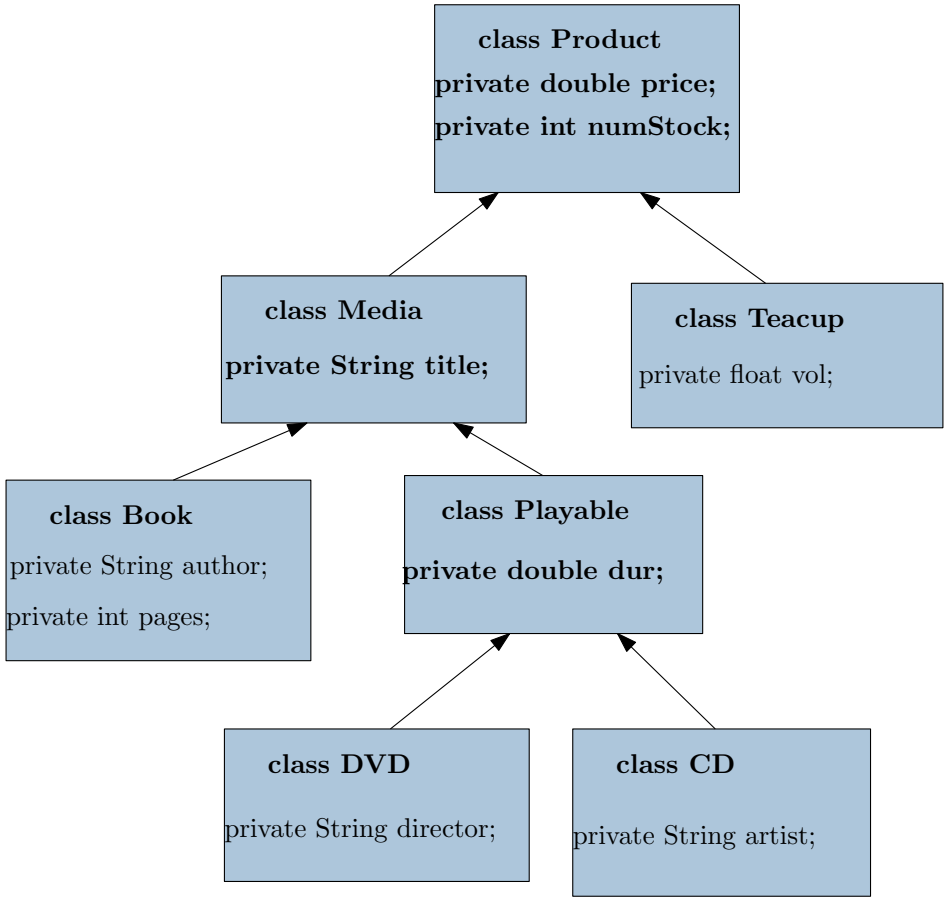
Inheritance II

Daniel Archambault

Previously in CS-115



Hierarchies avoid duplication!

Previously in CS-115

- Why bother with hierarchies and inheritance?

Previously in CS-115

- Why bother with hierarchies and inheritance?
- How do we allow attributes and methods to be confined to the hierarchy?

Previously in CS-115

- Why bother with hierarchies and inheritance?
- How do we allow attributes and methods to be confined to the hierarchy?
- What keyword defines an inheritance relationship?

Previously in CS-115

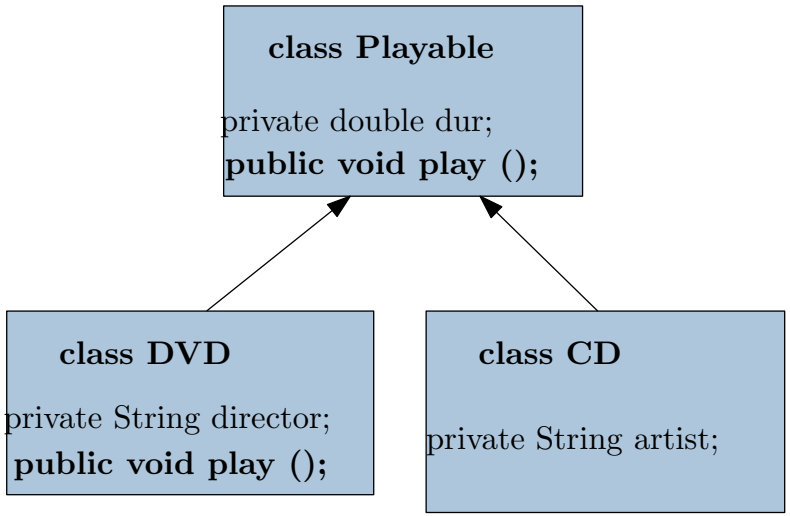
- Why bother with hierarchies and inheritance?
- How do we allow attributes and methods to be confined to the hierarchy?
- What keyword defines an inheritance relationship?
- What is `super`? Why do we need to call it in constructors?

Previously in CS-115

- Time to finish up with objects

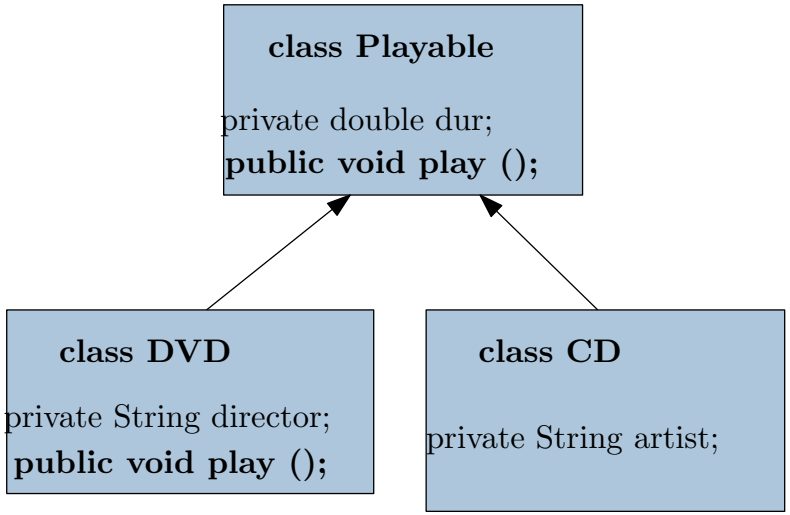
Inheritance II

Hierarchies Can Cause Name Conflicts



- Suppose `Playable` and `DVD` both have a `play ()` method with the exact same return type and parameters
- Why would you even want to do this in the first place?
  - define default behaviour for `Playable` objects
  - conversely define more specific behaviour for `DVD`
- What method gets called when you do a `p.play ()`?

Method Overriding



- In this case, `play ()` in `DVD` overrides the version in `Playable`
  - thus, DVD provides more specific behaviour for `play ()`
- Independent of the type of the reference, the `play ()` method in DVD gets called
  - because that's the "real" type of the object

Getting the Director of a DVD...

- Suppose there is a method `public String getDirector ()` in `DVD`. Is the following valid?

```
Playable p = new DVD ();
String s = p.getDirector ();
```

- No it is not. The type of the reference does not have a `getDirector ()` method
- How can we transform this `DVD` back into it's true form?

Reference Casting

- You can use casting to transform the type of references
- For any object, you can cast it into it's own type or any of it's superclasses

```
Playable p = new DVD ();
DVD d = (DVD) p; //change reference into a DVD
String s = d.getDirector ();
```

- We could cast the following references into a `DVD` (assuming the object really is a `DVD`):
  - `Product`, `Media`, `Playable`
- But, not `Book` or any unrelated types
- A casting error is thrown `ClassCastException`

What is `Object`?

- `Object` is a very useful reference type in Java
- `Object` can refer **any** object in java
  - it is the root of all class hierarchies
- What does `Object` allow us to do?
  - using `Object` you can define programs that work on any type

Method Overloading

- Overloading allows us to define different parameters for the same method
- For example, suppose we want to set the components of a colour
- Sometimes, the RGB components are defined from $[0, 1]$
  - `void setColour (float r, float g, float b)`
- Sometimes, the RGB components are integers between $[0, 255]$
  - `void setColour (int r, int g, int b)`
- The method is *overloaded* because it can take different parameters
- Parameters must be different, simple return type change won't work

Constructor Overloading

- Overloading can be very useful for constructors
- We use `this` like `super` in this case

```
public class Colour {
     private float red; // 0.0 to 1.0
     private float green; // 0.0 to 1.0
     private float blue; // 0.0 to 1.0

     public Colour(float r, float g, float b) {
          this.red = r;
          this.green = g;
          this.blue = b;
     }

     // Create a colour representing black.
     public Colour() {
          this(0.0, 0.0, 0.0);
     }

     // Constructor allowing 0 to 255 values for r, g, b
     public Colour(int r, int g, int b) {
          this(r / 255.0, g / 255.0, b / 255.0);
     }
}
```

Abstract Classes

- Sometimes you want to force the implementation of a method
- But, don't know how to implement it just yet
  - i.e. draw a picture of the product on screen
- Abstract classes allow us to defer the implementation to the person extending the code
- Ensures that it is implemented properly by the person extending the class
- Abstract classes cannot be instantiated (not complete)
- But complete extensions can be

Abstract Classes in Java

```
public abstract class Product {
    private double price;
    private int numStock
    ...

    public double getPrice() {
       return price;
    }
   /**
    *Draws the product to the screen.  Don't know
    *how to do it yet.  Subclass will tell me.
    */
    public abstract void draw();
}
```

Abstract Classes in Java

```
public class Teacup extends Product {
    private int volume;
    ...
    /**
     *Now I implement the draw method.  If I don't, I
     *get a compile error if I try to do new Teacup.
     */
     public void draw () {
         ... draw my Tea cup ...
     }
}
```

Interfaces in Java

- Sometimes you want to enforce certain operations but don't want a hierarchy
- In this case, an interface (not GUI) is a solution
- In an interface, you only have abstract methods
- You have zero data
- An interface is the closest thing to an ADT in Java

Interfaces in Java

```
public Interface List {
    public boolean isEmpty ();
    public int numberOfElements ();
    public void addItem (ListItem li, int pos);
    public ListItem getItem (int pos);
}
```

- Use **implements** keyword to have your class implement the interface
- If you implement the interface, you need to provide an implementation for all the methods
- If you don't, compile error.
- You can implement many interfaces, but only inherit from one superclass

Interfaces as References

- An interface can act as a reference type
- But can't be instantiated

```
public class MyArray implements List {
      ...
}

//In the outside world
//Can only call list methods on this object

List a = new MyArray (...); //new List (...)
                            //never allowed
```