();  --> I am A
```

```
b.hello();    --> Hello
```

```
b.display();  --> I am B
```

# 1.1. Creating override methods

- DO NOT confused **overriding** with **overloading**





## 1.2. Overriding method with reuse

- Overriding methods
  - Possibility to reuse code from inherited method (`super`)

```
public class Student {
    String surname;
    String givenname;
    int age;

    // ...
    public void display() {
        System.out.println("Surname : " + surname + " Given Name : " + givenname);
        System.out.println("Age : " + age);
        // ...
    }
    // ..
}

public class SportStudent extends Student {
    String practicalSport;

    // ...
    public void display() {
        super.display();

        System.out.println("Practical Sport : " + practicalSport);
        // F...
    }
}
```

## 1.3. Specialty of overriding methods

- Simple Inheritance
  - A class can not inherit more than one class
    - In some languages such as C++, can inherit more than one
- Inheritance hierarchy is a tree that its root is Object class (in package java.lang)
  - All classes has a super class except Object class
  - All classes that inherit direct or indirect from class Object
  - By default a class without declaring extends clause, inherits from class Object

```
public class Point extends Object {  
    int x;  
    int y;  
    // ...  
}
```



```
public class Point {  
    int x;  
    int y;  
    // ...  
}
```



## 1.3. Specialty of overriding methods

---

- Principal methods of class Object:
  - `protected Object clone()`  
Creates and returns a copy of this object.
  - `boolean equals(Object obj)`  
Indicates whether some other object is "equal to" this one.
  - `Class<?> getClass()`  
Returns the runtime class of this Object.
  - `int hashCode()`  
Returns a hash code value for the object.
  - `String toString()`  
Returns a string representation of the object.  
**`return getClass().getName() + "@" + Integer.toHexString(hashCode());`**
  - ...



## 1.3. Specialty of overriding methods

- About method `toString`

Concatenation  
operator

`<Expression type String> + <reference>`

$\Leftrightarrow$

`<Expression type String> + <reference>.toString()`

- Because there is method `toString` in class `Object`, we are sure that any type (class) of object will response to message `toString()` (because `Object` is root of all classes)



## 1.3. Specialty of overriding methods

```
public String toString(){  
    return getClass().getName() + "@"  
        + Integer.toHexString(hashCode());  
}
```

```
public class Object {  
    ...  
}
```

```
public class Point {  
    private double x;  
    private double y;  
    ...  
}
```

```
public String toString(){  
    return "Point:[" + x  
        + "," + y + "];"  
}
```

```
Point p = new Point(15, 11);  
System.out.println(p);
```

In case, in class Point  
does not override toString()  
of super class Object:

**Point@2a139a55**

In case, in class Point  
does override toString()  
of super class Object:

**Point:[15,11]**



## 2.1. Reusing Constructors

---

- Method overriding:
  - Possibility of reusing code of inherited method (`super`)
- In the same way, it is important to be able to **reuse code of constructors of super class** in definition of constructor of the **new class**
  - Calling a constructor of super class:  
`super(parameters of constructor)`
  - Using *`super(...)`* is similar to *`this(...)`*

## 2.1. Reusing Constructors

```
public class Point {  
    double x,y;  
  
    public Point(double x, double y){  
        this.x = x;  
        this.y = y;  
    }  
    ...  
}
```

```
public class PointGraphic extends Point {  
    Color c;  
  
    public PointGraphic(double x, double y, Color c) {  
        super(x, y);  
        this.c = c;  
    }  
    ...  
}
```

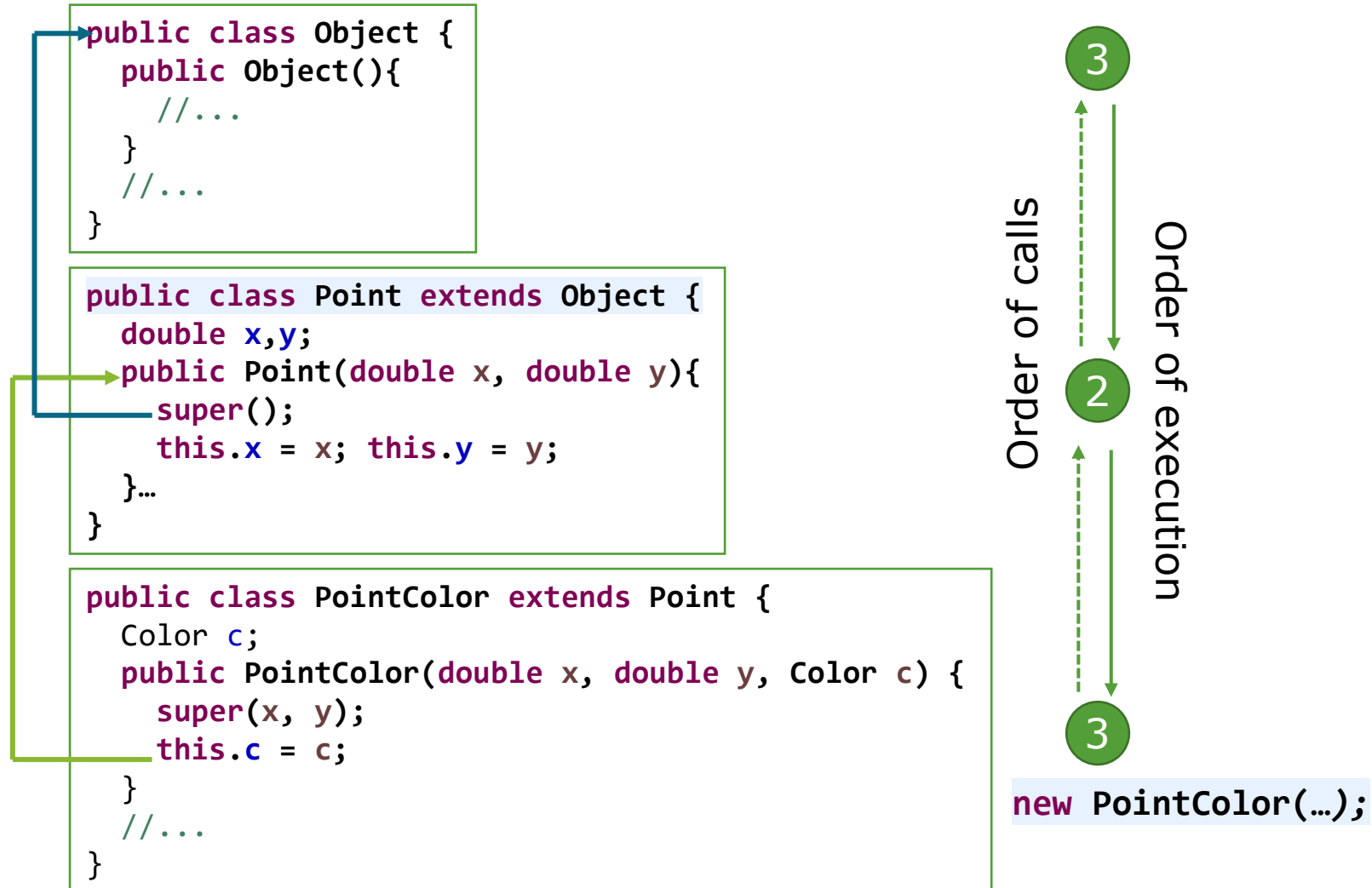
## 2.2. Constructors Chain

---

- Calling to a **constructor of super class** must always be the **first instruction** in body of constructor
  - If first instruction in constructor is **not an explicit call** to one of constructors of super class, **JAVA inserts implicitly the call** to `super()`
  - Each time an object is created, the constructors are **executed upward in sequence** of class by class in **hierarchy** up until Object class
  - Body of **constructor of Object** class is **always** been **executed** the first, then followed by body of constructors of different classes in descendent in hierarchy.
- Ensure that **constructor** of a class is **always been called** each time its **sub-class is created**
  - An object `c` instance of C sub-class of B, that is also sub-class of A, is an object of class C but also an object of class B and object of A  
When `c` is created, it is with characteristics of A, B and C



## 2.3. Example





### 3. Default Constructor

---

- When a class does not provide any explicit declaration of constructor, there is a default constructor:
  - Without parameter
  - Its body is empty
  - Not exists if there is another constructor defined

## 3.1. Introduction Example

```
public class Object {  
    public Object(){  
        //...  
    }  
    //...  
}
```

```
public class A extends Object {  
    //Attributes  
    String name;  
  
    //Methods  
    String getName(){  
        return name;  
    }  
}
```

```
public A(){  
    super();  
}
```

Implicit default  
constructor

Guaranty that  
constructor chain works



## 3.2. Masking default constructor

```
public class ClassA {  
    double x;  
  
    // constructor  
    public ClassA(double x) {  
        this.x = x;  
    }  
}
```

Explicit constructor  
masked default constructor

No constructor  
without parameter

```
public class ClassB extends ClassA {  
    double y = 0;  
    // no constructor  
}
```

```
public ClassB(){  
    super();  
}
```

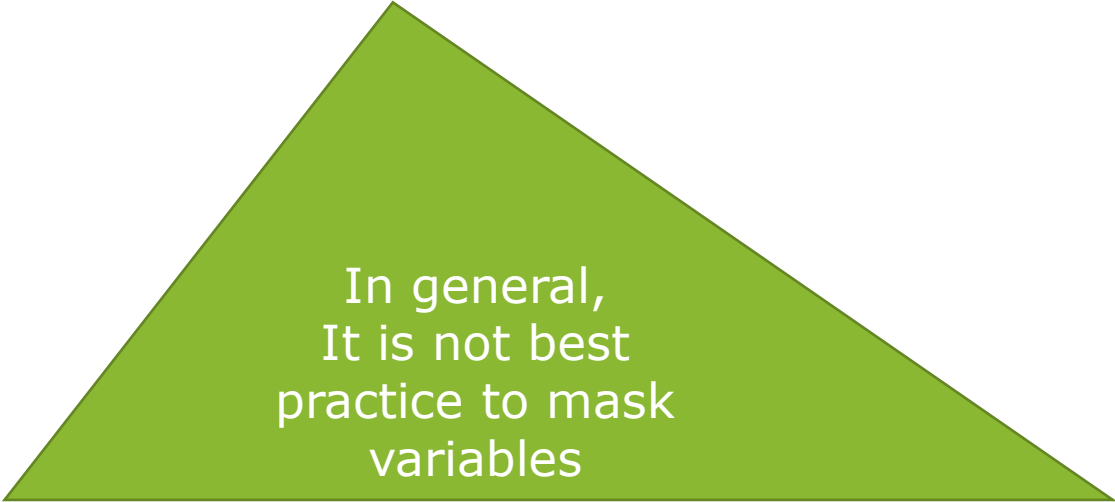
Implicit  
default  
constructor

```
C:>javac ClasseB.java  
ClasseB.java:3: No constructor matching ClasseA() found in class ClasseA.  
    public ClasseB() {  
        ^  
1 error  
  
Compilation exited abnormally with code 1 at Sat Jan 29 09:34:24
```

## 3.3. Masking Attributes

---

- Masking Attributes (Shadowed variables)
  - When a **sub-class** defined an **instance variable** (non-static variable) with the same name as variable in super class, the new definition masked the inherited definition
    - To access to inherited variable, can be done through **super**



In general,  
It is not best  
practice to mask  
variables

## 3.3. Masking Attributes

```
public class ClassA {  
    int x;  
}
```

```
public class ClassB extends ClassA {  
    double x;  
}
```

```
public class ClassC extends ClassB {  
    char x;  
    public void doSomething(){  
        ((ClassA)this).x = 5;  
        super.super.x = 5; ✗  
        super.x = 3.0;  
        x = 'A';  
        this.x = 'A';  
    }...  
}
```



## 4. Visibilities

---

- Principal of Encapsulation: **all data** belong to an object **can not be accessible** except from methods of this object
  - **Data security**: it is confident when we are sure that there is only one way to modify our data is through methods
  - **Mask implementation**: Implementation is masked inside the class, so we can make change inside with no one know



## 4.1. Visibilities List in JAVA

---

- In JAVA, it is possible to control visibility on properties (variables and methods) of a class
  - `public` accessible from all other class
  - `private` is accessible only in the class where it defined
  - `protected` is accessible in class where it defined, in all classes that is sub-class of this class and classes in the same package
  - `package` (default visibility) can be accessible only in classes in the same package as the class it defined





## 4.1. Visibilities List in JAVA

	<b>private</b>	<b>- (package)</b>	<b>protected</b>	<b>public</b>
In Class itself	YES	YES	YES	YES
In Classes in the same package	NO	YES	YES	YES
In sub-classes in other package	NO	NO	YES	YES
In classes (not sub-classes) in other package	NO	NO	NO	YES

## 4.2. Attributes and Methods visibility

Package mypackage

```
package mypackage;  
public class Class2 {  
    Class1 o1;  
}
```

```
package mypackage;  
public class Class3  
    extends Class1 {  
}
```

Declare the  
package where  
this class  
belongs to

```
package mypackage;  
public class Class1 {  
    private int a;  
    int b;  
    protected int c;  
    public int d;  
}
```

Package otherpackage

```
package otherpackage;  
import mypackage.Class1;  
public class Class4 {  
    Class1 o1;  
}
```

```
package otherpackage;  
import mypackage.Class1;  
public class Class5  
    extends Class1 {  
}
```

## 4.3. Class visibility

- 2 levels of visibilities for classes:
  - public: the class is usable by any other class
  - - (package): class is usable only by any classes in the same package where it belongs to

### Package A

```
package A;  
public class ClassA {  
    ClassB b;  
}
```

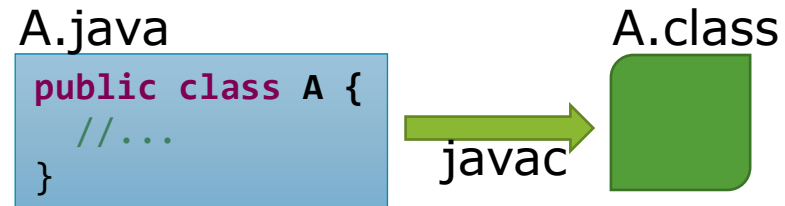
```
package A;  
class ClassB extends ClassA {  
  
}
```

### Package B

```
package B;  
import visibleclass.A.ClassA;  
public class ClassC {  
  
    ClassA a;  
    ClassB b;  
}
```

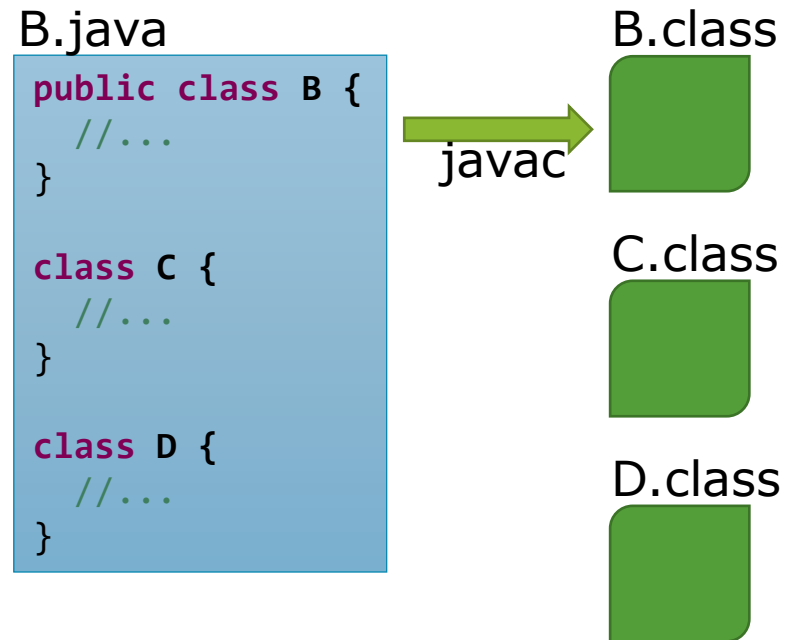
## 4.3. Class visibility

- Up until now, we always say:
  - One class, One file, or A class per file
  - Name of source file: name of class with extension .java



## 4.3. Class visibility

- In fact, the real rule is:
  - One public class per file
  - Name of source file: name of public class





## 5.1. Final Attributes

---

- A final variable can only be initialized once, either via an **initializer** or an **assignment statement**.
- It does not need to be initialized at the point of declaration
  - this is called a "blank final" variable
  - must be definitely assigned in every constructor of the class in which it is declared
  - must be definitely assigned in a static initializer of the class in which it is declared



## 5.1. Final Attributes

```
public class Sphere {  
    // pi is a universal constant, about as constant as anything can be.  
    public static final double PI = 3.141592653589793;  
  
    public final double radius;  
    public final double xPos;  
    public final double yPos;  
    public final double zPos;  
  
    Sphere(double x, double y, double z, double r) {  
        radius = r;  
        xPos = x;  
        yPos = y;  
        zPos = z;  
    }  
    [...]  
}
```



## 5.2. Final Methods

---

```
public final void methodX(...) {  
    ...  
}
```

- “Lock” this method for preventing for all overriding in sub-classes
- Efficiency:
  - When compiler meets a call to a final method, it can replace the call normal of this method (stack the arguments on the stack, jump to the code of the method, return to the calling code, unstacking arguments, obtaining the return value)





## 5.3. Final Classes

---

- A class can be defined as final

```
public final class Aclass {  
    ...  
}
```

- This will deny all inheritances for this class, can not have sub-class
  - All methods in this final class will become implicitly final methods (can not be overridden)
  - Example: Class String is final class
- **Pay attention on usage of final:** it will lost efficiency of reusability through inheritance.



## 6. Sealed Classes

---

- Since Java SE 15, A keyword sealed and permits are used to restrict which other classes or interfaces from extending or implementing a class or interface.

```
public sealed class Shape
    permits Circle, Square, Rectangle {

}
```

- Class Shape is sealed (not allow any inheritance) but excepts (permits) only child classes Circle, Square and Rectangle



## 6.1. Sealed Class Constraints

Permitted subclasses have the following constraints:

- They must be accessible by the sealed class at compile time.  
For example, to compile `Shape.java`, the compiler must be able to access all of the permitted classes of `Shape`: `Circle.java`, `Square.java`, and `Rectangle.java`. In addition, because `Rectangle` is a sealed class, the compiler also needs access to `FilledRectangle.java`.
  - They must directly extend the sealed class.
  - They must have exactly one of the following modifiers to describe how it continues the sealing initiated by its superclass:
    - `final`: Cannot be extended further
    - `sealed`: Can only be extended by its permitted subclasses
    - `non-sealed`: Can be extended by unknown subclasses; a sealed class cannot prevent its permitted subclasses from doing this
- For example, the permitted subclasses of `Shape` demonstrate each of these three modifiers: `Circle` is `final` while `Rectangle` is `sealed` and `Square` is `non-sealed`.
- They must be in the same module as the sealed class (if the sealed class is in a named module) or in the same package (if the sealed class is in the unnamed module, as in the `Shape.java` example).



## 6.2. Sealed Interfaces

Like sealed classes, to seal an interface, add the sealed modifier to its declaration. Then, after any extends clause, add the permits clause, which specifies the classes that can implement the sealed interface and the interfaces that can extend the sealed interface.

```
sealed interface Expr permits ConstantExpr, PlusExpr {  
    public int eval();  
}
```

```
final class ConstantExpr implements Expr {  
    int i;  
    ConstantExpr(int i) {  
        this.i = i;  
    }  
    public int eval() { return i; }
```

```
final class PlusExpr implements Expr {  
    Expr a, b;  
    PlusExpr(Expr a, Expr b) {  
        this.a = a; this.b = b;  
    }  
    public int eval() {  
        return a.eval() + b.eval();  
    }  
}
```



## 7. Record classes

---

- Introduced as a preview feature in Java SE 14, record classes help to model plain data aggregates with less ceremony than normal classes. Java SE 17 is a permanent feature. This means that we can use this feature from Java SE 17 onward.
- A record class declares a sequence of fields, and then the appropriate accessors, constructors, equals, hashCode, and toString methods are created automatically. The fields are final because the class is intended to serve as a simple "data carrier".



## 7.1. Record class declaration

---

- Syntax

```
record RecordName(<fields>) { }
```

- Example:

```
record Rectangle(double length, double width) { }
```



## 7.1. Record class declaration

---

A record class declaration consists of a name, a header (which lists the fields of the class, known as its "*components*"), and a body.

A record class declares the following members automatically:

- For each component in the header, the following two members:
  - A private final field.
  - A public accessor method with the same name and type of the component; in the Rectangle record class example, these methods are Rectangle::length() and Rectangle::width().
- A canonical constructor whose signature is the same as the header. This constructor assigns each argument from the new expression that instantiates the record class to the corresponding private field.
- Implementations of the equals and hashCode methods, which specify that two record classes are equal if they are of the same type and contain equal component values.
- An implementation of the toString method that includes the string representation of all the record class's components, with their names.



## 7.1. Record class declaration

- Example: `record Rectangle(double length, double width) { }`

It is equivalent to normal class:

```
public final class Rectangle {  
    private final double length;  
    private final double width;  
  
    public Rectangle(double length,  
                     double width) {  
        this.length = length;  
        this.width = width;  
    }  
  
    double length() { return this.length; }  
    double width() { return this.width; }
```

```
// Implementation of equals() and hashCode(),  
// which specify that two record objects are equal if they  
// are of the same type and contain equal field values.  
public boolean equals...  
public int hashCode...  
  
// An implementation of toString() that returns a string  
// representation of all the record class's fields,  
// including their names.  
public String toString() {...}  
}
```





## 7.2. Record class object

---

- Just like normal class, its object instance is created using **new** keyword. For example:

```
Rectangle r = new Rectangle(4,5);
```



## 7.3. Explicit Declaration of Members

---

- You can explicitly declare any of the members derived from the header, such as the public accessor methods that correspond to the record class's components, for example:

```
record Rectangle(double length, double width) {  
  
    // Public accessor method  
    public double length() {  
        System.out.println("Length is " + length);  
        return length;  
    }  
}
```



## 7.3. Explicit Declaration of Members

- You can declare static fields, static initializers, and static methods in a record class, and they behave as they would in a normal class, for example:

```
record Rectangle(double length, double width) {  
  
    // Static field  
    static double goldenRatio;  
  
    // Static initializer  
    static {  
        goldenRatio = (1 + Math.sqrt(5)) / 2;  
    }  
  
    // Static method  
    public static Rectangle createGoldenRectangle(double width) {  
        return new Rectangle(width, width * goldenRatio);  
    }  
}
```

# Test



Question	Possible Answers	Correct Answer
1) Fill in the gaps:	Calling to a constructor of super class must always be the ..... in body of constructor	first instruction
2) Defined a method with the same name as super class, we call:	a) Inheritance b) Overriding c) Overloading d) Same method e) Delegation	c) Overloading
3) How do we instantiate a class ClassA without declaring a constructor?:	a) Not possible b) ClassA.getInstance() c) new ClassA() d) Class.forName("ClassA") e) Instantiate(ClassA)	c) new ClassA()

# Test



Question	Possible Answers	Correct Answer
4) If we want method can be accessible from our sub-classes and classes in the same package, what are possible visibility modifiers?	a) public b) protected c) - (package) d) private e) Not possible	b) protected
5) What are modifiers that allow to place in declaration of class?	a) public b) protected c) - (package) d) private	a) public c) - (package)

# Practice



No.	Exercise	Solution
1.	Implement class Point and PointGraphic in course.	
2,	Create class Butterfly	
3,	Implement class Butterfly Dictionary	



# Summarize

---

- Overriding method is defining method identical to a method of super-class
- Object constructor is always get executed the first, and we can reuse constructors
- Default constructor is implicit constructor that exist only when there is no other constructors
- There are 4 types of visibility in JAVA including public, private, package, and protected
- Final methods and attributes is used to prevent further overriding in sub-classes



# Reference

---

- “Object Oriented Programming” by Ph. Genoud – Université Joseph Fourier, 2006
- “Building Skills in Object-Oriented Design” by Steven F. Lott, 2009
- “Exercices en Java” 2nd edition by Groupe Eyrolles, 2006
- “Java Programming – Introductory” by Joyce Farrell, 1999
- “Java Examples in a Nutshell” 3rd Edition by David Flanagan, 2004
- “Programmer en Java” 5th edition by Groupe Eyrolles, 2008