AP CSA

zhang si 张思

zhangsi@rdfz.cn

ICC 609

Inheritance——preview

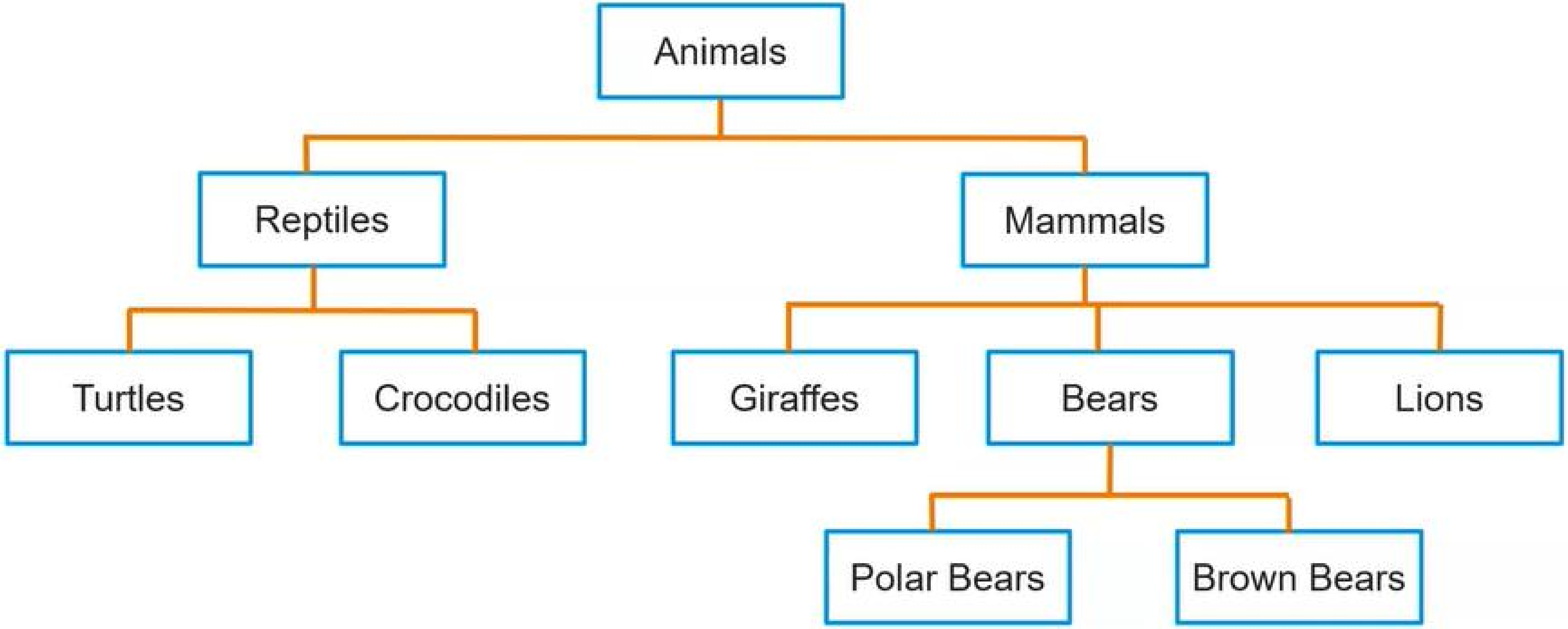
- focus on
  - ✓Creating Superclasses and Subclasses
  - ✓Creating References Using Inheritance Hierarchies
  - ✓Polymorphism

what is inheritance?

what is inheritance

- When we think of *inheritance*, we generally think of an attribute or trait that a child may receive from a parent, like eye color, athletic skill, or some other biological characteristic.
- In Java, we use inheritance to build a hierarchy of classes that have similar characteristics (This is similar to "taxonomic classification" in Biology).
- Parent classes (or **superclasses**) have attributes and behaviors that can be **inherited** by child classes (or **subclasses**).

Analogy Using Taxonomic Classification



Why do we use inheritance in java?

- Code reusability
  Higher-level classes can be used over and over again in many situations.

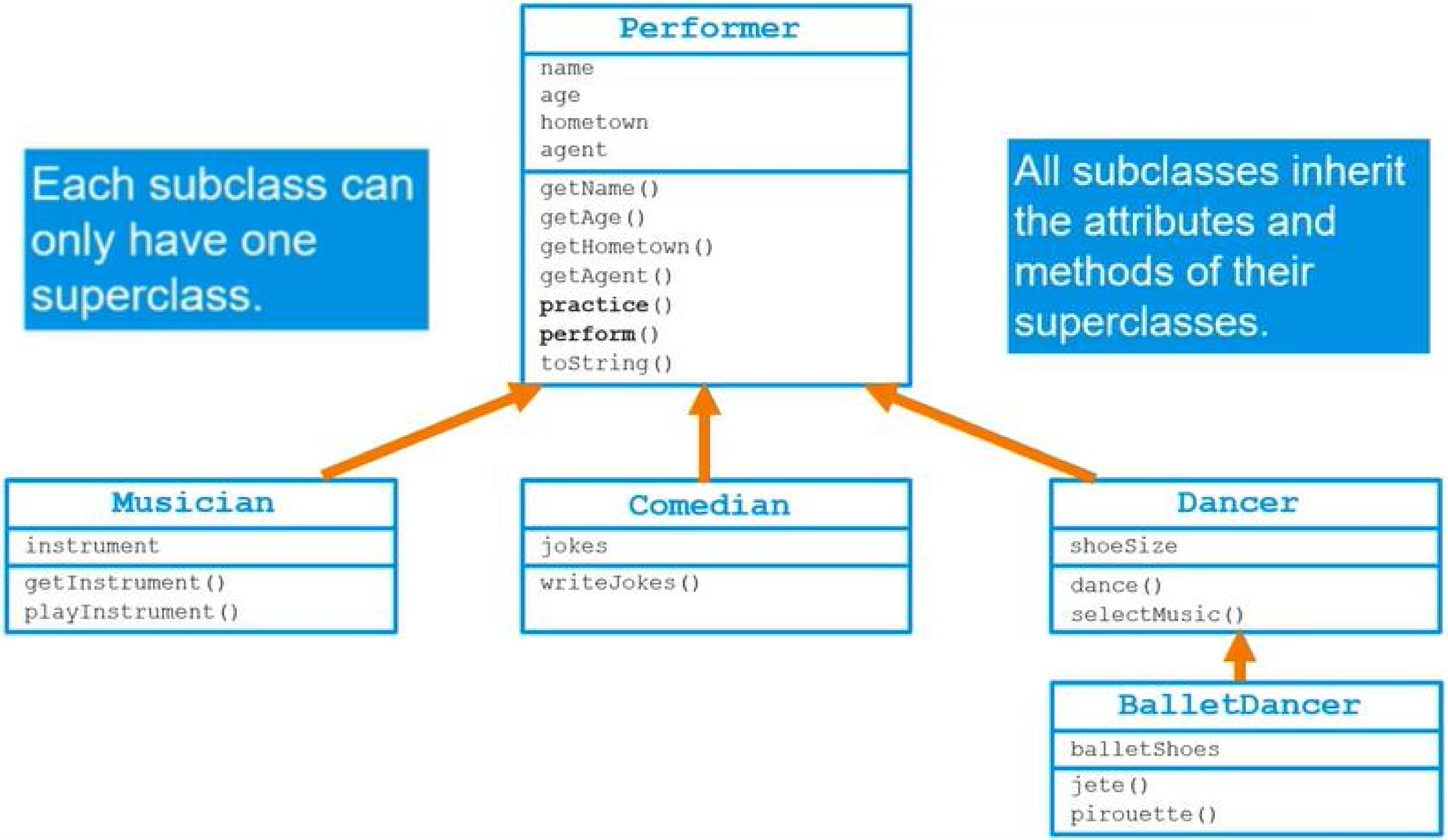
- Prevents repeating code
  Common methods and variables are now in one location, rather than many.

- Readability and organization
  Having a solid, organized structure of your classes and objects allows for greater readability and cohesion.

- Ease of maintenance
  Changing a general behavior in one place rather than many saves time and effort.

Performers,Musicians,Comedians,Dancers and Ballet Dancer

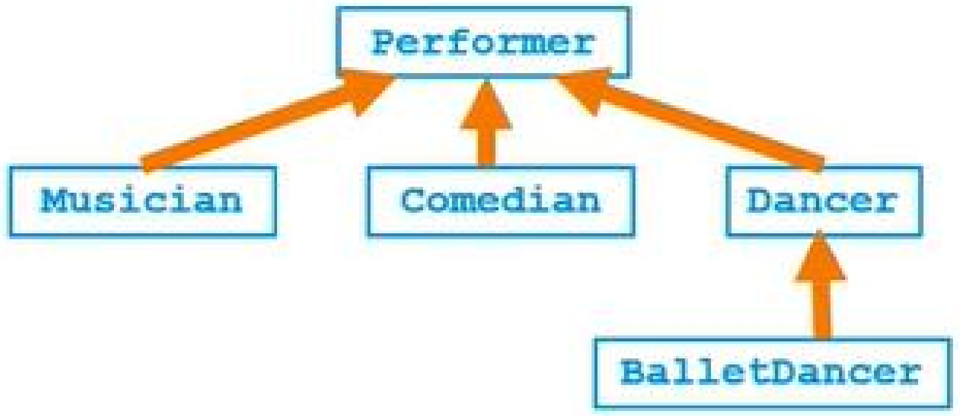
Consider a scenario where you are designing a simulation game. Among the many different types of characters in the game, there are **performers**. Performers have a name, an age, a hometown, and an agent. All performers can practice and perform. There are also many different types of performers, including **musicians**, **comedians**, **dancers**, and **ballet dancers**. Each type of performer will have the common performer characteristics, but they will also have specific attributes and behaviors that relate to their individual group.



Each subclass can only have one superclass.

All subclasses inherit the attributes and methods of their superclasses.

Superclass and subclass

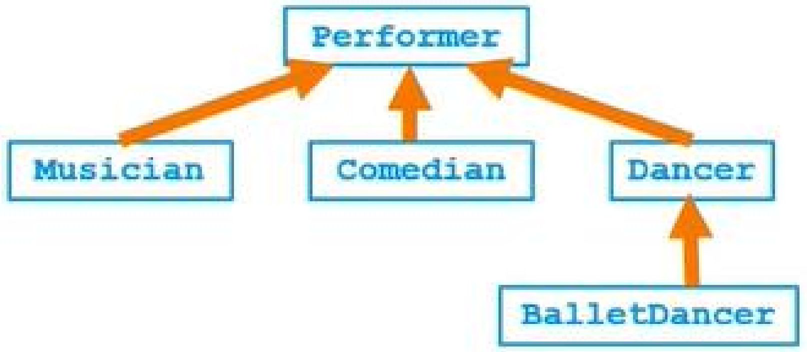


- `Musician` is a subclass of the `Performer` superclass.
- `Comedian` is a subclass of the `Performer` superclass.
- `Dancer` is a subclass of the `Performer` superclass.
- `BalletDancer` is a subclass of the `Dancer` superclass.

How do we know which way the arrows go? Using the “is-a” method!

- A **Musician** is a **Performer** YES!
- A **Comedian** is a **Performer** YES!
- A **Dancer** is a **Performer** YES!
- A **BalletDancer** is a **Dancer** YES!

- A **Comedian** is a **Dancer** NO!
- A **Performer** is a **Musician** NO!
- A **Performer** is a **Ballet Dancer** NO!
- A **Dancer** is a **Ballet Dancer** NO!



Quiz

In a particular school system, there will be two types of people: students and teachers. Both types of people will have names, ages, addresses, and ID numbers. Students will also have grade level and GPA, while teachers will have a hire date and a salary.

Which of the following would be the best class design for this scenario?

A. Three independent, unrelated classes: `Person`, `Student`, and `Teacher`

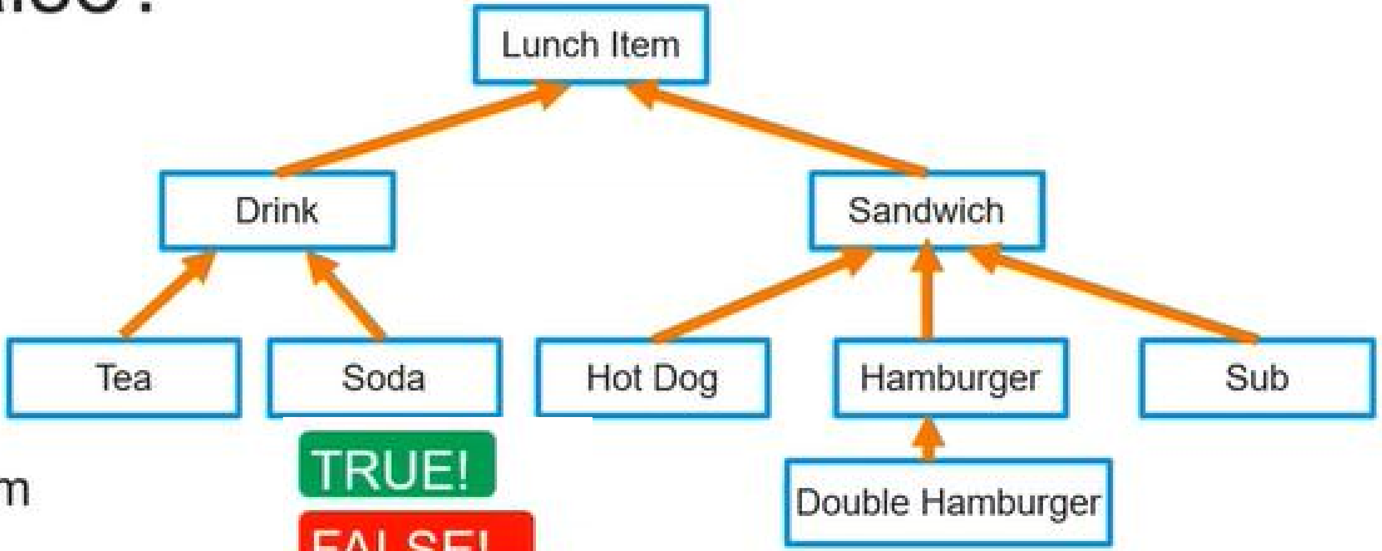
B. A `Person` superclass, and two subclasses of `Person`: `Student` and `Teacher`

C. A `Person` superclass. `Teacher` is a subclass of `Person`, and `Student` is a subclass of `Teacher`.

D. A `Student` superclass. `Teacher` is a subclass of `Student`, and `Person` is a subclass of `Teacher`.

E. Two superclasses: `Teacher` and `Student`, with `Person` as a subclass of each

Think Fast — True or False?

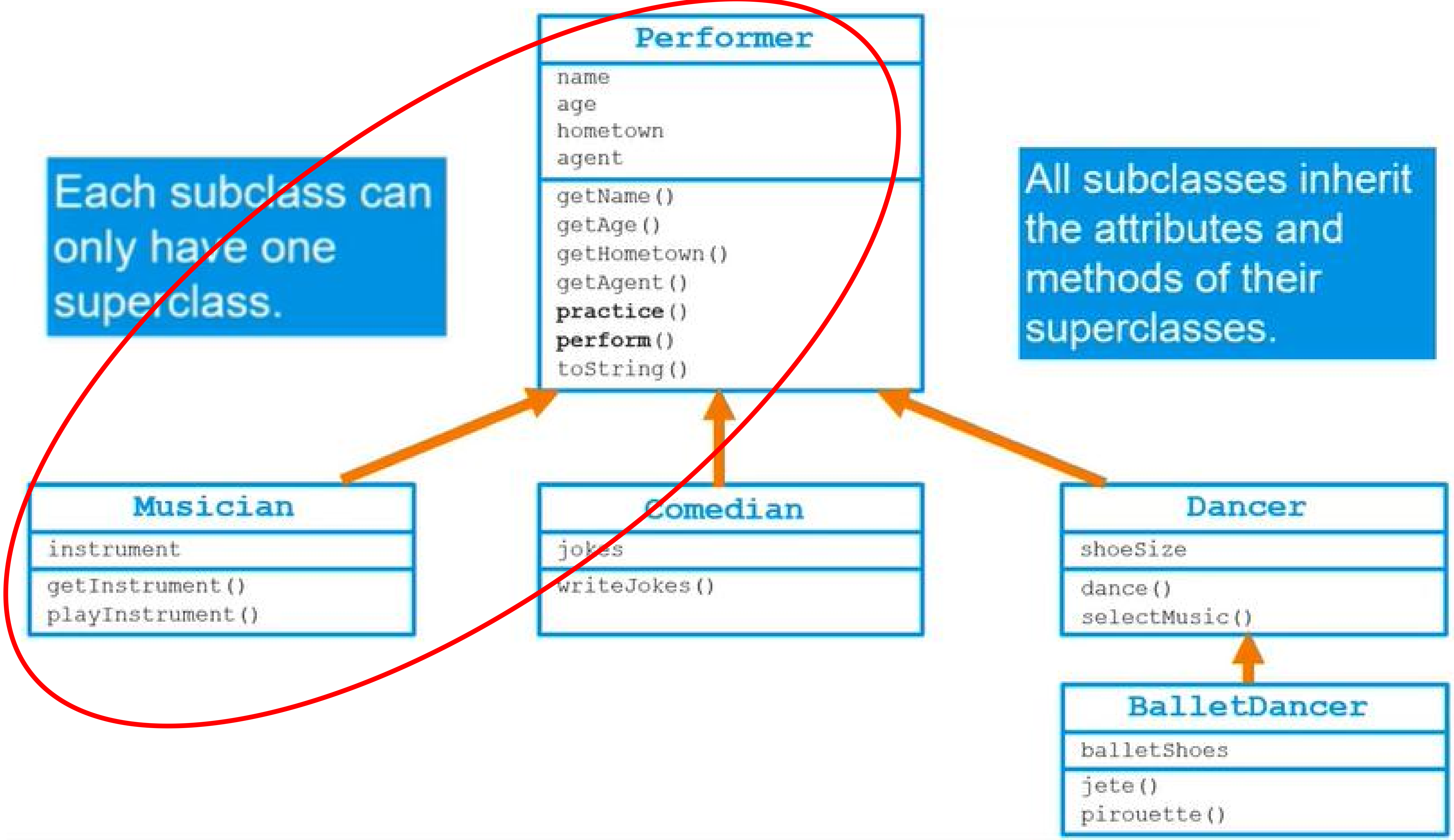


True or False: A Sandwich is a Lunch Item	TRUE!
True or False: A Drink is a Soda	FALSE!
True or False: A Hamburger is a Lunch Item	TRUE!
True or False: A Tea is a Drink	TRUE!
True or False: A Soda is a Tea	FALSE!
True or False: A Hot Dog is a Sandwich	TRUE!
True or False: A Lunch Item is a Sub	FALSE!
True or False: A Double Hamburger is a Lunch Item	TRUE!

how to realize inheritance?

Each subclass can only have one superclass.

All subclasses inherit the attributes and methods of their superclasses.



A **Musician** is a **Performer**: Using **extends**

```
public class Performer {
   private String name;
   private int age;
   public Performer() { ... }

   public String getName(){
      return name;
   }
   public void practice() {
      System.out.println("Honing my craft!");
   }
   public void perform() {
      System.out.println(
          "Performing for an audience!");
   }
}
```

```
public class Musician extends Performer{
   private String instrument;

   public Musician() { ... }

   public String getInstrument(){
      return instrument;
   }
   public void playInstrument(){
      System.out.println(
          "Making music with my " + instrument);
   }
}
```

A **`Musician`** is a **`Performer`**: Using **`extends`**

```
public class Performer {
   public String getName(){ ... }
   public void practice(){... }
   public void perform(){... }
}
```

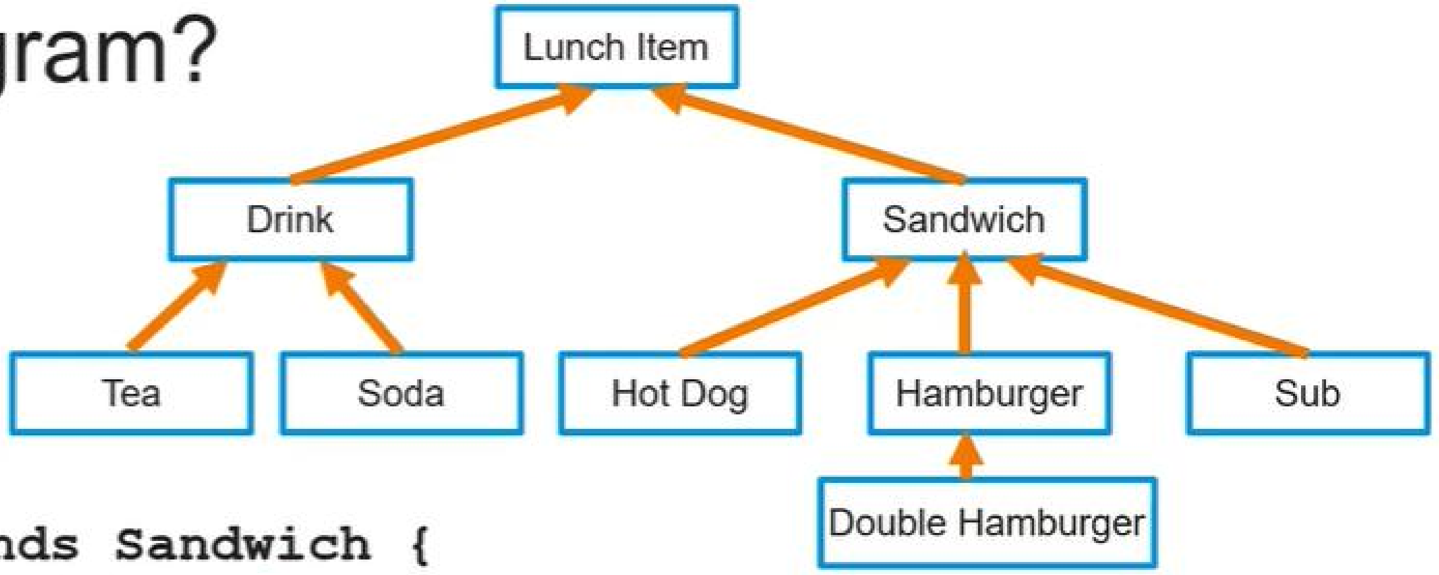
```
public class Musician extends Performer{
   public String getInstrument(){... }
   public void playInstrument(){... }
}
```

The subclass inherits all methods and attributes of the superclass without duplicating code.

```
Performer wynton = new Performer();
Musician branford = new Musician();
```

```
branford.getInstrument();    ✓          wynton.getName();    ✓
branford.playInstrument();   ✓          wynton.practice();   ✓
branford.getName();   ✓                 wynton.perform();    ✓
branford.practice();  ✓                 wynton.getInstrument();   ✗
branford.perform();   ✓                 wynton.playInstrument();  ✗
```

Which are valid class header declarations for this diagram?



- `public class Hamburger extends Sandwich {`
- `public class Tea extends Drink {`
- `public class Drink extends LunchItem {`
- `public class Sandwich extends HotDog {`
- `public class DoubleHamburger extends Sandwich {`
- `public class Soda extends LunchItem {`
- `public class DoubleHamburger extends Hamburger {`

Creating Superclasses and Subclasses

- ESSENTIAL KNOWLEDGE
- A class hierarchy can be developed by putting common attributes and behaviors of related classes into a single class called a superclass.
- Classes that extend a superclass, called subclasses, can draw upon the existing attributes and behaviors of the superclass without repeating these in the code.
- Extending a subclass from a superclass creates an “is-a” relationship from the subclass
- The keyword extends is used to establish an inheritance relationship between a subclass and a superclass. A class can extend only one superclass.

about overriding method

Iheritance and Methods

What are our options with methods when we extend a superclass?

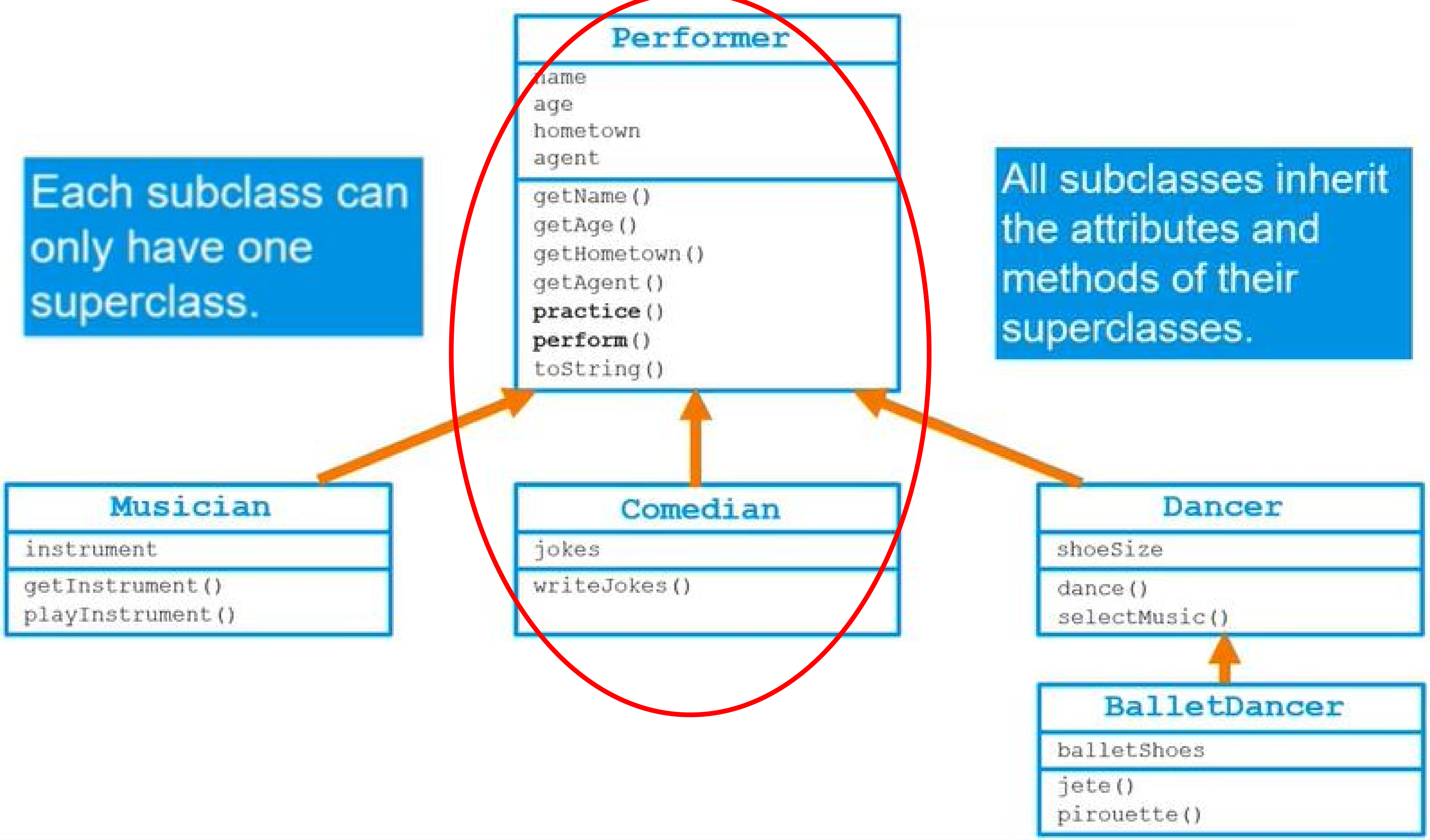
- **Inherit methods**: Any public methods in the superclass become valid public methods of the subclass. These are especially important to access private instance variables of the superclass.

- **Write new methods**: The subclass can have additional methods that are completely independent of methods in the superclass. This includes methods that are overloaded (same method name, but different signatures) and treated as independent methods.

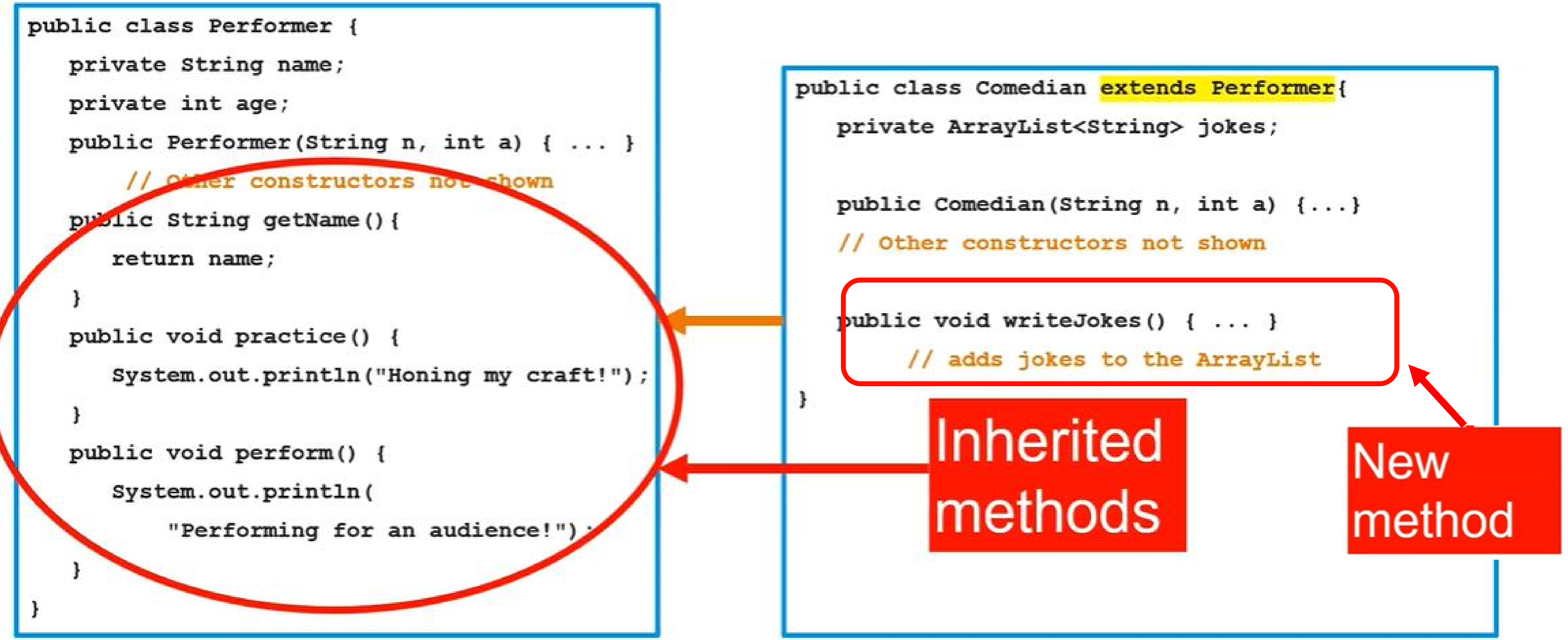
- **Override methods**: Write a new and different implementation of a method that already exists in the superclass.

Each subclass can only have one superclass.

All subclasses inherit the attributes and methods of their superclasses.



A `Comedian` is a `Performer`: Using `extends`



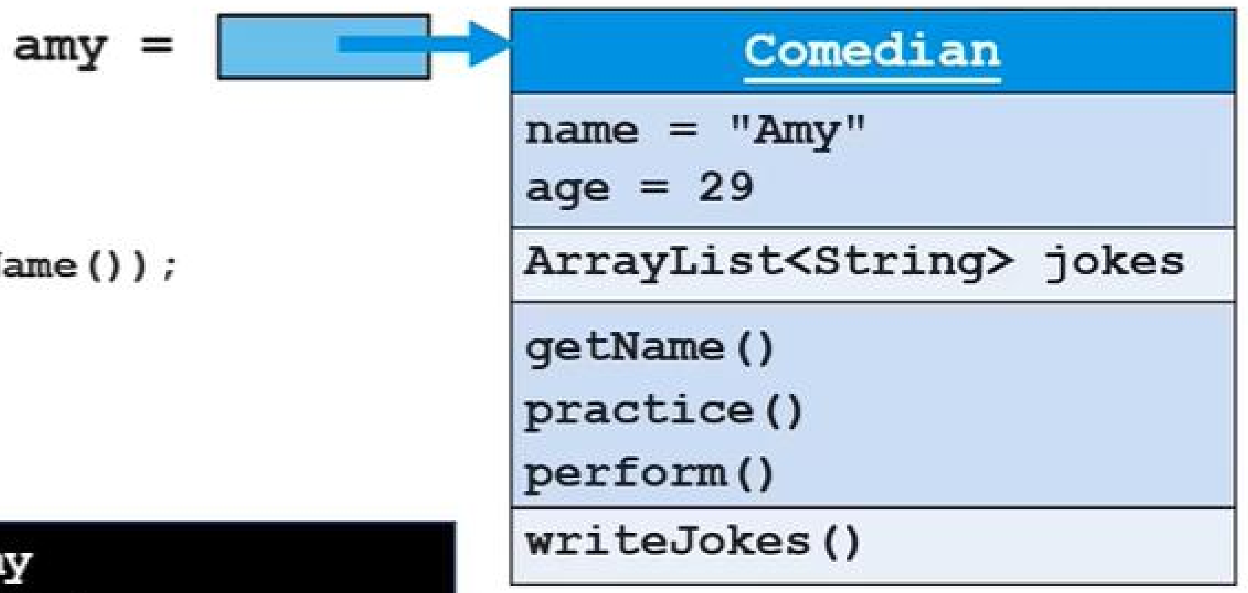
```
public class Performer {
   private String name;
   private int age;
   public Performer(String n, int a) { ... }
       // Other constructors not shown
   public String getName(){
      return name;
   }
   public void practice() {
      System.out.println("Honing my craft!");
   }
   public void perform() {
      System.out.println(
          "Performing for an audience!");
   }
}
```

```
public class Comedian extends Performer{
   private ArrayList<String> jokes;

   public Comedian(String n, int a) {...}
   // Other constructors not shown

   public void writeJokes() { ... }
       // adds jokes to the ArrayList
}
```

```
public class Performer {
   public String getName(){... }
   public void practice() {... }
   public void perform() {... }
```

```
public class Comedian extends Performer{
   public void writeJokes(){... }
```

```
public static void main(String[] args)
{
   Comedian amy = new Comedian("Amy", 29);

   System.out.println("My name is " + amy.getName());
   amy.writeJokes();
   amy.practice();
   amy.perform();
}
```



```
My name is Amy
Honing my craft!
Performing for an audience!
_
```

Overriding a Method

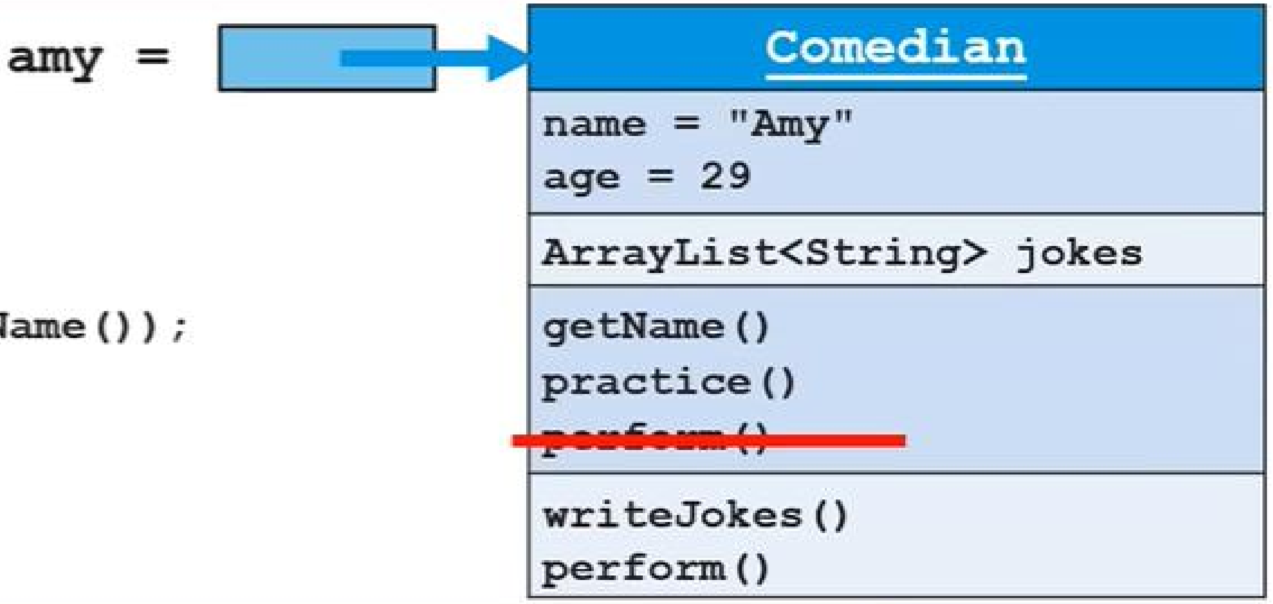
```
public class Performer {
   private String name;
   private int age;
   public Performer(String n, int a) { ... }
       // Other constructors not shown
   public String getName(){
      return name;
   }
   public void practice() {
      System.out.println("Honing my craft!");
   }
   public void perform() {
      System.out.println(
          "Performing for an audience!");
   }
}
```

```
public class Comedian extends Performer{
   private ArrayList<String> jokes;

   public Comedian(String n, int a) {...}
      // Other constructors not shown

   public void writeJokes() { ... }
        // adds jokes to the ArrayList

   public void perform() {
      for (String joke : jokes)
         System.out.println(joke);
   }
}
```

```java
public class Performer {
   public String getName(){... }
   public void practice() {... }
   public void perform() {... }
```

```java
public class Comedian extends Performer{
   public void writeJokes(){... }
```

```java
public static void main(String[] args)
{
   Comedian amy = new Comedian("Amy", 29);
   System.out.println("My name is " + amy.getName());
   amy.writeJokes();
   amy.practice();
   amy.perform();
}
```



```
My name is Amy
Honing my craft!
My software doesn't have bugs - it has "unintentional features"!
I did great on my 15-point hexadecimal test - I got an "F"!
How do you cut a program in half?  With a C-saw!
```

```java
public class Taxi extends Car{
    public int getNumPassengers() {
        return 6;
    }
    public boolean usedForBusiness(){
        return true;
    }
```

```java
public class Car {
    public String drive() {
        return "Driving around town";
    }
    public String getColor() {
        return "Yellow";
    }
    public boolean usedForBusiness() {
        return false;
    }
}
```

```java
public class SportsCar extends Car{
    public String drive() {
        return "Driving FAST";
    }
    public String getColor() {
        return "Red";
    }
```

```java
Car auto = new Car();
Taxi myCab = new Taxi();
SportsCar speedy = new SportCar();
```

	Valid?
`System.out.println(auto.drive());`	✓
`System.out.println(speedy.drive());`	✓
`System.out.println(auto.getColor());`	✓
`System.out.println(myCab.getColor());`	✓
`System.out.println(speedy.getColor());`	✓
`System.out.println(myCab.getNumPassengers());`	✓
`System.out.println(speedy.getNumPassengers());`	✗
`System.out.println(myCab.usedForBusiness());`	✓
`System.out.println(speedy.usedForBusiness());`	✓

Output?

```
Driving around town
Driving FAST
Yellow
Yellow
Red
6
✗
true
false
```

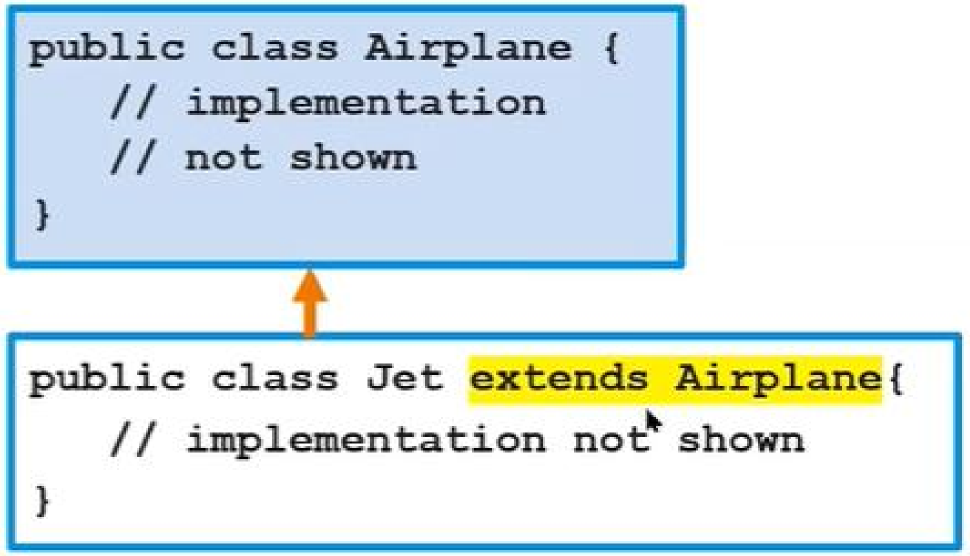
Multiple-Choice Practice

Consider the class definitions shown to the right.

The following code segment appears in a method in a class other than `Airplane` or `Jet`:

```
Jet myPersonalJet = new Jet();
myPersonalJet.getFuelLevel();
```

Which of the following must be true so that the code segment will compile without error?



```
public class Airplane {
   // implementation
   // not shown
}
```

```
public class Jet extends Airplane{
   // implementation not shown
}
```

Ⓐ The `Airplane` class must have a public method named `getFuelLevel` that takes no parameters.

Ⓑ The `Jet` class must have a public method named `getFuelLevel` that takes no parameters.

Ⓒ Both the `Airplane` class and the `Jet` class must have a public method named `getFuelLevel` that takes no parameters.

Ⓓ Either the `Airplane` class or the `Jet` class (or both) must have a public method named `getFuelLevel` that takes no parameters.

Overriding method

- Essential knowledge
- Method overriding occurs when a public method in a subclass has the same method signature as a public method in the superclass.
- Any method that is called must be defined within its own class or its superclass.
- A subclass is usually designed to have modified (overridden) or additional methods or
- instance variables.
- A subclass will inherit all public methods from the superclass; these methods remain public in the subclass

Vocabulary

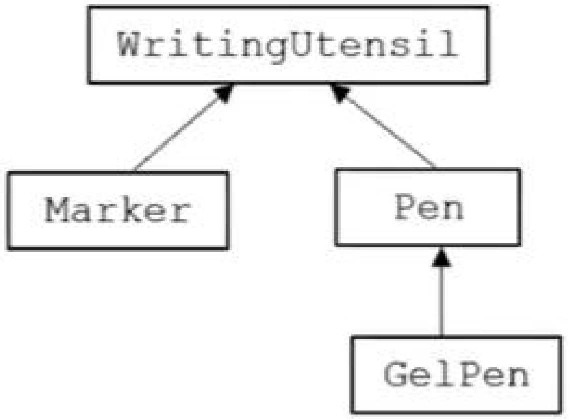
- A class that extends another class is called a **subclass**. The class being extended is called the **superclass**.
  - A subclass demonstrates an "**is-a**" relationship with the superclass.

- A reference variable is **polymorphic** when it can refer to objects from different classes at different points in the code.
  - A reference variable can store a reference to its declared class or to any subclass of its declared class.

Inheritance Hierarchy

Consider the following class declarations:

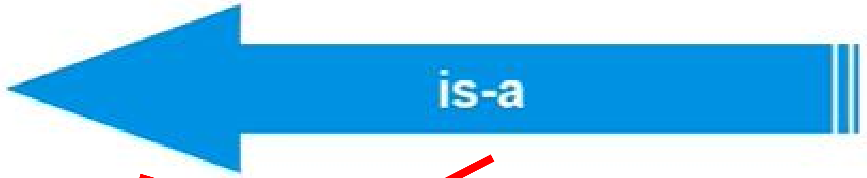
```
public class WritingUtensil{}

public class Marker extends WritingUtensil{}

public class Pen extends WritingUtensil{}

public class GelPen extends Pen{}
```



In some other class, assuming each of the above classes contains a parameterless constructor:

```
WritingUtensil writer2 = new Marker();
WritingUtensil writer3 = new Pen();
Pen writer4 = new GelPen();
WritingUtensil writer5 = new GelPen();
```

Marker is-a WritingUtensil
Pen is-a WritingUtensil
GelPen is-a Pen
GelPen is-a WritingUtensil



~~GelPen bad = new Pen();~~

The power of polymorphic variables

Why would we declare a variable using a superclass if we plan to store a reference to a subclass object?

- A collection (array or `ArrayList`) needs to be declared as a datatype.
- What do we store in a pencil case?
    - **All** of our writing utensils!

```
WritingUtensil [] pencilCase = new WritingUtensil[3];
pencilCase[0] = new Pen();
pencilCase[1] = new GelPen();
pencilCase[2] = new Marker();
```

- This in turn supports polymorphism. Each of these writing utensils likely displays writing in a different way. Let's say the behavior is implemented through a method **`public void write(String text)`** in the **`WritingUtensil`** class, which is overridden appropriately in each subclass.

```
for(WritingUtensil wu: pencilCase)
   wu.write("Hello!");
```

Exercise

Consider the following classes:

```
public class Dwelling{
     private int numBedrooms;
     private int numBaths;
     public Dwelling(int bed, int bath){
          numBedrooms = bed;
          numBaths = bath;
     }
     //other methods not shown
}

public class Appartment extends Dwelling{
     private int leaseLength;
     public Apartment(int lease, int bed, int bath){
          super(bed, bath);
          leaseLength = lease;
     }
     //other methods not shown
}
```

Consider the following declarations which occur in a different class:

```
Dwelling[] list = new Dwelling[3];
list[0] = new Dwelling(2, 1);
list[1] = new Apartment(12, 1, 1);
Apartment apt = new Dwelling(3, 2);
list[2] = apt;
```

Which of the following reasons best describes why the code does not compile?

(A) References to **`Apartment`** objects cannot be stored in **`list`**.

(B) An **`Apartment`** object must be constructed with two parameters, not three parameters.

(C) **`apt`** cannot be stored in **`list[2]`**, because an array cannot contain a reference to an object that is referenced by another variable.

(D) **`apt`** cannot store a reference to a **`Dwelling`** object.

(E) **`list[0]`** causes an **`IndexOutOfBoundsException`**, because the valid indices of an array are **`1`**, **`2`**, and **`3`**.

Exercise

Consider the following classes:

```
public class Dwelling{
    private int numBedrooms;
    private int numBaths;
    public Dwelling(int bed, int bath){
        numBedrooms = bed;
        numBaths = bath;
    }
    //other methods not shown
}
public class Appartment extends Dwelling{
    private int leaseLength;
    public Apartment(int lease, int bed, int bath){
        super(bed, bath);
        leaseLength = lease;
    }
    //other methods not shown
}
```

A different class contains the following method:

```
public void placeAd(Dwelling d){
    //code not shown
}
```

Which of the following method calls will not compile?

(A) `placeAd(new Dwelling(3, 1));`

(B)
```
Dwelling d = new Apartment(30, 4, 2);
placeAd(d);
```

(C)
```
Apartment a = new Apartment(24, 2, 1);
placeAd(a);
```

(D) `placeAd(new Apartment(36, 3, 1));`

(E)
```
Apartment a2 = new Dwelling(2, 1);
placeAd(a2);
```

Authored by: Sage Miller, 2020

Polymorphism & Method Parameters

Consider the following class hierarchy:

```
public class WritingUtensil{}

public class Marker extends WritingUtensil{}

public class Pen extends WritingUtensil{}

public class GelPen extends Pen{}
```

`Marker` is-a `WritingUtensil`
`Pen` is-a `WritingUtensil`
`GelPen` is-a `Pen`
`GelPen` is-a `WritingUtensil`

Now consider a method in a different class that contains `WritingUtensil` parameter:

```
public void displayText(WritingUtensil wu, String text){
    //implementation not shown
}
```

Due to the inheritance relationship between these classes, this method can be given **any** of the above objects!

```
WritingUtensil wtg = new WritingUtensil();
displayText(wtg, “hello”);
```

```
Marker m = new Marker();
displayText(m, “hello”);
```

```
GelPen gp = new GelPen();
displayText(gp, “hello”);
```

```
Pen p = new Pen();
displayText(p, “hello”);
```

Creating References Using Inheritance Hierarchies

- ESSENTIAL KNOWLEDGE
- When a class **S** “is-a” class **T**, **T** is referred to as a superclass, and **S** is referred to as a subclass.
- If **S** is a subclass of **T** of type **S** to a reference of type , then assigning an object **T** facilitates polymorphism.
- If **S** is a subclass of **T**, then a reference of type **T** can be used to refer to an object of type **T** or **S.**
- Declaring references of type **T**, when **S** is a subclass of **T**, is useful in the following declarations:
  - Formal method parameters
  - arrays — **T[]** var ArrayList<**T**> var

Polymorphism

- A reference variable is **polymorphic** when it can refer to objects from different classes at different points in the code.
    - A reference variable can store a reference to its declared class or to any subclass of its declared class.

- A method is considered **polymorphic** when it is overridden in at least one subclass.

- **Polymorphism** is the act of executing an overridden non-`static` method from the correct class at runtime based on the actual object type.

which method calls compile?
which methods execute?

The keyword `super` can be used to call a superclass’ s constructors and methods.

```
public class Entertainer{
     private String talent;
     public Entertainer (String t){
          talent = t;
     }
     public String getTalent(){
          return talent;
     }
}

public class Comedian extends Entertainer{
     private ArrayList<String> jokes;
     public Comedian(String t, ArrayList<String> jks){
          super(t);
          jokes = jks;
     }
     public String getTalent(){
          return "Comedy style: " + super.getTalent();
     }
     public String tellJoke(){
          return jokes.get((int)(Math.random()*jokes.size()));
     }
}
```

```
Entertainer fred = new Entertainer("musician");
System.out.println(fred.getTalent());
~~System.out.println(fred.tellJoke());~~
```

- since fred was declared as Entertainer type, and it also references an Entertainer.
  - ✓ at compile time, the compiler make sure that `getTalent()` exists in the Entertainer class
  - ✓ At runtime getTalent() is executed from the Entertainer class
- what about `telljoke()`, can it compile?

Authored by: Sage Miller, 2020

which method calls compile?
which methods execute?

```
public class Entertainer{
     private String talent;
     public Entertainer (String t){
          talent = t;
     }
     public String getTalent(){
          return talent;
     }
}

public class Comedian extends Entertainer{
     private ArrayList<String> jokes;
     public Comedian(String t, ArrayList<String> jks){
          super(t);
          jokes = jks;
     }
     public String getTalent(){
          return "Comedy style: " + super.getTalent();
     }
     public String tellJoke(){
          return jokes.get((int)(Math.random()*jokes.size()));
     }
}
```

```
ArrayList<String> oneLiners = new ArrayList<String>();
//code to add jokes to oneLiners
Entertainer sally = new Comedian("satire", oneLiners);
System.out.println(sally.getTalent());

System.out.println(((Comedian)sally).tellJoke());
```

- since sally was declared as `Entertainer` type, and it also references an `Comedian`.
  - ✓ at compile time, the compiler make sure that `getTalent()` exists in the Entertainer class
  - ✓ At runtime `getTalent()` is executed from the Comedian class
- what if asked sally to tell joke?

downcasting

Practice

```
public class Entertainer{
    private String talent;
    public Entertainer (String t){
        talent = t;
    }
    public String getTalent(){
        return talent;
    }
}

public class Comedian extends Entertainer{
    private ArrayList<String> jokes;
    public Comedian(String t, ArrayList<String> jks){
        super(t);
        jokes = jks;
    }
    public String getTalent(){
        return "Comedy style: " + super.getTalent();
    }
    public String tellJoke(){
        return jokes.get((int)(Math.random()*jokes.size()));
    }
}
```

```
ArrayList<String> jokeList = new ArrayList<String>();
//code to add jokes to jokeList
Entertainer talisa = new Comedian("parody", jokeList);
System.out.println(talisa.tellJoke());
```

The above segment of code occurs in a class other than **`Entertainer`** or **`Comedian`** and is intended to display a random joke from **`talisa`**'s **`jokes`**. Which of the following explains why it does not?

(A) A compile-time error occurs because **`talisa`** is declared as type **`Entertainer`** but instantiated as type **`Comedian`**.

(B) A compile-time error occurs because **`tellJoke()`** is not defined for the **`Entertainer`** object **`talisa`**.

(C) A run-time error could occur, because the random number generated in **`tellJoke()`** might not be a valid index in **`jokes`**.

(D) A compile-time error occurs because **`jokes`** is a **`private`** data member and cannot be accessed in **`tellJoke()`**.

(E) A run-time error occurs because **`tellJoke()`** is not defined for the **`Entertainer`** object **`talisa`**.

Authored by: Sage Miller, 2021

Practice

```
public class Parent{
    public void display1(){
        System.out.print("P");
    }
    public void display2(){
        System.out.print("W");
    }
}

public class Child extends Parent{
    public void display1(){
        super.display1();
        System.out.print("C");
    }
    public void display2(){
        System.out.print("X");
    }
    public void display3(){
        System.out.print("Y");
    }
}
```

In a different class:

```
    Parent obj = new Child();
    obj.display1();
    obj.display2();
```

What is displayed as a result of executing the above segment of code?

(A) PW
(B) PCW
(C) CX
(D) PCX
(E) CW

Authored by: Sage Miller, 2020

Polymorphism

- ESSENTIAL KNOWLEDGE
- Utilize the Object class through inheritance.
- **At compile time**, methods in or inherited by the declared type determine the correctness of a non-static method call.
- **At run-time**, the method in the actual object type is executed for a non-static method call.