CE204 Object-Oriented Programming

Week-2 (OOP with Java-II)

Spring Semester, 2021-2022

Download DOC, SLIDE, PPTX

OOP with Java-II

Outline (1)

- Java super Keyword
- Java final Keyword
- Java Polymorphism / Encapsulation
- Java Method Overriding
- Java Nested Inner Class
- Java Static Class
- Java Anonymous Class

Outline (2)

- Java Enums / Enum-Constructor / Enum-String
- Java Abstract Class
- Java Object Class
- Java Forms of Inheritance
- Java Benefits and Costs of Inheritance
- Java Packages
- Java Access Protection in Packages

Java super keyword

Java super keyword

- In java, `super` is a keyword used to refers to the **parent class object**.
- The `super` keyword came into existence to solve the *naming conflicts* in the inheritance.
- When both parent class and child class have members with the same name,
  - then the super keyword is used to refer to the parent class version.

Java super keyword

- In another word, The super keyword in Java is used in subclasses to access superclass members (attributes, constructors and methods).

Java super keyword

- In java, the super keyword is used for the following purposes.
  - To refer parent class **data members**
  - To refer parent class **methods**
  - To call parent class **constructor**

Java super keyword

- To call methods of the superclass that is overridden in the subclass.
- To access attributes (fields) of the superclass if both superclass and subclass have attributes with the same name.
- To explicitly call superclass no-arg (default) or parameterized constructor from the subclass constructor.

Java super keyword

- The super keyword is used inside the child class only.

super to refer parent class data members

- When both parent class and child class have data members with the same name,
  - then the super keyword is used to refer to the parent class data member from child class.

super to refer parent class data members

```
class ParentClass{

        int num = 10;

}
```

```
class ChildClass extends ParentClass{

        int num = 20;

        void showData() {
                System.out.println("Inside the ChildClass");
                System.out.println("ChildClass num = " + num);
                System.out.println("ParentClass num = " + super.num);
        }
}
```

super to refer parent class data members

```
public class SuperKeywordExample {

        public static void main(String[] args) {
                ChildClass obj = new ChildClass();

                obj.showData();

                System.out.println("\nInside the non-child class");
                System.out.println("ChildClass num = " + obj.num);
                //System.out.println("ParentClass num = " + super.num); //super can't be used here
        }
}
```

super to refer parent class method

- When both parent class and child class have method with the same name,
  - then the super keyword is used to refer to the parent class method from child class.

super to refer parent class method

```
class ParentClass{

        int num1 = 10;

        void showData() {
                System.out.println("\nInside the ParentClass showData method");
                System.out.println("ChildClass num = " + num1);
        }
}
```

super to refer parent class method

```
class ChildClass extends ParentClass{

        int num2 = 20;

        void showData() {
                System.out.println("\nInside the ChildClass showData method");
                System.out.println("ChildClass num = " + num2);

                super.showData();

        }
}
```

super to refer parent class method

```
public class SuperKeywordExample {

        public static void main(String[] args) {
                ChildClass obj = new ChildClass();

                obj.showData();
                //super.showData();     // super can't be used here

        }
}
```

super to call parent class constructor

- When an object of child class is created, it automatically calls the parent class default-constructor before it's own.
- But, the parameterized constructor of parent class must be called explicitly using the super keyword inside the child class constructor.

super to call parent class constructor

```
class ParentClass{

        int num1;

        ParentClass(){
                System.out.println("\nInside the ParentClass default constructor");
                num1 = 10;
        }

        ParentClass(int value){
                System.out.println("\nInside the ParentClass parameterized constructor");
                num1 = value;
        }
}
```

super to call parent class constructor

```
class ChildClass extends ParentClass{

        int num2;

        ChildClass(){
                super(100);
                System.out.println("\nInside the ChildClass constructor");
                num2 = 200;
        }
}
```

super to call parent class constructor

```
public class SuperKeywordExample {

        public static void main(String[] args) {

                ChildClass obj = new ChildClass();

        }
}
```

super to call parent class constructor

- To call the parameterized constructor of the parent class,
- the super keyword must be the first statement inside the child class constructor,
- and we must pass the parameter values.

Access Overridden Methods of the superclass

- If methods with the same name are defined in both superclass and subclass, the method in the subclass overrides the method in the superclass. This is called method overriding.

Example 1: Method overriding

```
class Animal {

  // overridden method
  public void display(){
    System.out.println("I am an animal");
  }
}
```

Example 1: Method overriding

```
class Dog extends Animal {

  // overriding method
  @Override
  public void display(){
    System.out.println("I am a dog");
  }

  public void printMessage(){
    display();
  }
}
```

Example 1: Method overriding

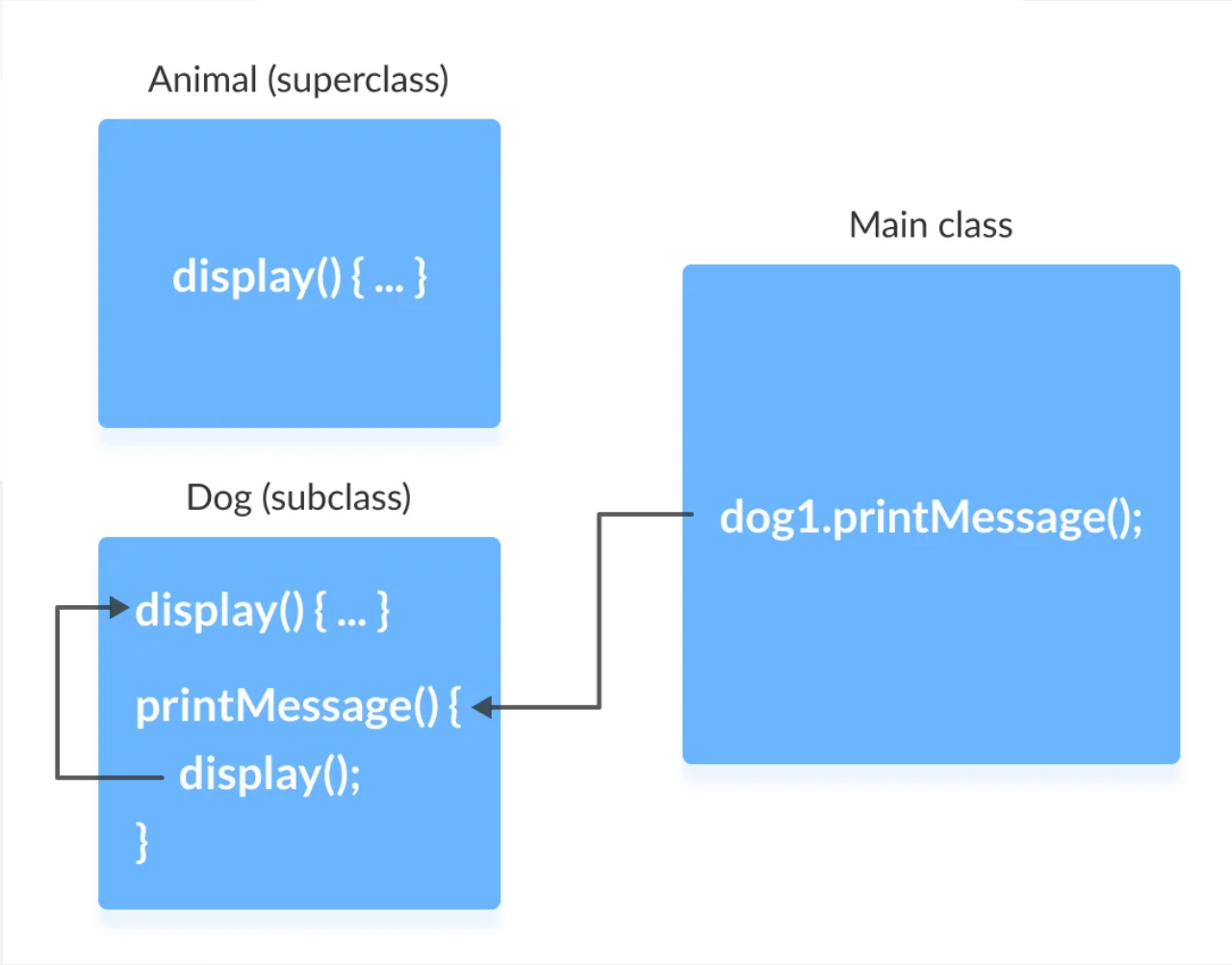
```
class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}
```

Example 1: Method overriding

In this example, by making an object dog1 of Dog class, we can call its method printMessage() which then executes the display() statement.

Since display() is defined in both the classes, the method of subclass Dog overrides the method of superclass Animal. Hence, the display() of the subclass is called.

Example 1: Method overriding



What if the overridden method of the superclass has to be called?

- We use super.display() if the overridden method display() of superclass Animal needs to be called.

Example 2: super to Call Superclass Method

```
class Animal {

  // overridden method
  public void display(){
    System.out.println("I am an animal");
  }
}
```

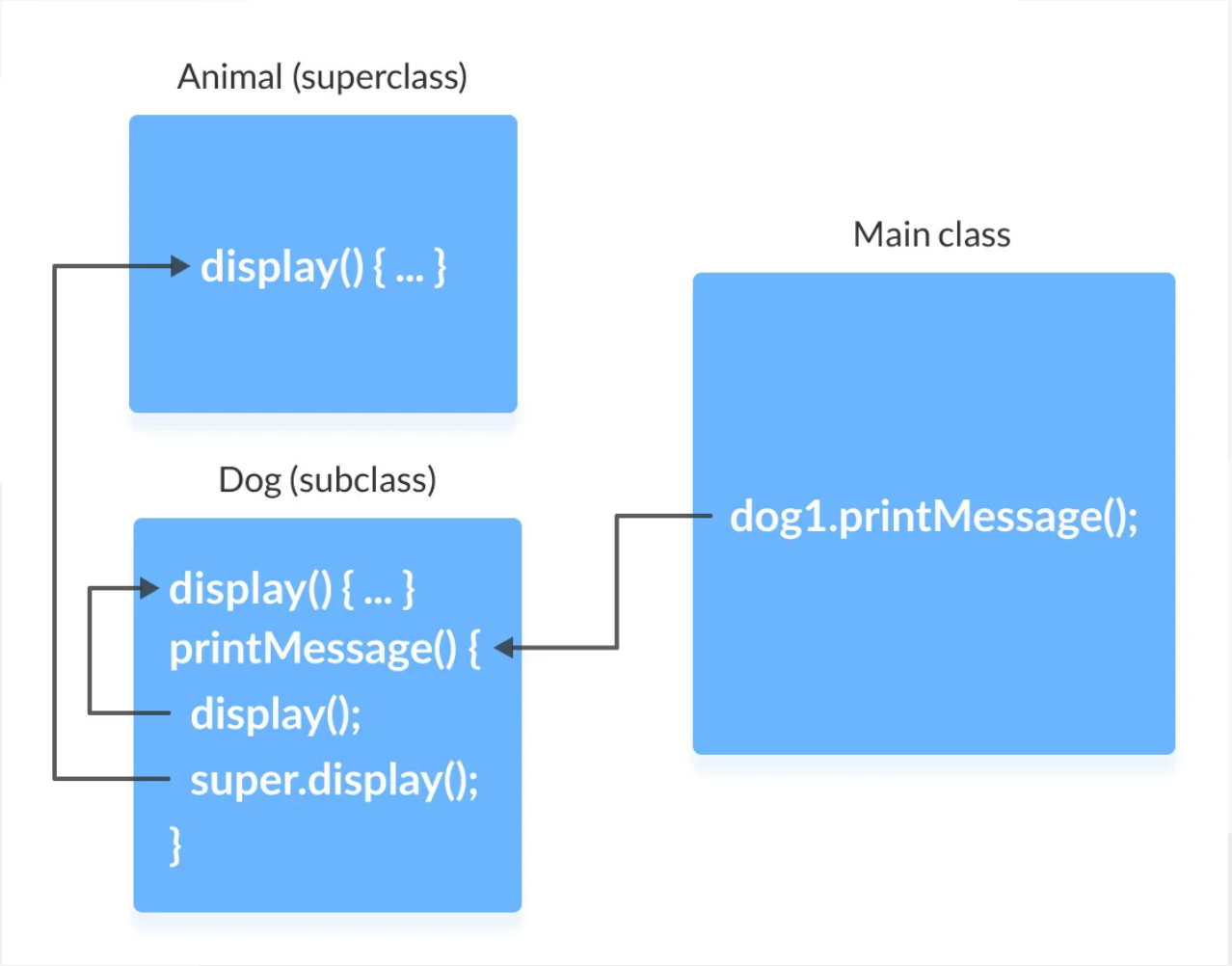
Example 2: super to Call Superclass Method

```
class Dog extends Animal {

  // overriding method
  @Override
  public void display(){
    System.out.println("I am a dog");
  }

  public void printMessage(){

    // this calls overriding method
    display();

    // this calls overridden method
    super.display();
  }
}
```

Example 2: super to Call Superclass Method

```
class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}
```

Example 2: super to Call Superclass Method



Access Attributes of the Superclass

- The superclass and subclass can have attributes with the same name.
  - We use the super keyword to access the attribute of the superclass.

Example 3: Access superclass attribute

```
class Animal {
  protected String type="animal";
}
```

```
class Dog extends Animal {
  public String type="mammal";

  public void printType() {
    System.out.println("I am a " + type);
    System.out.println("I am an " + super.type);
  }
}
```

Example 3: Access superclass attribute

```
class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printType();
  }
}
```

Example 3: Access superclass attribute

- In this example, we have defined the same instance field `type` in both the superclass `Animal` and the subclass `Dog`.
- We then created an object `dog1` of the Dog class. Then, the `printType()` method is called using this object.
  - Inside the `printType()` function,
    - `type` refers to the attribute of the subclass `Dog`.
    - `super.type` refers to the attribute of the superclass Animal.

Use of super() to access superclass constructor

- As we know, when an object of a class is created, its default constructor is automatically called.
- To explicitly call the superclass constructor from the subclass constructor, we use `super()`. It's a special form of the super keyword.
- `super()` can be used only inside the subclass constructor and must be the first statement.

Example 4: Use of super()

```
class Animal {

  // default or no-arg constructor of class Animal
  Animal() {
    System.out.println("I am an animal");
  }
}
```

Example 4: Use of super()

```
class Dog extends Animal {

  // default or no-arg constructor of class Dog
  Dog() {

    // calling default constructor of the superclass
    super();

    System.out.println("I am a dog");
  }
}
```

Example 4: Use of super()

```
class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
  }
}
```

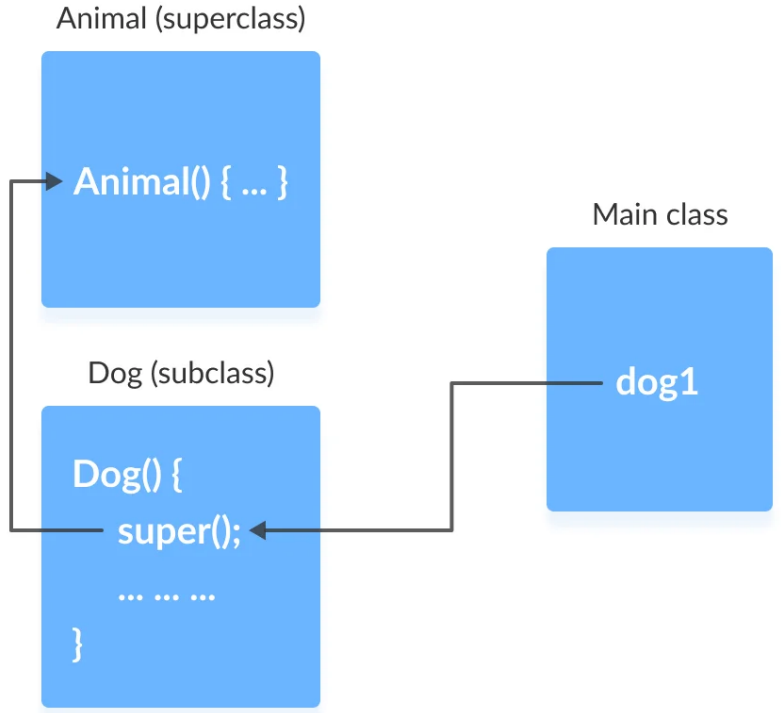
Example 4: Use of super()

- when an object dog1 of Dog class is created, it automatically calls the default or no-arg constructor of that class.
- Inside the subclass constructor, the super() statement calls the constructor of the superclass and executes the statements inside it. Hence, we get the output I am an animal.

Example 4: Use of super()

The flow of the program then returns back to the subclass constructor and executes the remaining statements. Thus, I am a dog will be printed.

However, using super() is not compulsory. Even if super() is not used in the subclass constructor, the compiler implicitly calls the default constructor of the superclass.



Example 4: Use of super()

- **So, why use redundant code if the compiler automatically invokes super()?**
  - It is required if the parameterized constructor (a constructor that takes arguments) of the superclass has to be called from the subclass constructor.
- The parameterized super() must always be the first statement
  - in the body of the constructor of the subclass,
  - otherwise, we get a compilation error.

Example 5: Call Parameterized Constructor Using super()

```
class Animal {

  // default or no-arg constructor
  Animal() {
    System.out.println("I am an animal");
  }

  // parameterized constructor
  Animal(String type) {
    System.out.println("Type: "+type);
  }
}
```

Example 5: Call Parameterized Constructor Using super()

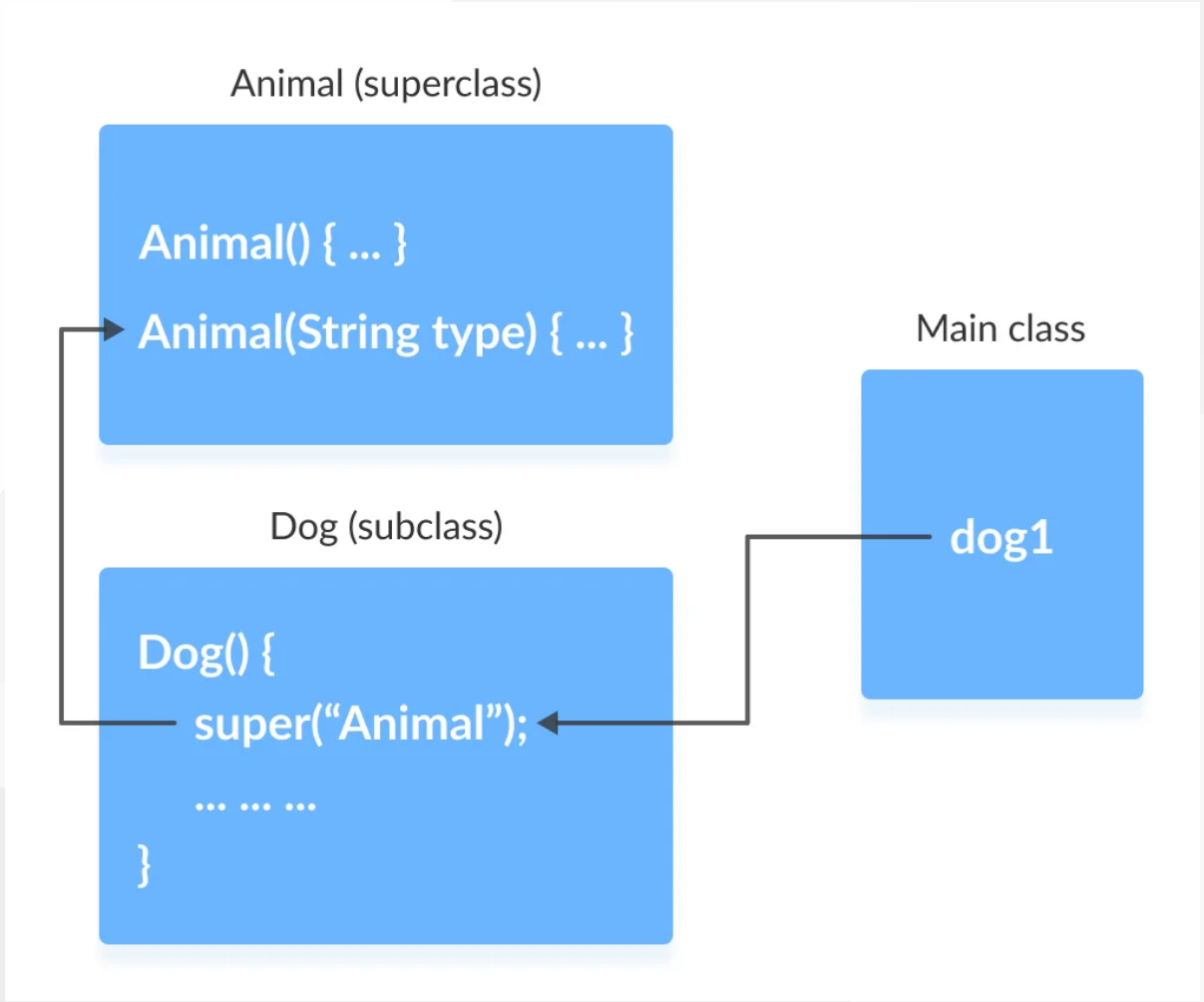
```
class Dog extends Animal {

  // default constructor
  Dog() {

    // calling parameterized constructor of the superclass
    super("Animal");

    System.out.println("I am a dog");
  }
}
```

Example 5: Call Parameterized Constructor Using super()

```
class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
  }
}
```

Example 5: Call Parameterized Constructor Using super()

If a parameterized constructor has to be called, we need to explicitly define it in the subclass constructor.



Example 5: Call Parameterized Constructor Using super()

Note that in the above example, we explicitly called the parameterized constructor super("Animal"). The compiler does not call the default constructor of the superclass in this case.

Java final keyword

Java final keyword

- In java, the final is a keyword and it is used with the following things.
  - With variable (to create constant)
  - With method (to avoid method overriding)
  - With class (to avoid inheritance)

Java final restrictions

- the final variable cannot be reinitialized with another value
- the final method cannot be overridden
- the final class cannot be extended

final with variables

- When a variable defined with the final keyword,
- it becomes a constant, and
  - it does not allow us to modify the value.
- The variable defined with the final keyword allows only a one-time assignment,
  - once a value assigned to it,
    - never allows us to change it again.

final with variables example-1

```
public class FinalVariableExample {
        public static void main(String[] args) {
                final int a = 10;
                System.out.println("a = " + a);
                a = 100;        // Can't be modified
        }
}
```

final with variables example-2

```
class Main {
  public static void main(String[] args) {

    // create a final variable
    final int AGE = 32;

    // try to change the final variable
    AGE = 45;
    System.out.println("Age: " + AGE);
  }
}
```

final with variables recommendation

- It is recommended to use uppercase to declare final variables in Java.

final with methods

- When a method defined with the final keyword,
  - it does not allow it to override.
- The final method extends to the child class,
  - but the child class can not override or re-define it.
- It must be used as it has implemented in the parent class.

final with methods example-1

```
class ParentClass{

        int num = 10;

        final void showData() {
                System.out.println("Inside ParentClass showData() method");
                System.out.println("num = " + num);
        }

}
```

final with methods example-1

```
class ChildClass extends ParentClass{

        void showData() {
                System.out.println("Inside ChildClass showData() method");
                System.out.println("num = " + num);
        }
}
```

final with methods example-1

```
public class FinalKeywordExample {

        public static void main(String[] args) {

                ChildClass obj = new ChildClass();
                obj.showData();

        }
}
```

final with methods example-2

```
class FinalDemo {
    // create a final method
    public final void display() {
      System.out.println("This is a final method.");
    }
}

class Main extends FinalDemo {
  // try to override final method
  public final void display() {
    System.out.println("The final method is overridden.");
  }

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

final with class

- When a class defined with final keyword, it can not be extended by any other class.

final with class example-1

```
final class ParentClass{

        int num = 10;

        void showData() {
                System.out.println("Inside ParentClass showData() method");
                System.out.println("num = " + num);
        }

}
```

final with class example-1

```
class ChildClass extends ParentClass{


}
```

final with class example-1

```
public class FinalKeywordExample {

        public static void main(String[] args) {

                ChildClass obj = new ChildClass();

        }
}
```

final with class example-2

```
// create a final class
final class FinalClass {
  public void display() {
    System.out.println("This is a final method.");
  }
}

// try to extend the final class
class Main extends FinalClass {
  public  void display() {
    System.out.println("The final method is overridden.");
  }

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

Java Polymorphism

Java Polymorphism

- The polymorphism is the process of defining same method with different implementation. That means creating multiple methods with different behaviors.
- In java, polymorphism implemented using
  - method overloading and
  - method overriding.

Ad hoc polymorphism

- The ad hoc polymorphism is a technique used to define
  - the same method with different implementations and
  - different arguments.
- In a java programming language, ad hoc polymorphism carried out with
  - a method overloading concept.

Ad hoc polymorphism

- In ad hoc polymorphism the method binding happens at the time of compilation.
- Ad hoc polymorphism is also known as compile-time polymorphism.
- Every function call binded with the respective overloaded method based on the arguments.

Ad hoc polymorphism

- The ad hoc polymorphism implemented within the class only.

Ad hoc polymorphism example-1

```
import java.util.Arrays;

public class AdHocPolymorphismExample {

        void sorting(int[] list) {
                Arrays.parallelSort(list);
                System.out.println("Integers after sort: " + Arrays.toString(list) );
        }
        void sorting(String[] names) {
                Arrays.parallelSort(names);
                System.out.println("Names after sort: " + Arrays.toString(names) );
        }
...
```

Ad hoc polymorphism example-1

```
...
        public static void main(String[] args) {

                AdHocPolymorphismExample obj = new AdHocPolymorphismExample();
                int list[] = {2, 3, 1, 5, 4};
                obj.sorting(list);      // Calling with integer array

                String[] names = {"rama", "raja", "shyam", "seeta"};
                obj.sorting(names);     // Calling with String array
        }
}
```

Pure polymorphism

- The pure polymorphism is a technique used to define the same method with the same arguments but different implementations.
- In a java programming language, pure polymorphism carried out with
  - a method overriding concept.

Pure polymorphism

- In pure polymorphism, the method binding happens at run time.
  - Pure polymorphism is also known as run-time polymorphism.
  - Every function call binding with the respective overridden method based on the object reference.
- When a child class has a definition for a member function of the parent class,
  - the parent class function is said to be overridden.

Pure polymorphism

- The pure polymorphism implemented in the inheritance concept only.

Pure polymorphism example-1

```
class ParentClass{

        int num = 10;

        void showData() {
                System.out.println("Inside ParentClass showData() method");
                System.out.println("num = " + num);
        }

}
```

Pure polymorphism example-1

```
class ChildClass extends ParentClass{

        void showData() {
                System.out.println("Inside ChildClass showData() method");
                System.out.println("num = " + num);
        }
}
```

Pure polymorphism example-1

```
public class PurePolymorphism {

        public static void main(String[] args) {

                ParentClass obj = new ParentClass();
                obj.showData();

                obj = new ChildClass();
                obj.showData();

        }
}
```

Java Method Overriding

- During inheritance in Java, if the same method is present in both the superclass and the subclass.
    - Then, the method in the subclass overrides the same method in the superclass. This is called method overriding.

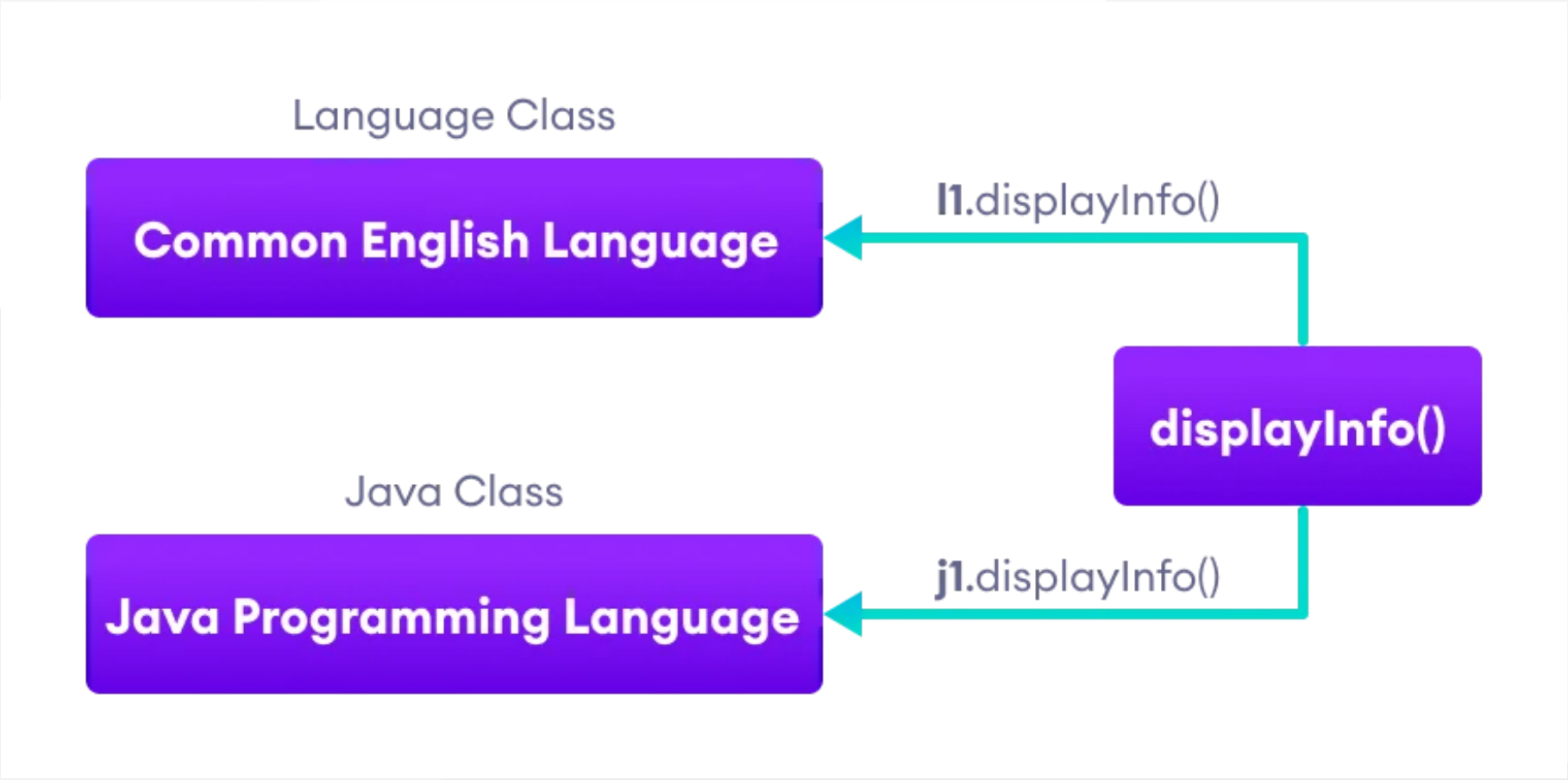
Polymorphism using method overriding example-2

```
class Language {
  public void displayInfo() {
    System.out.println("Common English Language");
  }
}

class Java extends Language {
  @Override
  public void displayInfo() {
    System.out.println("Java Programming Language");
  }
}
```

Polymorphism using method overriding example-2

```
class Main {
  public static void main(String[] args) {

    // create an object of Java class
    Java j1 = new Java();
    j1.displayInfo();

    // create an object of Language class
    Language l1 = new Language();
    l1.displayInfo();
  }
}
```

Polymorphism using method overriding example-2



Java Method Overloading

In a Java class, we can create methods with the same name if they differ in parameters. For example

```
void func() { ... }
void func(int a) { ... }
float func(double a) { ... }
float func(int a, float b) { ... }
```

This is known as method overloading in Java. Here, the same method will perform different operations based on the parameter.

Polymorphism using method overloading example-3

```
class Pattern {

  // method without parameter
  public void display() {
    for (int i = 0; i < 10; i++) {
      System.out.print("*");
    }
  }

  // method with single parameter
  public void display(char symbol) {
    for (int i = 0; i < 10; i++) {
      System.out.print(symbol);
    }
  }
}
```

Polymorphism using method overloading example-3

```
class Main {
  public static void main(String[] args) {
    Pattern d1 = new Pattern();

    // call method without any argument
    d1.display();
    System.out.println("\n");

    // call method with a single argument
    d1.display('#');
  }
}
```

Polymorphic Variables

- A variable is called polymorphic if it refers to different values under different conditions.
- Object variables (instance variables) represent the behavior of polymorphic variables in Java.
- It is because object variables of a class can refer to objects of its class as well as objects of its subclasses.

Polymorphic Variables Example-1

```
class ProgrammingLanguage {
  public void display() {
    System.out.println("I am Programming Language.");
  }
}
```

Polymorphic Variables Example-1

```
class Java extends ProgrammingLanguage {
  @Override
  public void display() {
    System.out.println("I am Object-Oriented Programming Language.");
  }
}
```

Polymorphic Variables Example-1

```
class Main {
  public static void main(String[] args) {

    // declare an object variable
    ProgrammingLanguage pl;

    // create object of ProgrammingLanguage
    pl = new ProgrammingLanguage();
    pl.display();

    // create object of Java class
    pl = new Java();
    pl.display();
  }
}
```

Java Encapsulation

Java Encapsulation

- It prevents outer classes from accessing and changing fields and methods of a class. This also helps to achieve data hiding

Java Encapsulation Example

```
class Area {

  // fields to calculate area
  int length;
  int breadth;

  // constructor to initialize values
  Area(int length, int breadth) {
    this.length = length;
    this.breadth = breadth;
  }

  // method to calculate area
  public void getArea() {
    int area = length * breadth;
    System.out.println("Area: " + area);
  }
}
```

Java Encapsulation Example

```
class Main {
  public static void main(String[] args) {

    // create object of Area
    // pass value of length and breadth
    Area rectangle = new Area(5, 6);
    rectangle.getArea();
  }
}
```

Why Encapsulation?

- In Java, encapsulation helps us to keep
  - related
    - fields and
    - methods together,
  - which makes our code cleaner and easy to read.

Why Encapsulation?

- It helps to control the values of our data fields

```
class Person {
  private int age;

  public void setAge(int age) {
    if (age >= 0) {
      this.age = age;
    }
  }
}
```

Why Encapsulation?

- The getter and setter methods provide
  - read-only or
  - write-only
- access to our class fields

```
getName()  // provides read-only access
setName() // provides write-only access
```

Why Encapsulation?

- It helps to decouple components of a system.
  - For example,
    - we can encapsulate code into multiple bundles.
- These decoupled components (bundle)
  - can be developed,
  - tested, and
  - debugged independently and concurrently.
- And, any changes in a particular component
  - do not have any effect on other components.

Why Encapsulation?

- We can also achieve data hiding using encapsulation.
- In the next example,
  - if we change the length and breadth variable into private,
  - then the access to these fields is restricted.
- And, they are kept hidden from outer classes.
  - This is called data hiding.

Why Encapsulation?

```
class Area {

  // fields to calculate area
  int length;
  int breadth;

  // constructor to initialize values
  Area(int length, int breadth) {
    this.length = length;
    this.breadth = breadth;
  }

  // method to calculate area
  public void getArea() {
    int area = length * breadth;
    System.out.println("Area: " + area);
  }
}
```

Why Encapsulation?

```
class Main {
  public static void main(String[] args) {

    // create object of Area
    // pass value of length and breadth
    Area rectangle = new Area(5, 6);
    rectangle.getArea();
  }
}
```

Data Hiding

- Data hiding is a way of restricting the access of our data members by hiding the implementation details.
- Encapsulation also provides a way for data hiding.
- We can use access modifiers to achieve data hiding

Data hiding using the private specifier example

- Making `age` private allowed us to restrict unauthorized access from outside the class. This is data hiding.

Data hiding using the private specifier example

```
class Person {

  // private field
  private int age;

  // getter method
  public int getAge() {
    return age;
  }

  // setter method
  public void setAge(int age) {
    this.age = age;
  }
}
```

Data hiding using the private specifier example

```
class Main {
  public static void main(String[] args) {

    // create an object of Person
    Person p1 = new Person();

    // change age using setter
    p1.setAge(24);

    // access age using getter
    System.out.println("My age is " + p1.getAge());
  }
}
```

Java Method Overriding

Java Method Overriding

- The method overriding is the process of re-defining a method in a child class that is already defined in the parent class.
- When both parent and child classes have the same method, then that method is said to be the overriding method.
- The method overriding enables the child class to change the implementation of the method which aquired from parent class according to its requirement.

Java Method Overriding

The method overriding is also known as

- dynamic method dispatch or
- run time polymorphism or
- pure polymorphism.

Java Method Overriding Example

```
class ParentClass{

        int num = 10;

        void showData() {
                System.out.println("Inside ParentClass showData() method");
                System.out.println("num = " + num);
        }
}
```

Java Method Overriding Example

```
class ChildClass extends ParentClass{

        void showData() {
                System.out.println("Inside ChildClass showData() method");
                System.out.println("num = " + num);
        }
}
```

Java Method Overriding Example

```
public class PurePolymorphism {

        public static void main(String[] args) {

                ParentClass obj = new ParentClass();
                obj.showData();

                obj = new ChildClass();
                obj.showData();

        }
}
```

Rules for method overriding

While overriding a method, we must follow the below list of rules.

- Static methods can not be overridden.
- Final methods can not be overridden.
- Private methods can not be overridden.
- Constructor can not be overridden.
- An abstract method must be overridden.
- Use super keyword to invoke overridden method from child class.

Rules for method overriding

- The return type of the overriding method must be same as the parent has it.
- The access specifier of the overriding method can be changed, but the visibility must increase but not decrease. For example, a protected method in the parent class can be made public, but not private, in the child class.

Rules for method overriding

- If the overridden method does not throw an exception in the parent class, then the child class overriding method can only throw the unchecked exception, throwing a checked exception is not allowed.
- If the parent class overridden method does throw an exception, then the child class overriding method can only throw the same, or subclass exception, or it may not throw any exception.

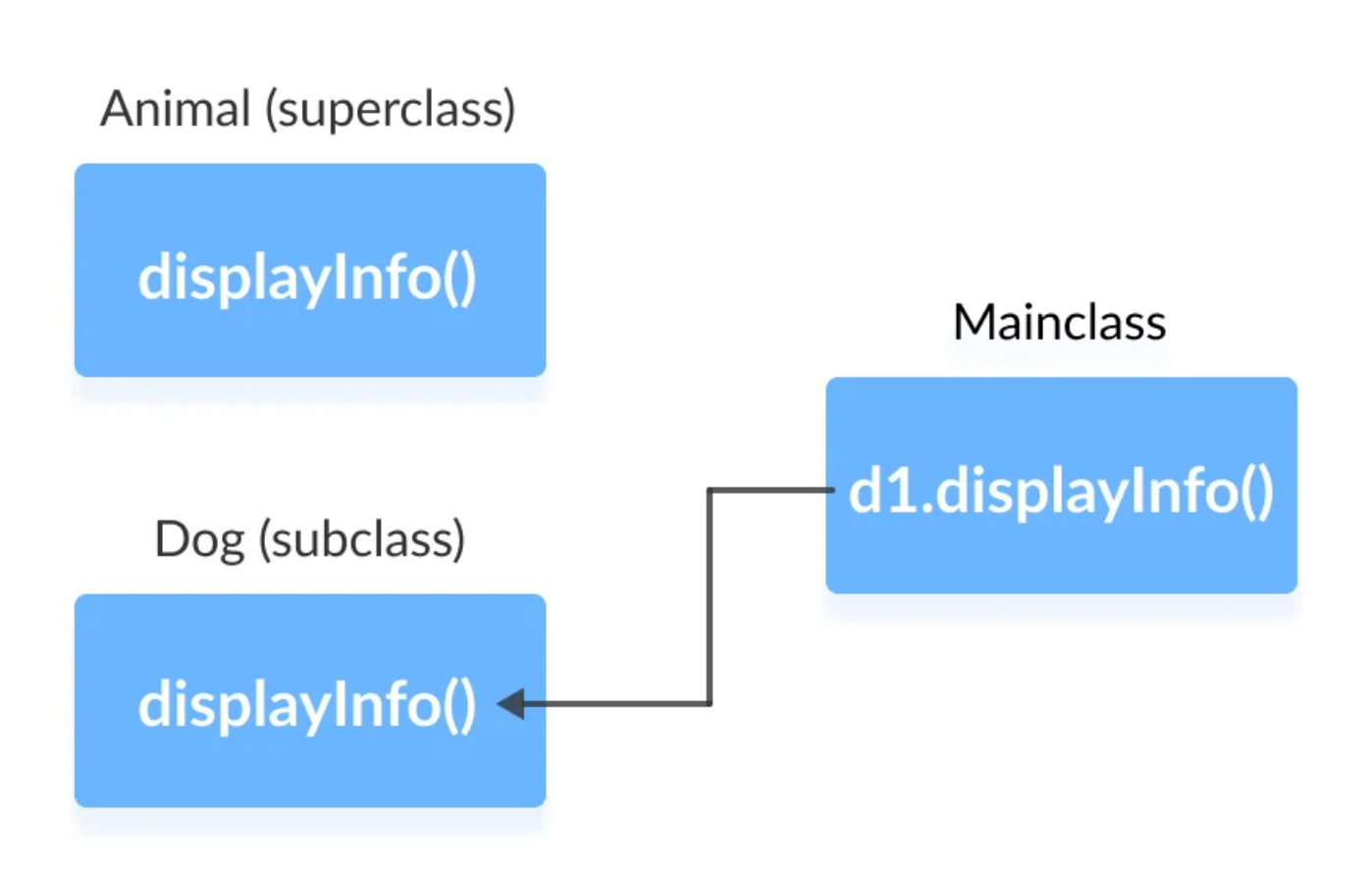
Method Overriding Example

```
class Animal {
    public void displayInfo() {
        System.out.println("I am an animal.");
    }
}

class Dog extends Animal {
    @Override
    public void displayInfo() {
        System.out.println("I am a dog.");
    }
}

class Main {
    public static void main(String[] args) {
        Dog d1 = new Dog();
        d1.displayInfo();
    }
}
```

Method Overriding Example

- annotations are the metadata that we used to provide information to the compiler
- It is not mandatory to use @Override. However, when we use this, the method should follow all the rules of overriding. Otherwise, the compiler will generate an error.

Method Overriding Example



super Keyword in Java Overriding

- Can we access the method of the superclass after overriding?
  - The answer is Yes. To access the method of the superclass from the subclass, we use the super keyword

Use of super Keyword Example

```
class Animal {
    public void displayInfo() {
        System.out.println("I am an animal.");
    }
}

class Dog extends Animal {
    public void displayInfo() {
        super.displayInfo();
        System.out.println("I am a dog.");
    }
}

class Main {
    public static void main(String[] args) {
        Dog d1 = new Dog();
        d1.displayInfo();
    }
}
```

Use of super Keyword Example

- In the above example, the subclass Dog overrides the method displayInfo() of the superclass Animal.
- When we call the method displayInfo() using the d1 object of the Dog subclass, the method inside the Dog subclass is called; the method inside the superclass is not called
- Inside displayInfo() of the Dog subclass, we have used super.displayInfo() to call displayInfo() of the superclass.

Use of super Keyword Example

- note that constructors in Java are not inherited. Hence, there is no such thing as constructor overriding in Java.
- However, we can call the constructor of the superclass from its subclasses. For that, we use super()

Access Specifiers in Method Overriding

- The same method declared in the superclass and its subclasses can have different access specifiers. However, there is a restriction.
- We can only use those access specifiers in subclasses that provide larger access than the access specifier of the superclass. For example,
- Suppose, a method myClass() in the superclass is declared protected. Then, the same method myClass() in the subclass can be either public or protected, but not private.

Access Specifier in Overriding Example

```
class Animal {
    protected void displayInfo() {
        System.out.println("I am an animal.");
    }
}

class Dog extends Animal {
    public void displayInfo() {
        System.out.println("I am a dog.");
    }
}

class Main {
    public static void main(String[] args) {
        Dog d1 = new Dog();
        d1.displayInfo();
    }
}
```

Access Specifier in Overriding Example

- In the above example, the subclass Dog overrides the method displayInfo() of the superclass Animal.
- Whenever we call displayInfo() using the d1 (object of the subclass), the method inside the subclass is called.
- Notice that, the displayInfo() is declared protected in the Animal superclass. The same method has the public access specifier in the Dog subclass.
- This is possible because the public provides larger access than the protected.

Overriding Abstract Methods

- In Java, abstract classes are created to be the superclass of other classes.
- And, if a class contains an abstract method,
  - it is mandatory to override it.

References

- BtechSmartClass-super Keyword
- Programiz-super Keyword
- BtechSmartClass-Java final Keyword
- Programiz-final Keyword
- BtechSmartClass-java Polymorphism
- Programiz-Polymorphism
- Programiz-Encapsulation
- BtechSmartClass-Java Method Overriding

References

- Programiz-Method Overriding
- Programiz-Nested Inner Class
- Programiz-Static Class
- Programiz-Anonymous Class
- Programiz-enums
- Programiz-enum constructor
- Programiz-enum string
- BtechSmartClass-Java Abstract Class
- Programiz-Abstract Classes Methods

References

- BtechSmartClass-Java Object class
- BtechSmartClass-Java Forms of Inheritance
- Programiz-Interfaces
- BtechSmartClass-Java Benefits and Costs of Inheritance
- BtechSmartClass-Java Defining Packages
- BtechSmartClass-Java Access Protection in Packages
- BtechSmartClass-Java Importing Packages

$$End - Of - Week - 2 - Module$$