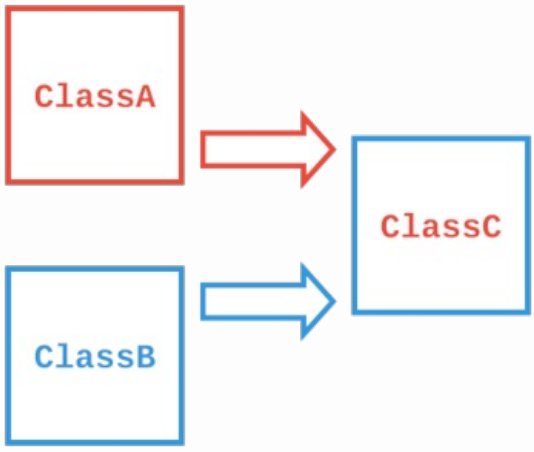
Learning Objectives - Multiple Inheritance

- **Define multiple inheritance**
- **Explain why multiple inheritance from two different classes is not allowed in Java**
- **Create multiple inheritance from a class with a superclass**
- **Define the inheritance hierarchy of an object with more than one superclass**

Multiple Inheritance

Multiple Inheritance

Multiple inheritance is a condition where a class inherits from more than one superclass. Java, however, does not allow multiple inheritance if the superclasses are of different types. The image below shows `ClassA` and `ClassB` being of different types. This causes an error message.



Multiple Superclasses

Java prohibits this kind of multiple inheritance because there is an ambiguity problem. Assume that both `ClassA` and `ClassB` have the method `greeting`. If an object of `ClassC` invokes the `greeting` method, which one does Java use? The one from `ClassA` or the one from `ClassB`? This is why a class cannot have more than one superclass of a different type.

Multilevel Inheritance

Another form of multiple inheritance is called multilevel inheritance. That is a condition where a class inherits from more than one superclass, but each superclass is the same type. The image below shows `ClassC` inheriting from `ClassB`, which in turn inherits from `ClassA`. This is allowed in Java.

Multilevel Inheritance

The classes `Carnivore` and `Dinosaur` are already defined. `Carnivore` is the superclass for `Dinosaur`. Create the `Tyrannosaurus` class which is a subclass for `Dinosaur`. The constructor for `Tyrannosaurus` takes a string and two doubles, and it calls the constructor from the `Dinosaur` class.

```
//add class definitions below this line

class Tyrannosaurus extends Dinosaur {
  public Tyrannosaurus(String d, double s, double w) {
    super(d, s, w);
  }
}

//add class definitions above this line
```

Instantiate a `Tyrannosaurus` object with the appropriate arguments. This t-rex `tiny` is 12 meters tall, weighs 14 metric tons, and eats whatever it wants. Print one of the attributes to make sure inheritance is working as expected.

```
    //add code below this line

    Tyrannosaurus tiny = new Tyrannosaurus("whatever it wants",
        12, 14);
    System.out.println(tiny.getSize());

    //add code above this line
```

challenge

Try these variations:

- Print the `weight` attribute
- Print the `diet` attribute

Extending and Overriding Methods

Extending a Class with Multiple Inheritance

Multilevel inheritance has no effect on extending a subclass. The `bonjour` method is not present in either superclass. There is no need for special syntax to extend `ClassC`. Extending a class works just like it does for single inheritance.

```
//add class definitions below this line

class ClassC extends ClassB {
  public void bonjour() {
    System.out.println("Bonjour");
  }
}

//add class definitions above this line
```

Instantiate a `ClassC` object and call the `bonjour` method.

```
    //add code below this line

    ClassC c = new ClassC();
    c.bonjour();

    //add code above this line
```

challenge

Try this variation:

- Extend `ClassC` with the method `goodbye` that prints `Goodbye`. Then call this method.
  ▼ **Solution**

```
//add class definitions below this line

class ClassC extends ClassB {
  public void bonjour() {
    System.out.println("Bonjour");
  }

  public void goodbye() {
    System.out.println("Goodbye");
  }
}

//add class definitions above this line
```

Overriding a Method with Multiple Inheritance

Like extending a class, overriding a method works the same in multilevel inheritance as it does in single inheritance. Override the `hello` method so that it prints a message.

```
//add class definitions below this line

class ClassC extends ClassB {
  public void hello() {
    System.out.println("Hello from Class C");
  }
}

//add class definitions above this line
```

Now call the `hello` method.

```
    //add code below this line

    ClassC c = new ClassC();
    c.hello();

    //add code above this line
```

Multiple Inheritance Hierarchy

Determining Inheritance Hierarchy

You can use the `instanceof` operator to determine inheritance hierarchy with multilevel inheritance just like you can with single inheritance. Create the following classes.

```
//add class definitions below this line

class ClassA {}
class ClassB extends ClassA {}
class ClassC {}
class ClassD extends ClassB {}

//add class definitions above this line
```

Now instantiate an object of type `ClassD`. Use the `instanceof` operator to determine the objects inheritance hierarchy. The program should print `true` twice because `ClassD` is a subclass of both `ClassA` and `ClassB` due to multilevel inheritance.

```
    //add code below this line

    ClassD d = new ClassD();
    System.out.println(d instanceof ClassA);
    System.out.println(d instanceof ClassB);

    //add code above this line
```

challenge

Try these variations:

- Change the print statement to:

```
    System.out.println(d instanceof ClassC);
```

▼ **Why is this an error?**
Object `d` is not an instance of `ClassC`, but Java does not return `false`, why? When there is no inheritance chain between the object and class being compared, Java returns an error.

- Change the program to be:

```
    //add code below this line

    ClassA a = new ClassA();
    System.out.println(a instanceof ClassB);

    //add code above this line
```

▼ **Why is this false?**
Object `a` is above `ClassB` in the inheritance chain. That is, `ClassB` is not a superclass to object `a`.

Overriden Methods in the Superclasses

We talked about the ambiguity problem when superclasses of different types have a method with the same name. Java does not know which one to call. With multilevel inheritance, however, this problem does not exist. The code below shows an inheritance chain where `Child` inherits from `Grandparent` and `Parent`. Both `Grandparent` and `Parent` have a method called `hello`.

```
//add class definitions below this line

class Grandparent {
  public void hello() {
    System.out.println("Hello from the Grandparent class");
  }
}

class Parent extends Grandparent {
  public void hello() {
    System.out.println("Hello from the Parent class");
  }
}

class Child extends Parent {}

//add class definitions above this line
```

Instantiate an object of type `Child` and call the `hello` method. Java prints `Hello from the Parent class`. Why? How come the ambiguity problem does not exist? When `Parent` declares the `hello` method, it overrides the `hello` method from `Grandparent`. Therefore Java uses the overriden method from `Parent` and not the method from `Grandparent`.

```
    //add code below this line

    Child c = new Child();
    c.hello();

    //add code above this line
```

Now imagine all three class have a `hello` method and that we want to access all of them from `Child`. This is possible as long as you use the `super` keyword. However, when an object of type `Child` uses `super` it is referencing `Parent`. In order for `Child` to invoke an overriden method in `Grandparent`, then `Parent` must also use the `super` keyword.

```
//add class definitions below this line

class Grandparent {
  public void hello() {
    System.out.println("Hello from the Grandparent class");
  }
}

class Parent extends Grandparent {
  public void hello() {
    System.out.println("Hello from the Parent class");
  }

  public void parentHello() {
    super.hello();
  }
}

class Child extends Parent {
  public void hello() {
    System.out.println("Hello from the Child class");
  }

  public void parentHello() {
    super.hello();
  }

  public void grandparentHello() {
    super.parentHello();
  }
}

//add class definitions above this line
```

Instantiate a `Child` object and call each of its three methods. You should see a message from the `Child`, `Parent`, and `Grandparent` classes.

```
    //add code below this line

    Child c = new Child();
    c.hello();
    c.parentHello();
    c.grandparentHello();

    //add code above this line
```