Relationships between Objects

Generalisations, Composition and Aggregation

COMP2603
Object Oriented Programming 1

Week 3, Lecture 2

Outline

- Types of Relationships in Object-Oriented Programming
  - Dependencies/Associations
  - Generalisations/Specialisations
  - Aggregration/Composition
- Implementing Relationships
  - Dependencies/Associations
  - Generalisations/Specialisations
  - Aggregration/Composition

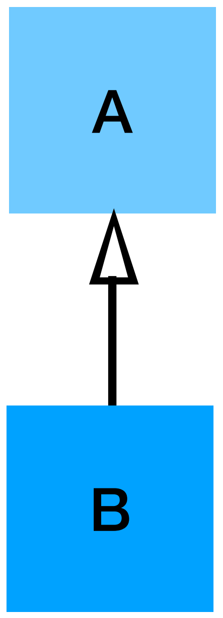
Inheritance

Inheritance is a relationship among classes wherein one class shares the **structure** and/or **behaviour** defined in one (single inheritance) or more (multiple inheritance) other classes (Booch 1994).

A class from which another class inherits its structure and/or behaviour is called the **superclass.** A class that inherits from one or more classes is called a **subclass**

Implementing Inheritance

The keyword **extends** is used to set up an inheritance relationship between a subclass and a superclass.

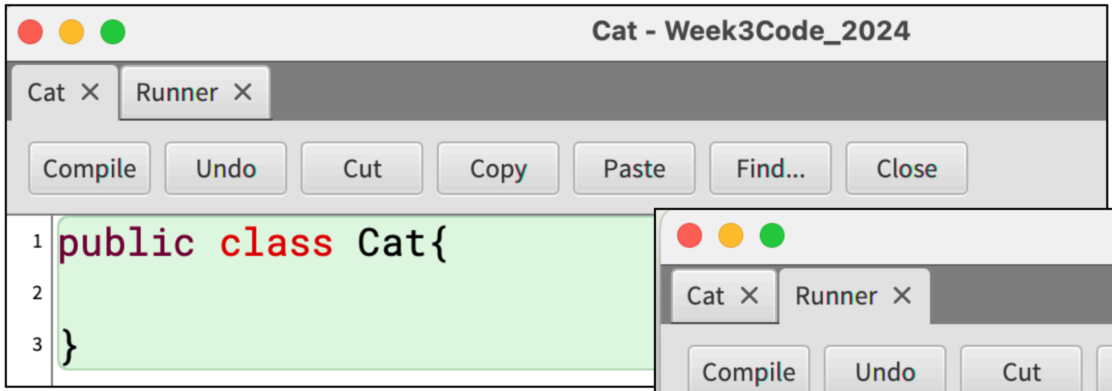


```
public class A {
     //state
     //behaviour
}
```

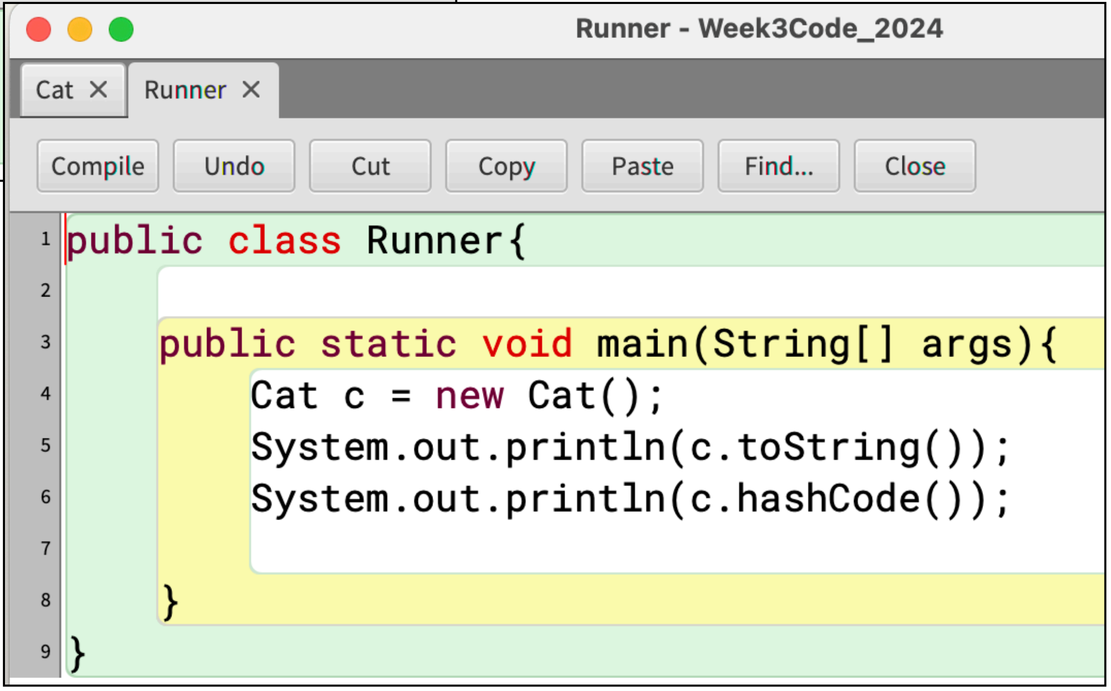
```
public class B extends A{
    //state
    //behaviour
}
```

The open, unshaded arrow is used to represent generalisation/specialisation relationships in UML

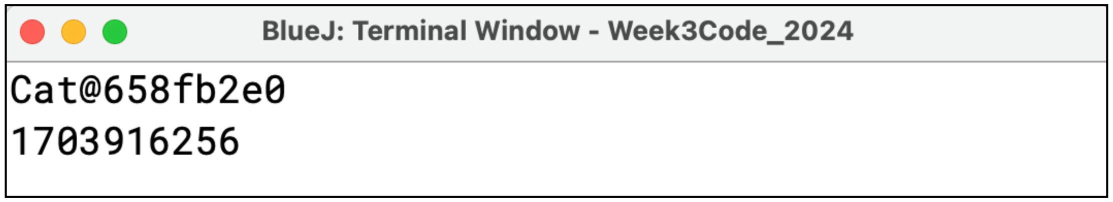
Example -Inheritance



```java
public class Cat{

}
```



```java
public class Runner{

    public static void main(String[] args){
        Cat c = new Cat();
        System.out.println(c.toString());
        System.out.println(c.hashCode());

    }
}
```



```
Cat@658fb2e0
1703916256
```

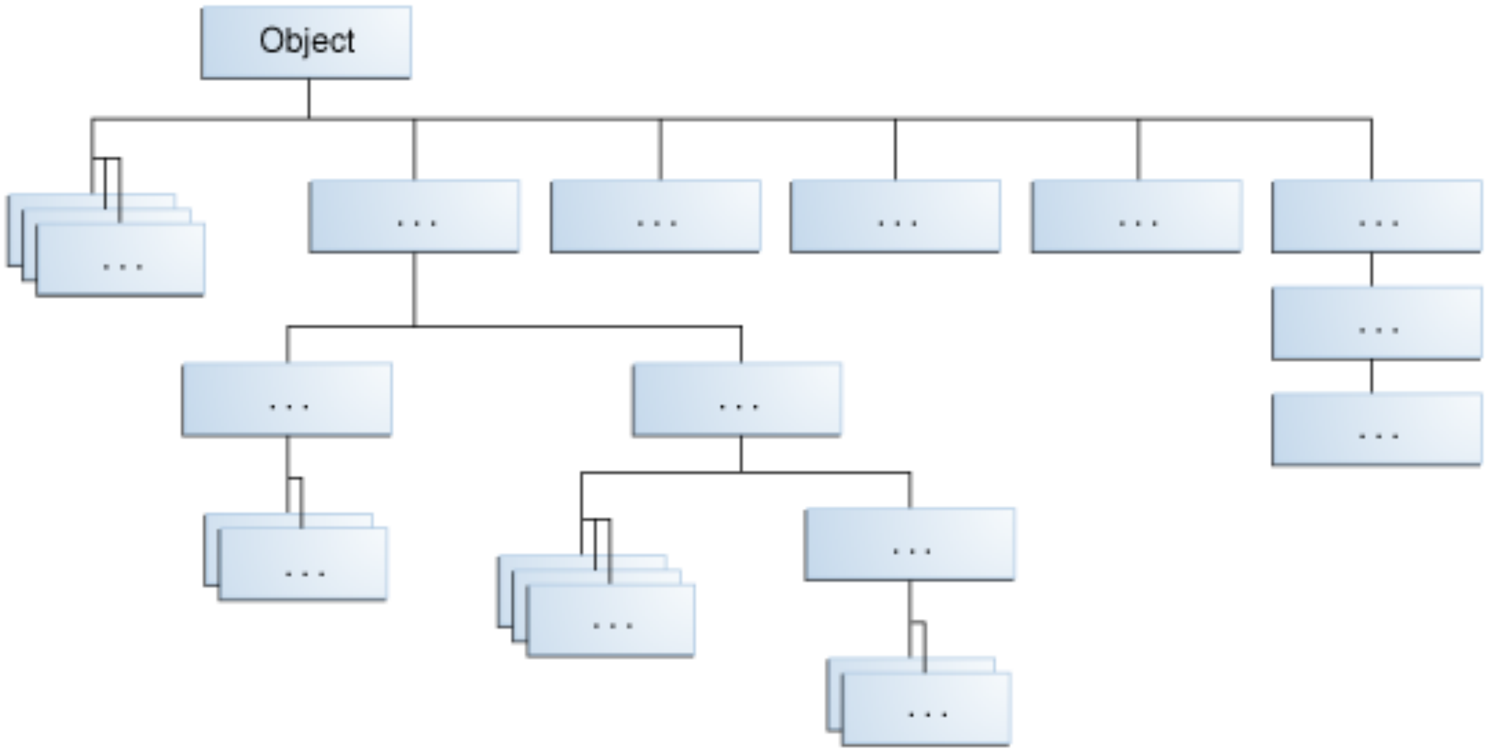
The Cat.java class is essentially empty but we can call methods that work!

Where do these methods come from? The Object class.

All Java classes are subclasses of the Object class. As a result of this, they have access to several inherited methods.

Object Class

The `Object` class, defined in the `java.lang` package, defines and implements behavior common to all classes—including the ones that you write. In the Java platform, many classes derive directly from `Object`, other classes derive from some of those classes, and so on, forming a hierarchy of classes.

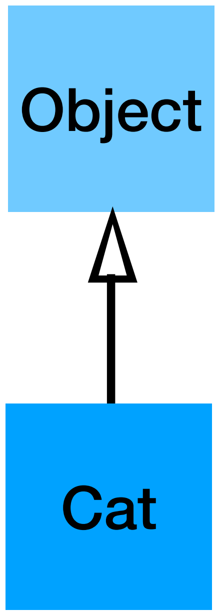


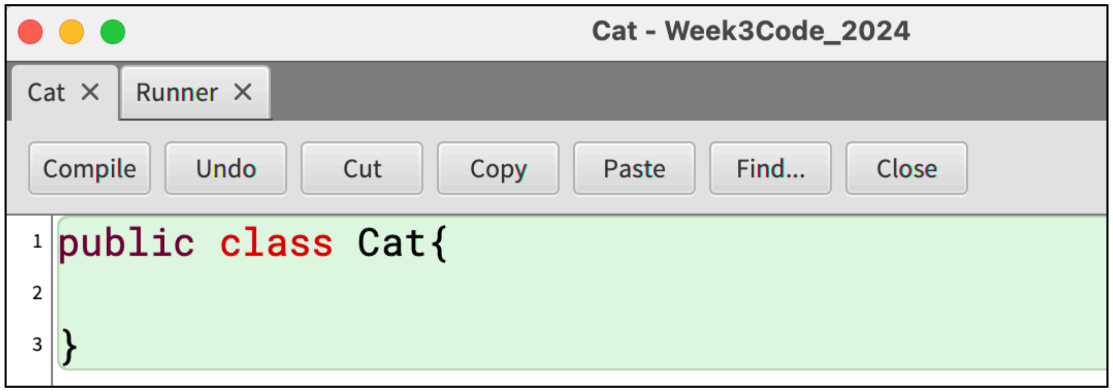
Object Class

Some of the methods inherited from Object are:

- `public boolean equals(Object obj)`
  Indicates whether some other object is "equal to" this one.
- `public int hashCode()`
  Returns a hash code value for the object.
- `public String toString()`
  Returns a string representation of the object.

Object Superclass



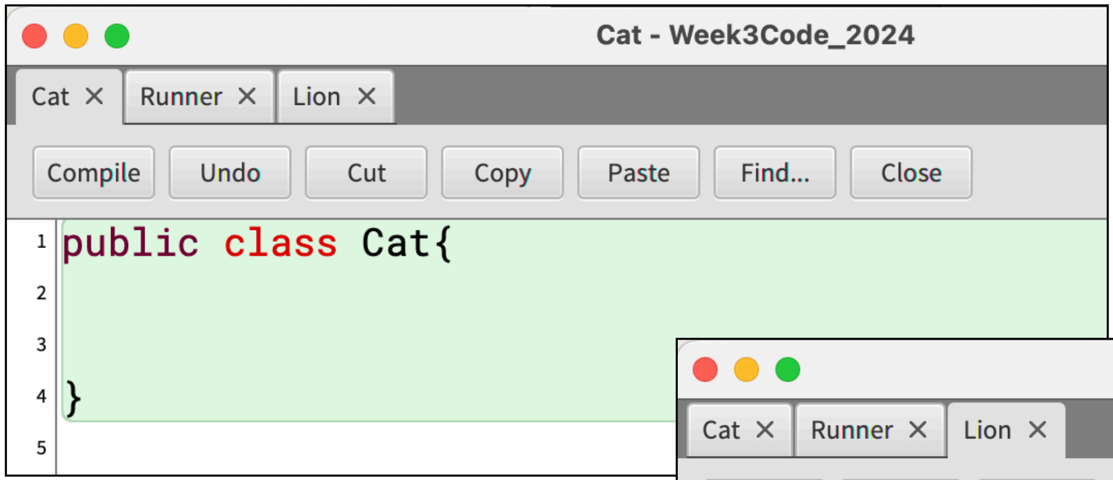


```
public class Cat{

}
```

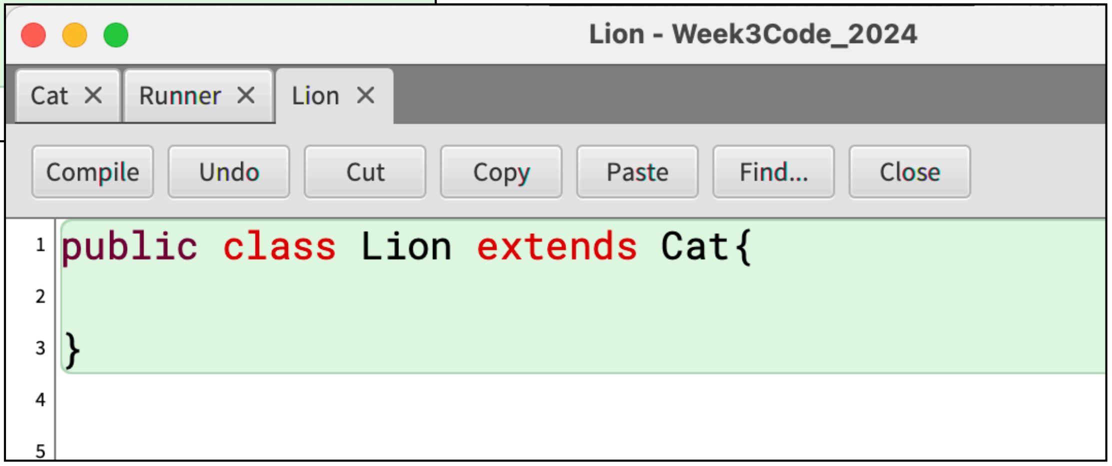
These methods can be called even though the Cat class is not explicitly using ‘extends Object’ in its class signature.

This statement is applied to the class by default. However, when setting up a custom inheritance hierarchy, we must include ‘extends’.

Inheritance Chains



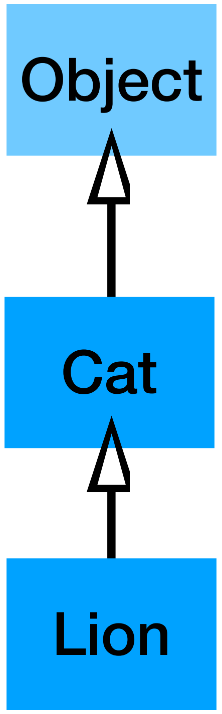
```java
public class Cat{


}
```

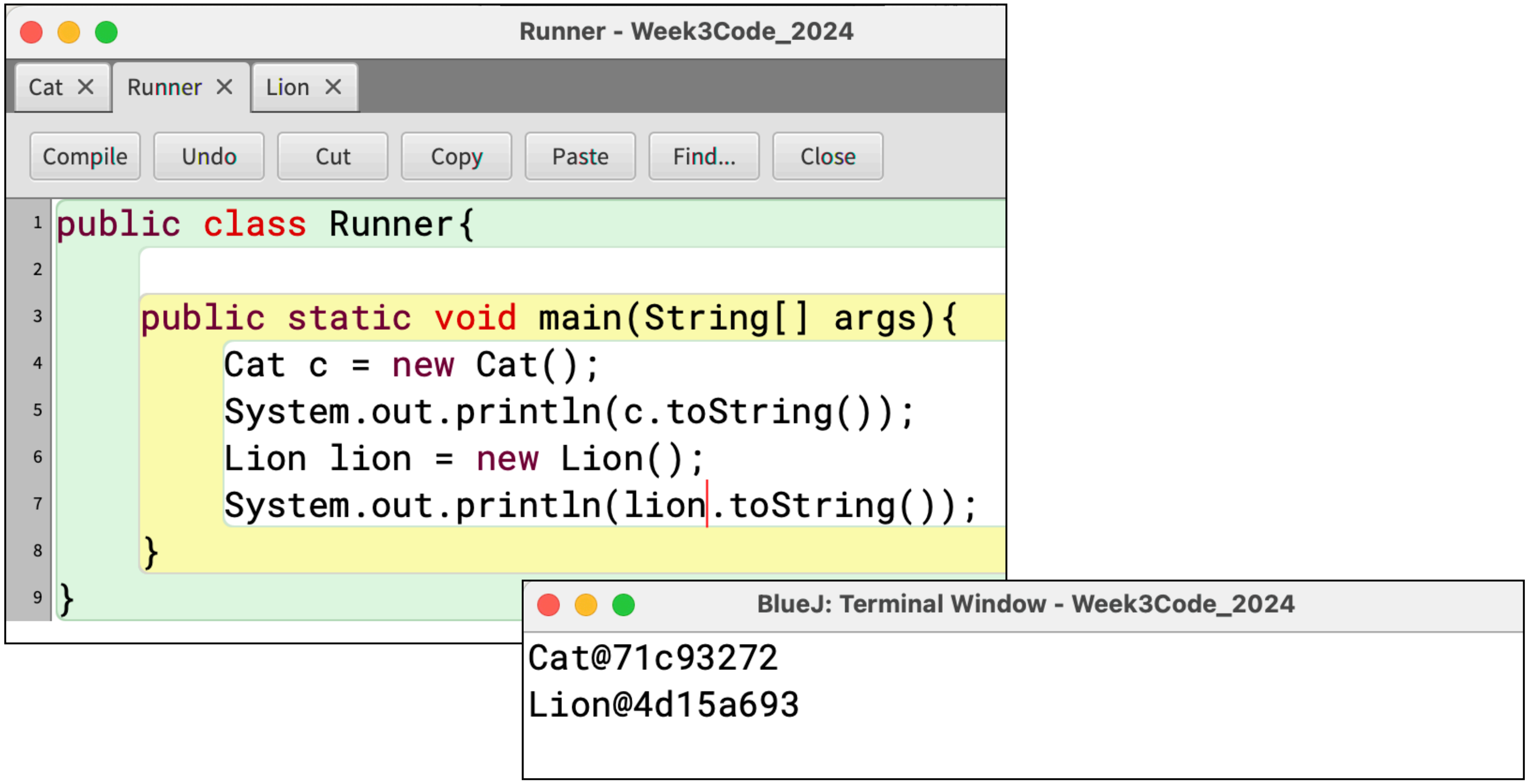


```java
public class Lion extends Cat{

}
```

Suppose we create a Lion class that is a subclass of the Cat class. Notice that the Lion class is just as empty as the Cat class.

Inheritance Chains



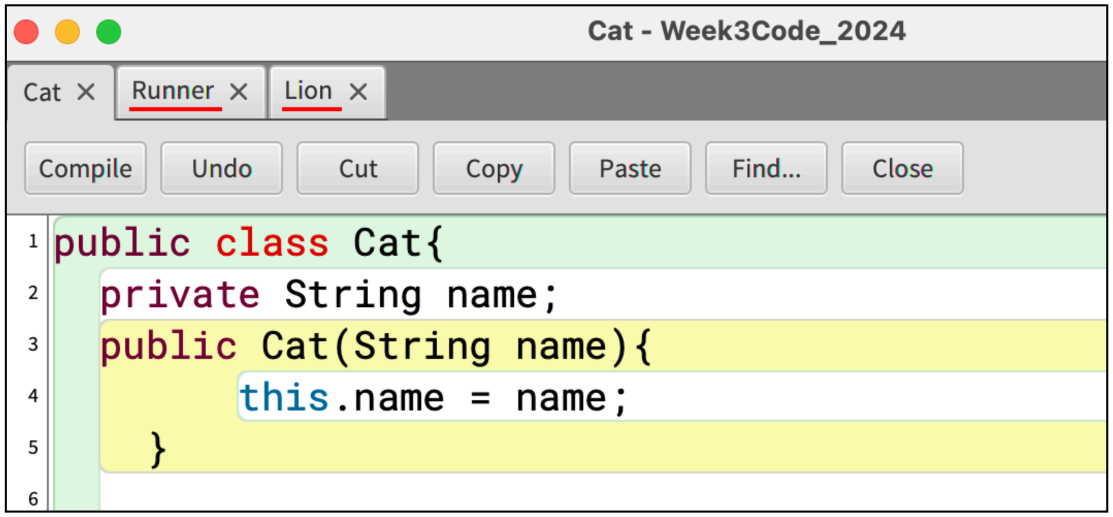


```
public class Runner{

    public static void main(String[] args){
        Cat c = new Cat();
        System.out.println(c.toString());
        Lion lion = new Lion();
        System.out.println(lion.toString());
    }
}
```

```
Cat@71c93272
Lion@4d15a693
```

The Lion class has the same benefits as the Cat class.
We can use the methods from Object (superclass).

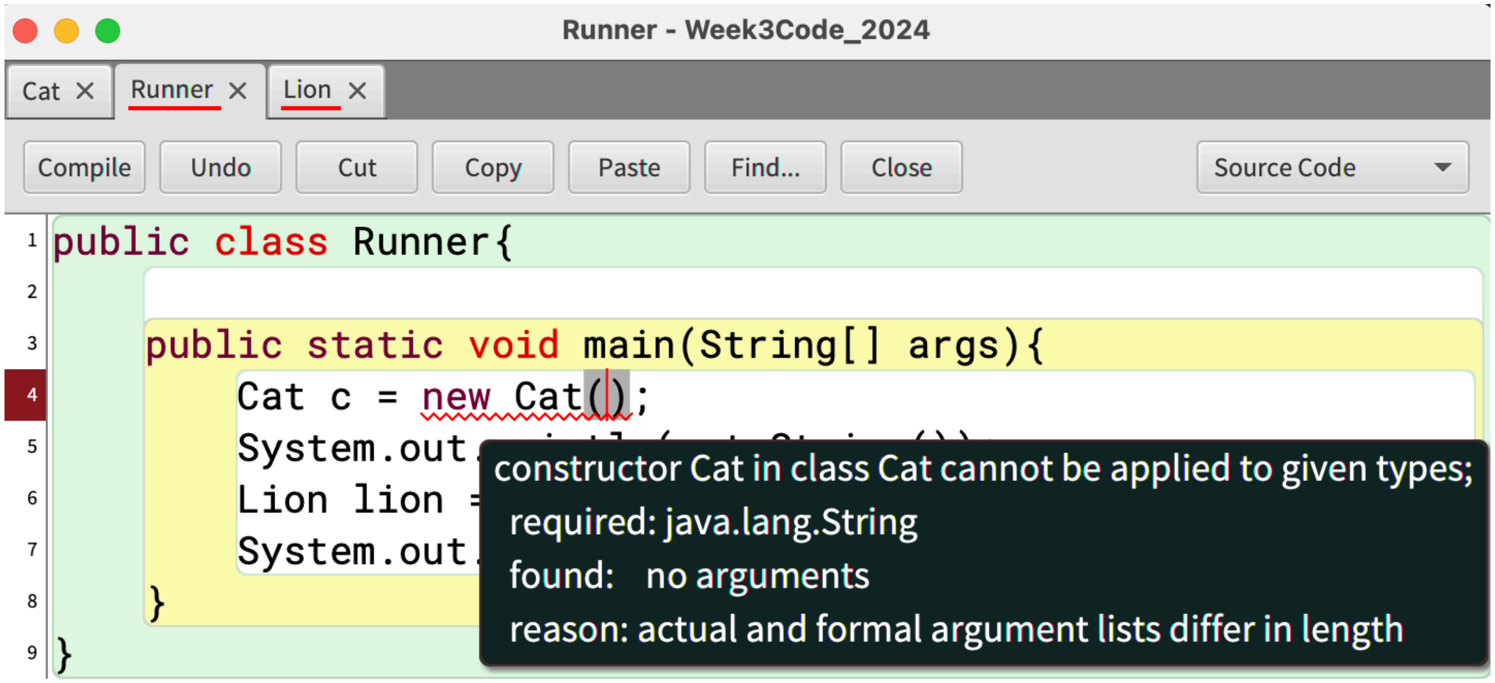
Inheritance and Constructors



```
public class Cat{
    private String name;
    public Cat(String name){
            this.name = name;
      }
```

Suppose we modify the Cat class with a new constructor that accepts a name. This constructor replaces the default no-argument constructor that the Runner class uses. So must fix that class.

Inheritance and Constructors



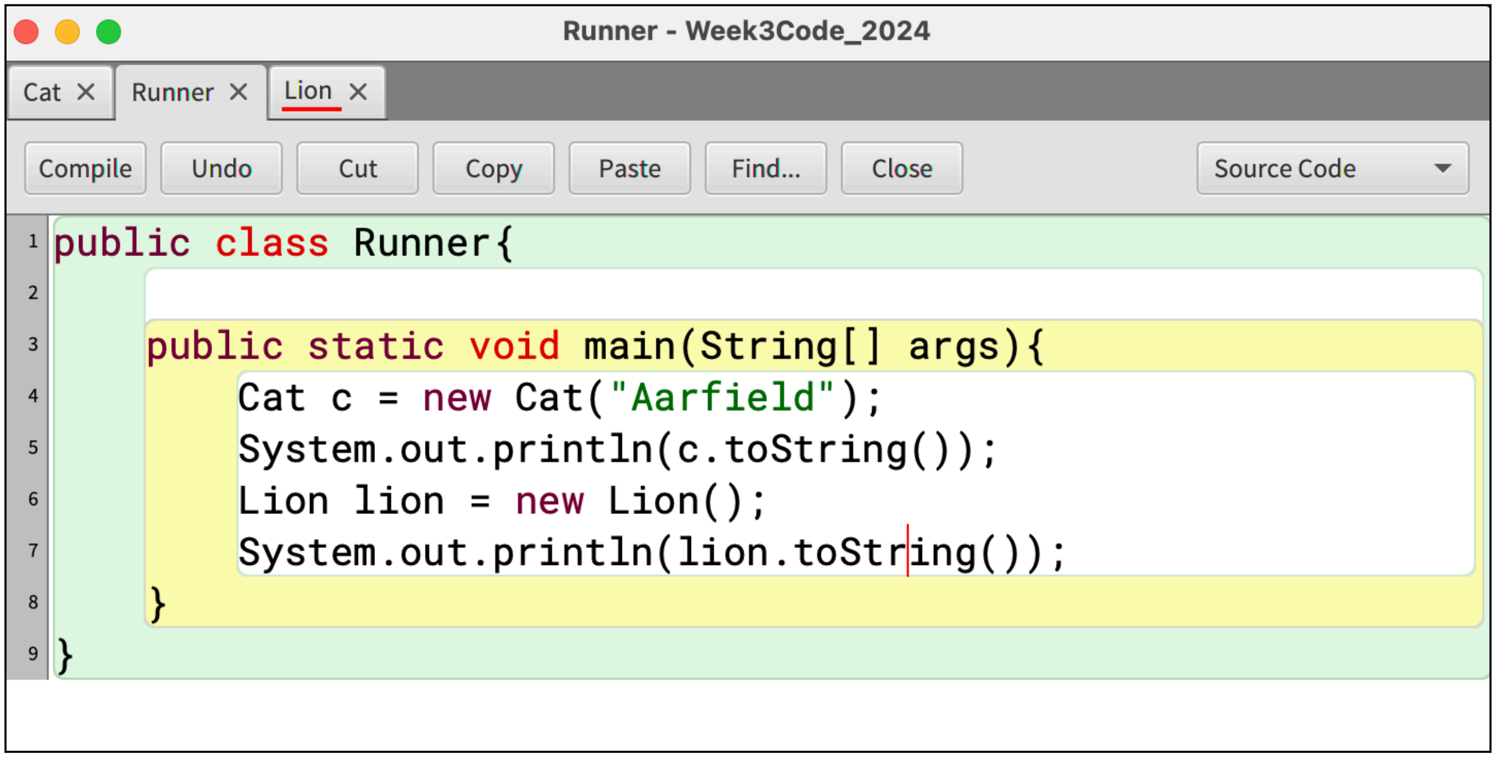
```
public class Runner{

    public static void main(String[] args){
        Cat c = new Cat();
        System.out.
        Lion lion =
        System.out.
    }
}
```

```
constructor Cat in class Cat cannot be applied to given types;
  required: java.lang.String
  found:    no arguments
  reason: actual and formal argument lists differ in length
```

The Cat class can be instantiated now ONLY via the constructor that requires a String. Therefore, any class that uses the Cat constructor must supply a String.

We can no longer use new Cat( ).

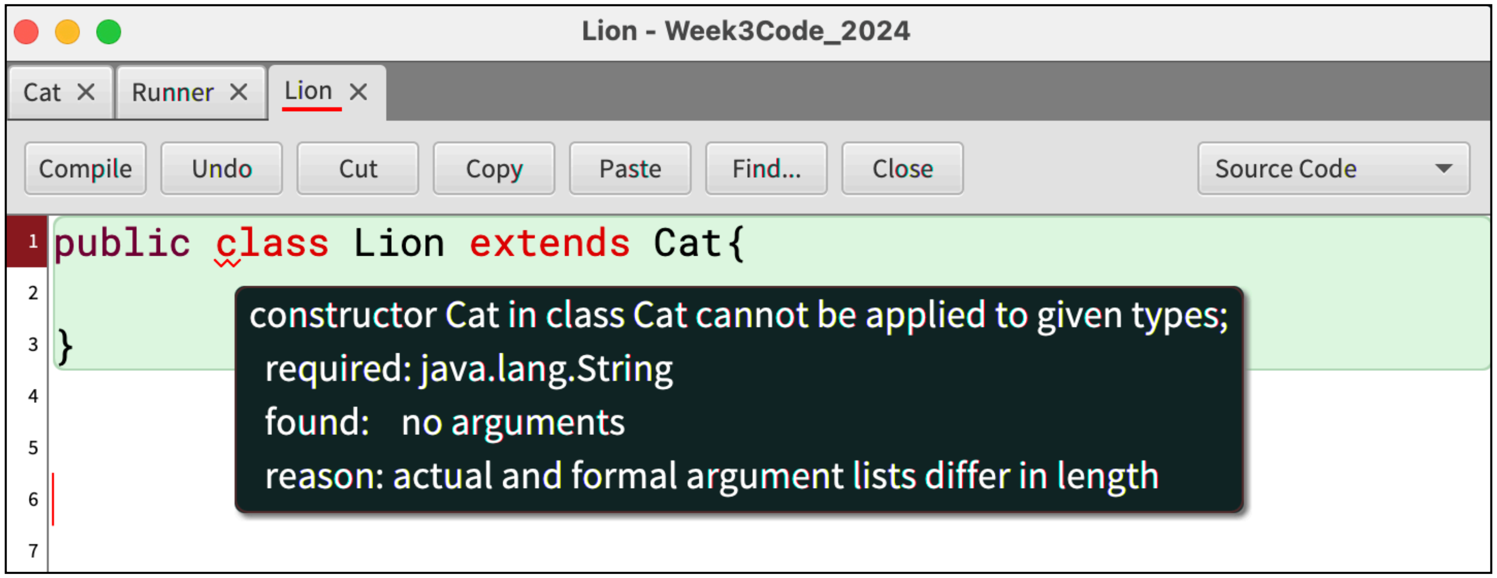
Inheritance and Constructors



```java
public class Runner{

    public static void main(String[] args){
        Cat c = new Cat("Aarfield");
        System.out.println(c.toString());
        Lion lion = new Lion();
        System.out.println(lion.toString());
    }
}
```

The Runner class is fixed but the Lion class still has an error.

Inheritance and Constructors



```
public class Lion extends Cat{

}
```

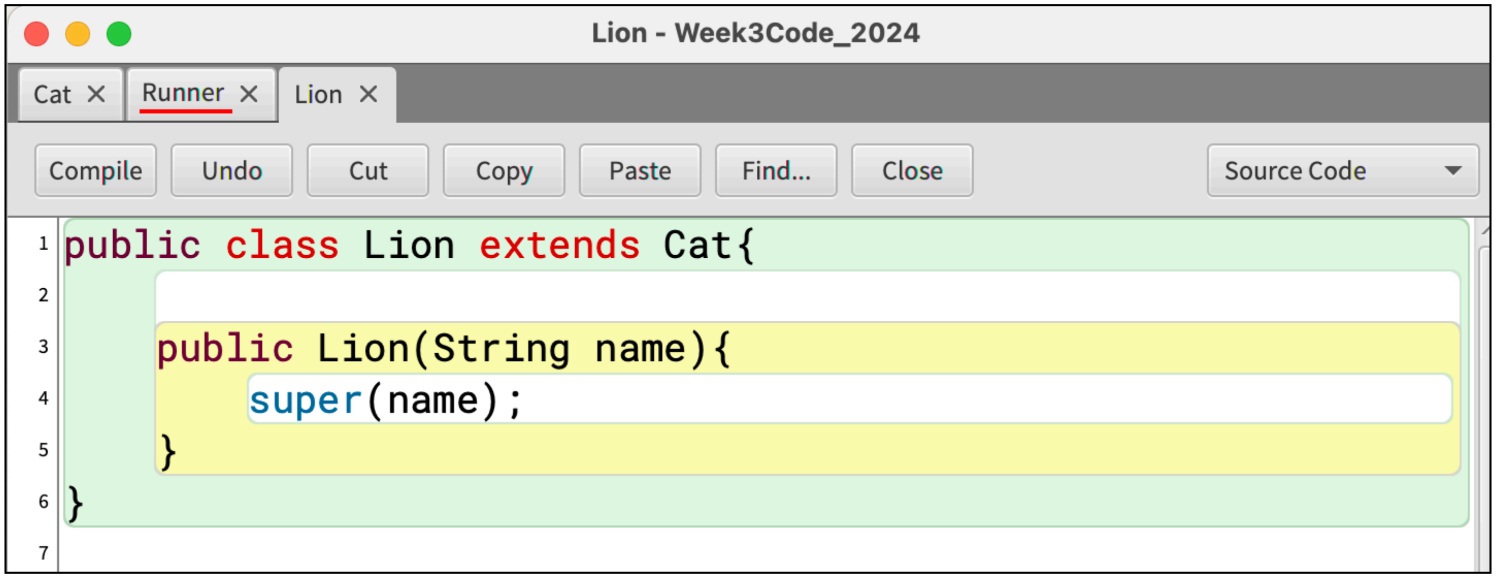
```
constructor Cat in class Cat cannot be applied to given types;
 required: java.lang.String
 found:    no arguments
 reason: actual and formal argument lists differ in length
```

All subclasses invoke their direct superclass' constructor when they are instantiated. Therefore, whenever new Lion( ) is called, that triggers a call to new Cat( ) because Lion is a subclass of Cat.

Remember, the Cat class can be instantiated now ONLY via the constructor that requires a String. Therefore, any class that uses the Cat constructor must supply a String.

We can no longer use new Cat( ) in Lion hence the error.

Inheritance and Constructors

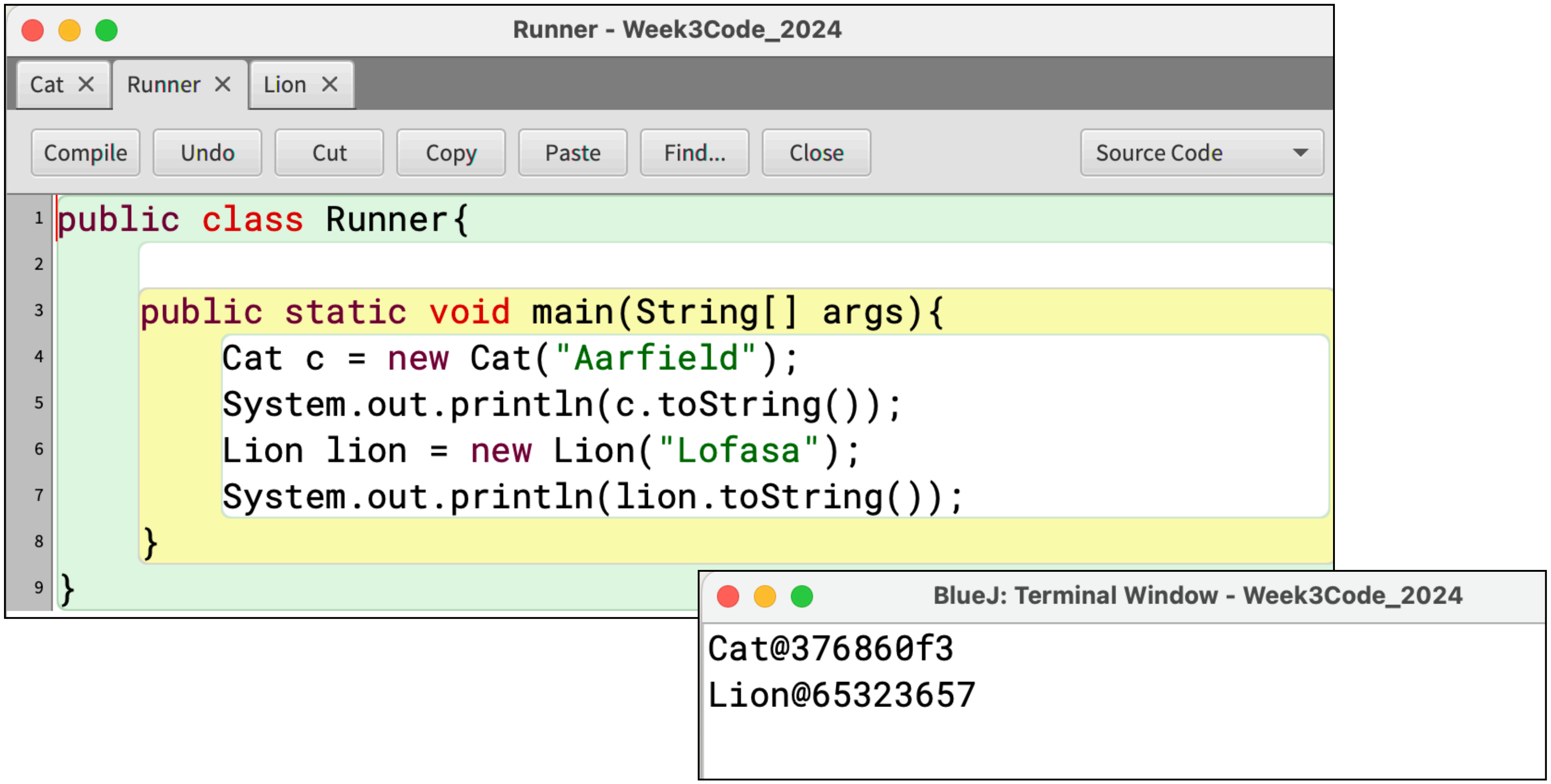


```java
public class Lion extends Cat{

    public Lion(String name){
        super(name);
    }
}
```

The Lion class needs to have a constructor that passes in a String parameter to the Cat superclass. The keyword **super** is used to call the constructor of a direct superclass.

This fixes the Lion class, but the Runner class has to be updated to use the new Lion constructor.

Inheritance and Constructors



```java
public class Runner{

    public static void main(String[] args){
        Cat c = new Cat("Aarfield");
        System.out.println(c.toString());
        Lion lion = new Lion("Lofasa");
        System.out.println(lion.toString());
    }
}
```

```
Cat@376860f3
Lion@65323657
```

The Runner class uses the correct constructors.

Why is the output like this? (Look at Slide 11).

Lion uses the custom toString( ) from its direct superclass, Cat, which uses the one from Object.

Aggregation

An aggregation relationship models a whole-part relationship between two classes. It is sometimes referred to as a part-of relationship.

Consider two classes, A and B. An instance of A (referred to as the “whole”), consists of instances of B (referred to as the “parts”).

Reading: Page 61, Chapter 3 - G. Booch; Page 134,

Aggregation

The multiplicity of the relationship determines how aggregation relationships are implemented.

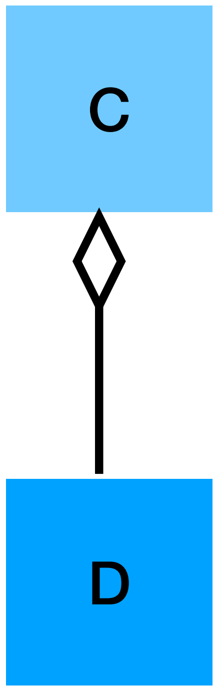
If an instance of class A contains one instance of class B, the relationship is implemented by storing the reference to an instance of B in A.

If an instance of class A contains more than one instance of B, some sort of data structure or collection will need to be used to store the object references of type B.

Reading: Page 61, Chapter 3 - G. Booch; Page 134,

Implementing Aggregation

Collections are often used to implement aggregation. However, a single reference can also be used.



```
public class C {
     private Collection<D> parts;
     //behaviour
}
```

```
public class D{
    //state
    //behaviour
}
```

The open, unshaded diamond is used to represent aggregation relationships in UML. The diamond points to the aggregate class (the one that contains the other class)

Aggregation

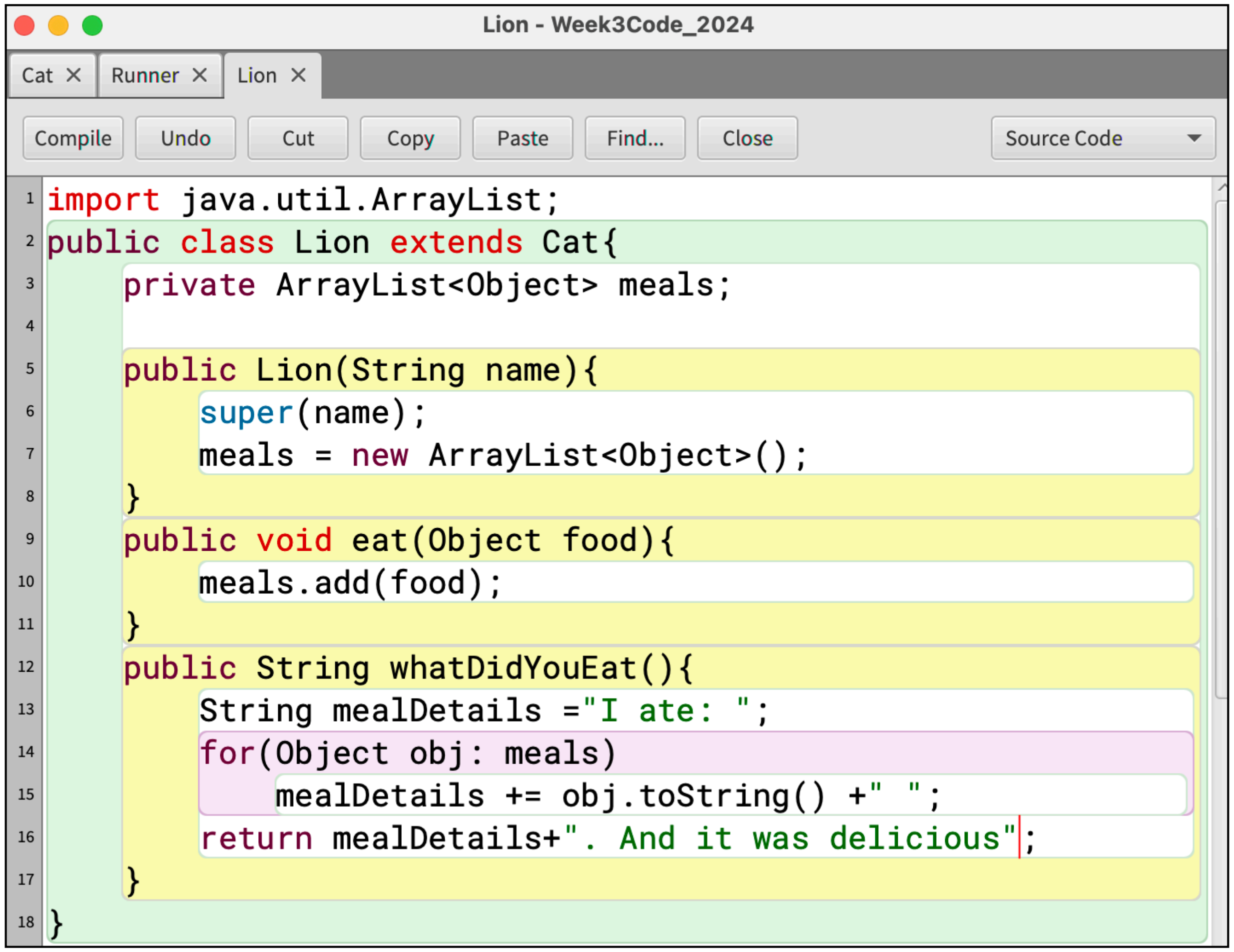
Aggregation raises the issue of ownership.

For example for the abstraction of a garden permits different plants to be raised in a garden over time, but replacing a plant does not change the identity of the garden as a whole, nor does removing a garden necessarily destroy all of its plants (they are likely just transplanted).

In other words, the lifetime of a garden and its plants are independent.

Reference: Page 62, Chapter 3 - G. Booch

Aggregation Implementation

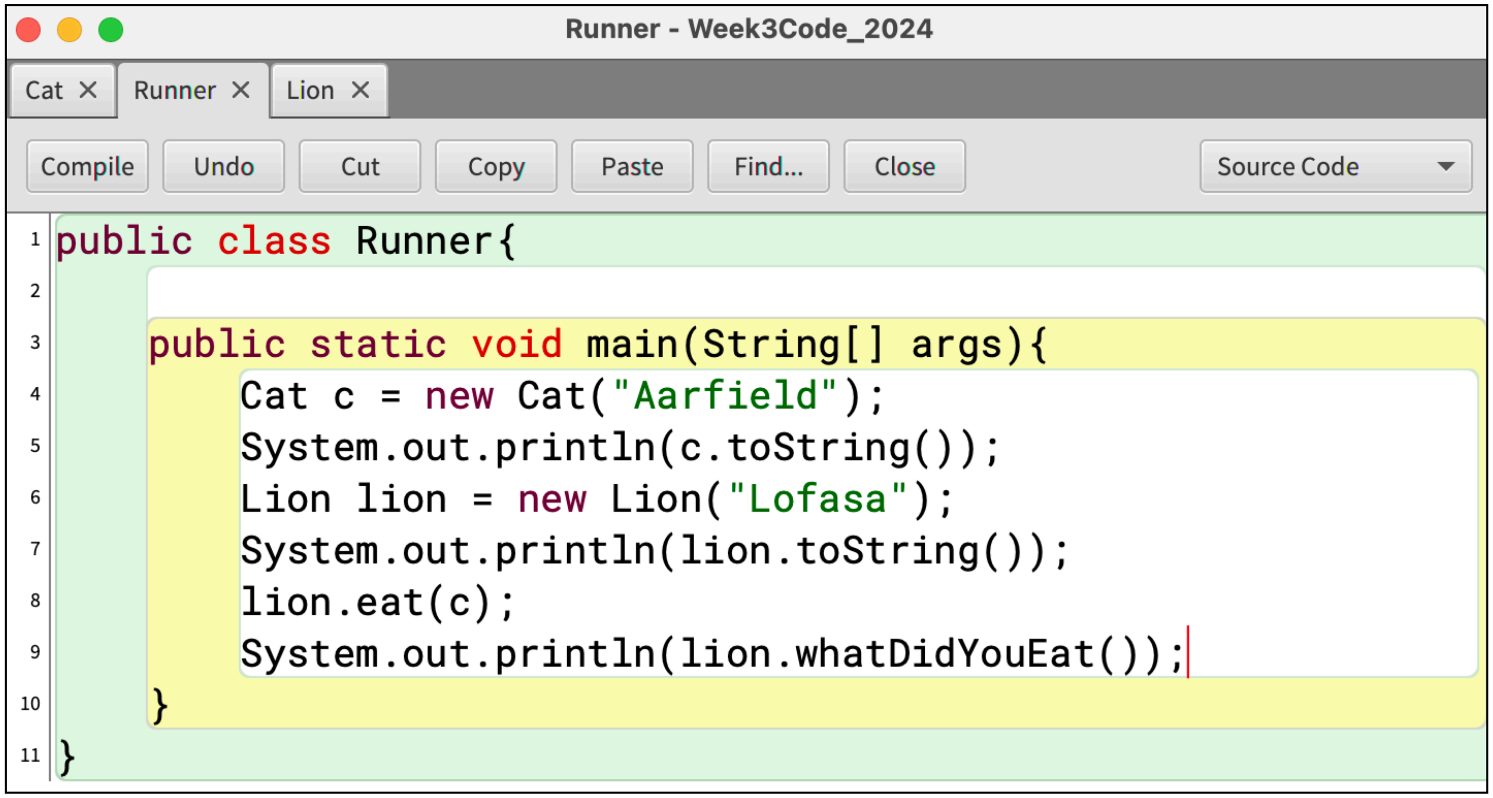


```java
import java.util.ArrayList;
public class Lion extends Cat{
	private ArrayList<Object> meals;

	public Lion(String name){
		super(name);
		meals = new ArrayList<Object>();
	}
	public void eat(Object food){
		meals.add(food);
	}
	public String whatDidYouEat(){
		String mealDetails ="I ate: ";
		for(Object obj: meals)
			mealDetails += obj.toString() +" ";
		return mealDetails+". And it was delicious";
	}
}
```

Suppose we model what a Lion eats* as an aggregation relationship using an ArrayList called meals. Two methods are added: eats(..) passes in an object to add to the meals collection. The whatDidYouEat( ) method returns a list of what the Lion ate.

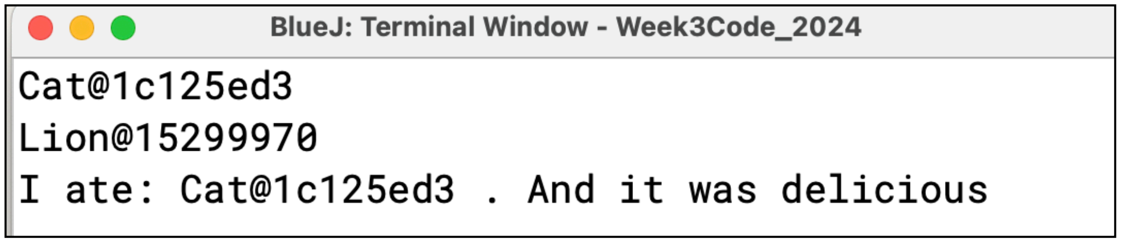
*scenario inspired by a discussion after the lecture on Feb 06,2024

Aggregation Implementation



```java
1  public class Runner{
2  
3      public static void main(String[] args){
4          Cat c = new Cat("Aarfield");
5          System.out.println(c.toString());
6          Lion lion = new Lion("Lofasa");
7          System.out.println(lion.toString());
8          lion.eat(c);
9          System.out.println(lion.whatDidYouEat());
10     }
11 }
```

Then, in Runner we invoke the eat( ) method on line 8.



```
Cat@1c125ed3
Lion@15299970
I ate: Cat@1c125ed3 . And it was delicious
```

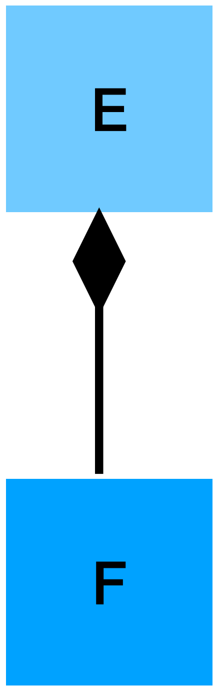
Composition

Composition is a form of aggregation with stronger ownership between the “whole” and its “parts”.

The “parts” in the relationship live and die with the “whole”. In other words, the “whole” is responsible for the creation and destruction of its “parts”.

Composition

Collections are often used to implement composition. However, a single reference can also be used.

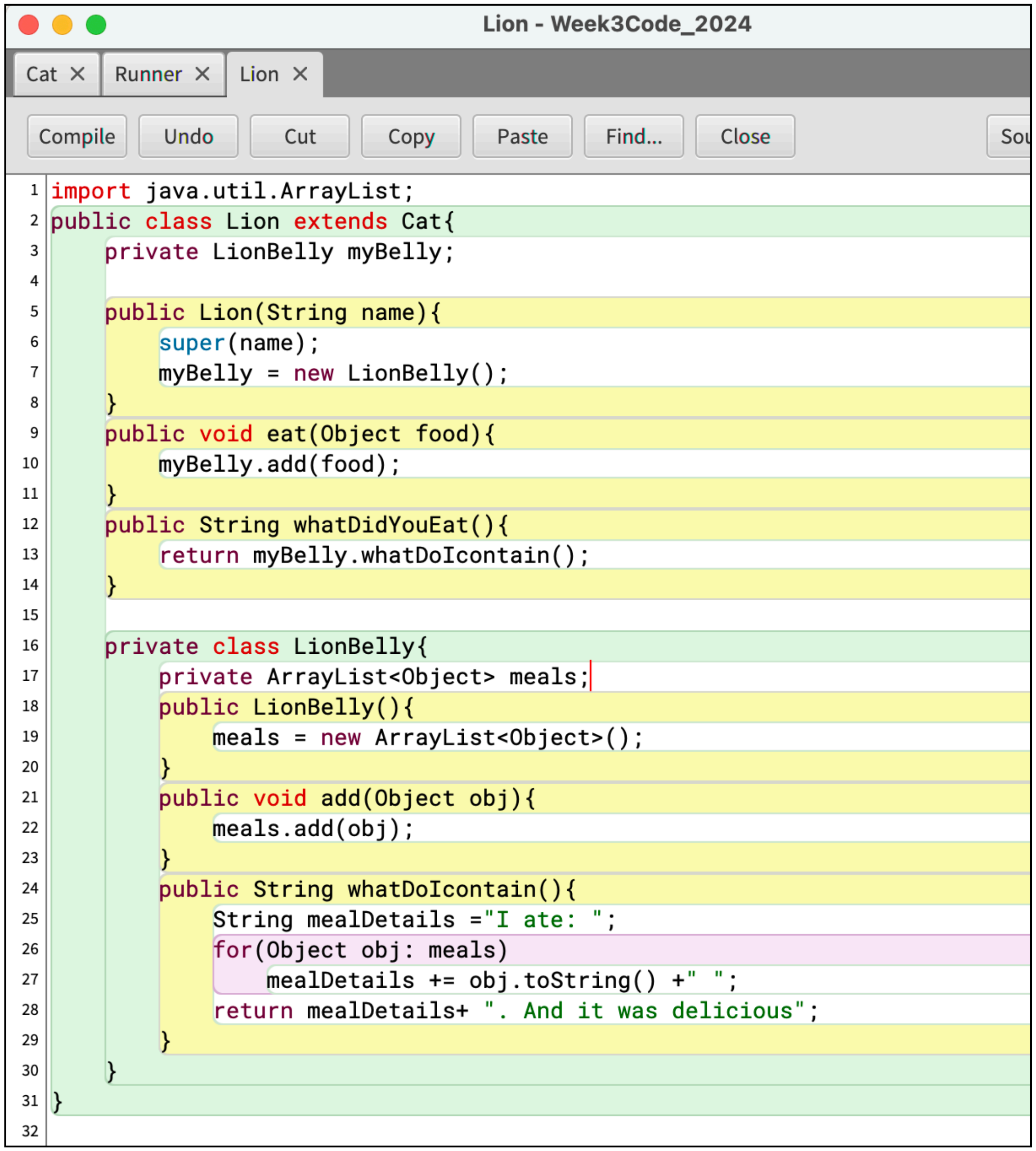


```
public class E {
     private F part;
     //behaviour
}
```

```
public class F{
    //state
    //behaviour
}
```

The open, shaded diamond is used to represent composition relationships in UML. The diamond points to the composite class (the one that contains the other class)

Example: Implementing Composition



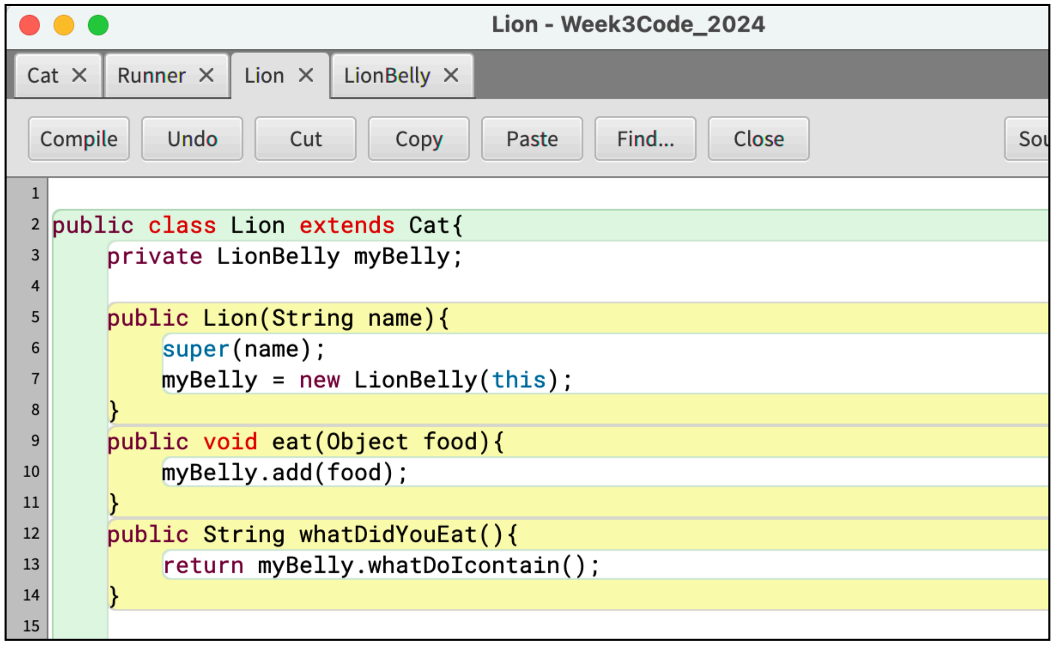
```java
import java.util.ArrayList;
public class Lion extends Cat{
    private LionBelly myBelly;

    public Lion(String name){
        super(name);
        myBelly = new LionBelly();
    }
    public void eat(Object food){
        myBelly.add(food);
    }
    public String whatDidYouEat(){
        return myBelly.whatDoIcontain();
    }

    private class LionBelly{
        private ArrayList<Object> meals;
        public LionBelly(){
            meals = new ArrayList<Object>();
        }
        public void add(Object obj){
            meals.add(obj);
        }
        public String whatDoIcontain(){
            String mealDetails ="I ate: ";
            for(Object obj: meals)
                mealDetails += obj.toString() +" ";
            return mealDetails+ ". And it was delicious";
        }
    }
}
```

This example refactors the Lion class to illustrate composition.

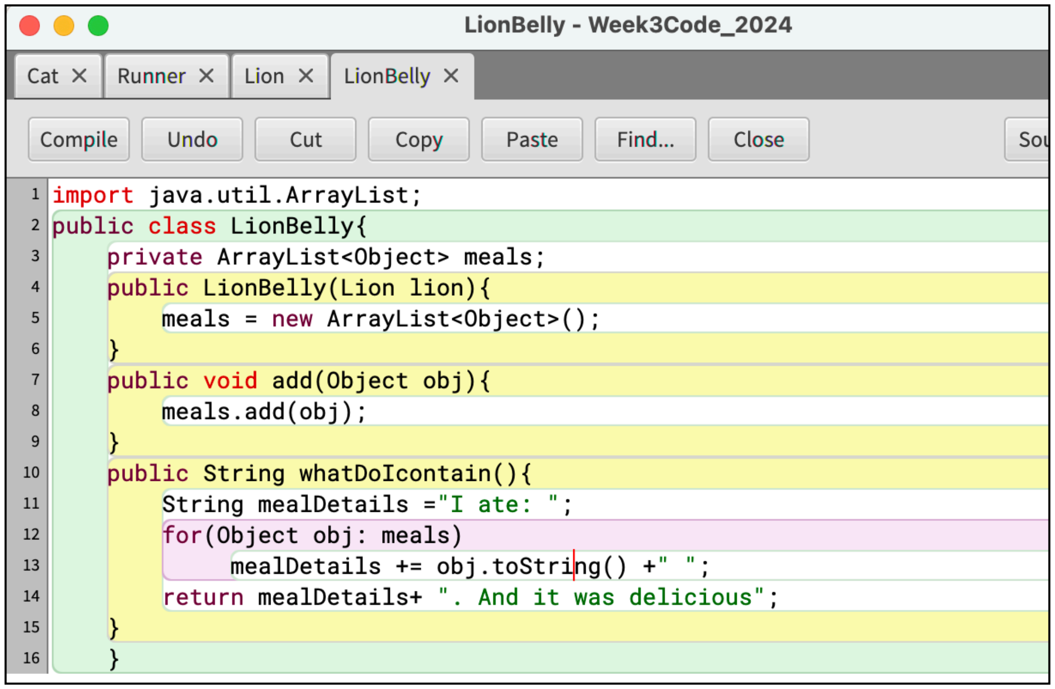
The ArrayList is moved into a new inner class (private), LionBelly, which only exists in the context of a Lion. Here, only the Lion class can create a new LionBelly object.

Once the Lion object disappears, the LionBelly object disappears as well. Therefore, we cannot have a LionBelly object existing without a Lion object existing first.

Example: Implementing Composition



```
1
2  public class Lion extends Cat{
3      private LionBelly myBelly;
4
5      public Lion(String name){
6          super(name);
7          myBelly = new LionBelly(this);
8      }
9      public void eat(Object food){
10         myBelly.add(food);
11     }
12     public String whatDidYouEat(){
13         return myBelly.whatDoIcontain();
14     }
15
```



```
1  import java.util.ArrayList;
2  public class LionBelly{
3      private ArrayList<Object> meals;
4      public LionBelly(Lion lion){
5          meals = new ArrayList<Object>();
6      }
7      public void add(Object obj){
8          meals.add(obj);
9      }
10     public String whatDoIcontain(){
11         String mealDetails ="I ate: ";
12         for(Object obj: meals)
13             mealDetails += obj.toString() +" ";
14         return mealDetails+ ". And it was delicious";
15     }
16     }
```

This is another way of implementing the concept where the LionBelly class can only be created (instantiated) if a Lion object is passed into its constructor(line 4 - LionBelly class).

There is less control of the LionBelly class from the Lion class but the idea is the same: a Lion is composed of a LionBelly.

Observe how the Lion class passes a reference of itself to the LionBelly constructor on line 7 of the Lion class.

Summary

Today you learned about:

- Inheritance
    - Using extends in a class signature
    - Constructors and the use of super
    - Effects of inheritance on object behaviour
    - UML diagram symbol
- Aggregration
    - Implementation using collections
    - UML diagram symbol
- Composition
    - Distinction from aggregation
    - UML diagram symbol

References

- Booch, G. (2007) Object-Oriented Analysis and Design. Chapter 2 - the Object Model
- Chapter 2 Objects: Using, Creating, and Defining: https://runestone.academy/ns/books/published/javajavajava/chapter-objects.html
- Chapter 3 Methods: Communicating With Objects: https://runestone.academy/ns/books/published/javajavajava/chapter-methods.html