Ollscoil na Gaillimhe
University of Galway

CT2106

Object Oriented Programming

Dr. Frank Glavin
Room 404, IT Building
Frank.Glavin@University*of*Galway.ie

School of Computer Science

University
*of*Galway.ie

Our Food Chain

Seeds

Canaries

Cats

- Canaries eat Seed
- Cats eat Canaries
- Energy passes from Seeds to the Canary to the Cat

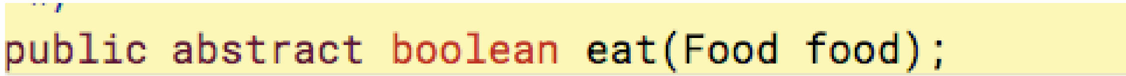
Implement Canary’s eat method

Canary’s *eat* method should do the following:

1. Check if the Food object is null
2. Checks if Food object is an *instanceof* Seed;
3. If it is a Seed, the canary calls the *extractEnergy* method and *adds* the value returned to its own energy level
4. It also calls the sing method (because it is now well fed)

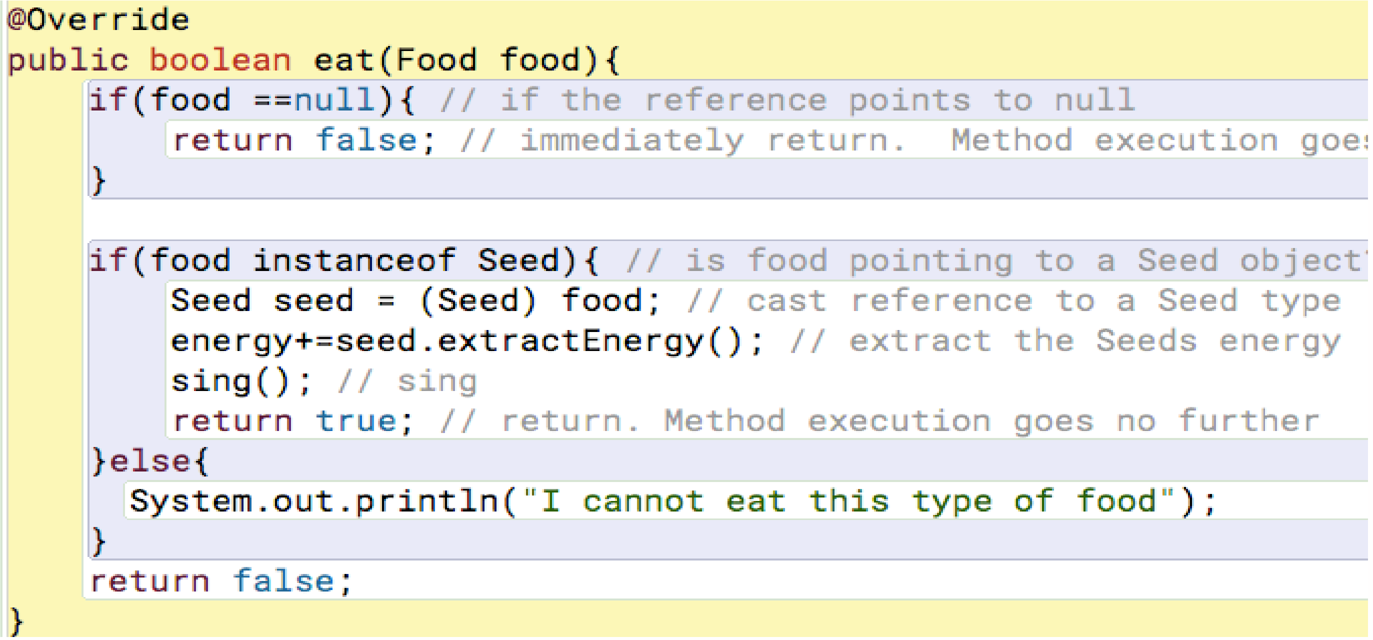
I would also suggest that this method is modified to return a boolean depending on whether the Food is edible (e.g it is a Seed or not)

Eat method



```
public abstract boolean eat(Food food);
```

“The eat method in Animal should be changed to return a boolean value. ”
“In Canary's case, the eat method should return *true* if the food variable is an instance of Seed. Otherwise, the method should return **false**.”



```
@Override
public boolean eat(Food food){
    if(food ==null){ // if the reference points to null
        return false; // immediately return.  Method execution goe
    }

    if(food instanceof Seed){ // is food pointing to a Seed object
        Seed seed = (Seed) food; // cast reference to a Seed type
        energy+=seed.extractEnergy(); // extract the Seeds energy
        sing(); // sing
        return true; // return. Method execution goes no further
    }else{
      System.out.println("I cannot eat this type of food");
    }
    return false;
}
```

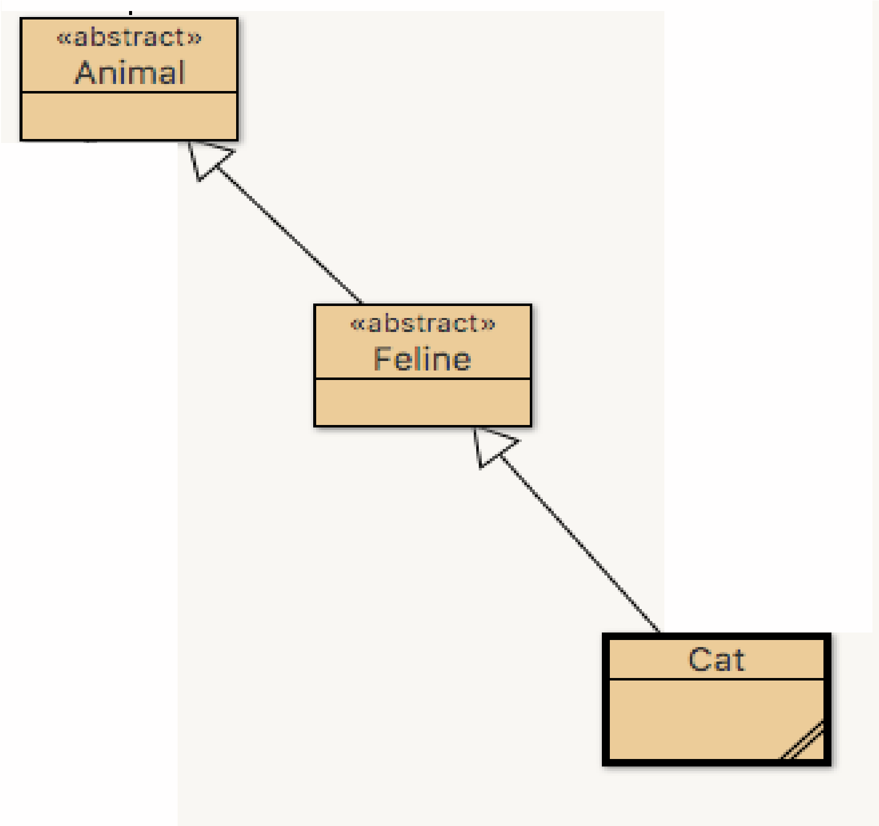
Adding Feline and Cat classes

Feline class (abstract)

- **Extends Animal**
- **Fields**
  - hasFur
- **Overrides**
  - move() method

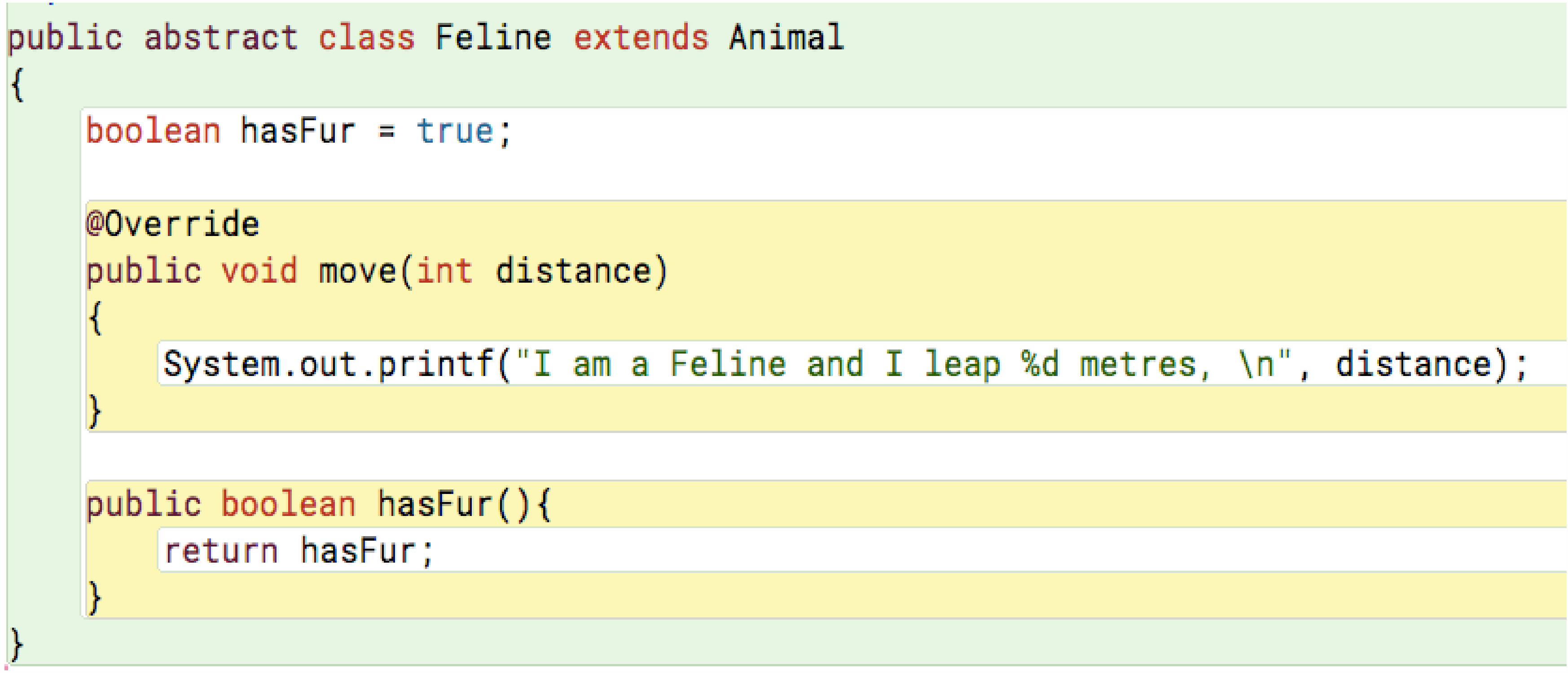
Cat class (concrete)

- **Extends Feline**
- **Fields**
  - name
- **Overrides**
  - colour field (colour=black)
  - eat (Food) method



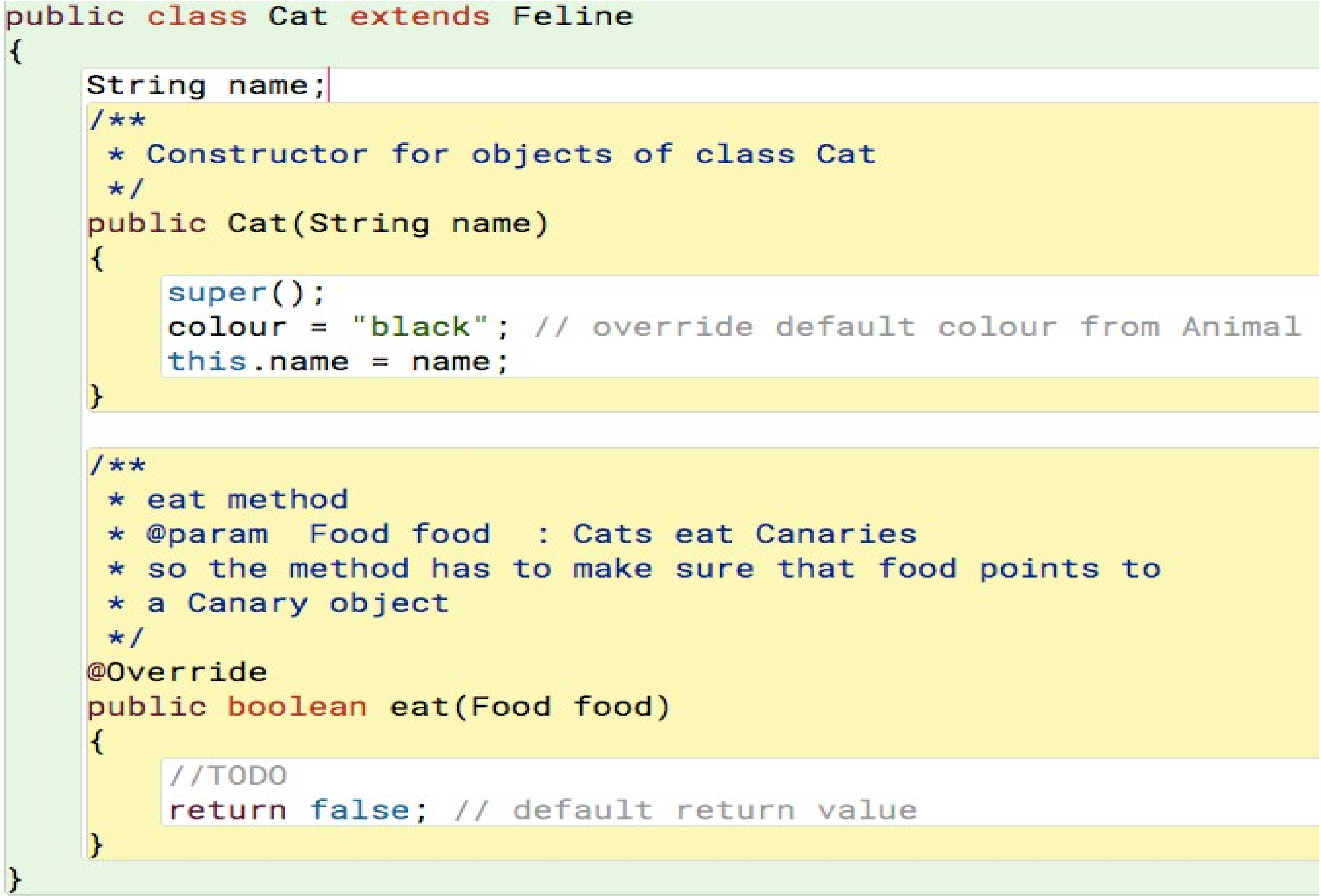
OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Feline class

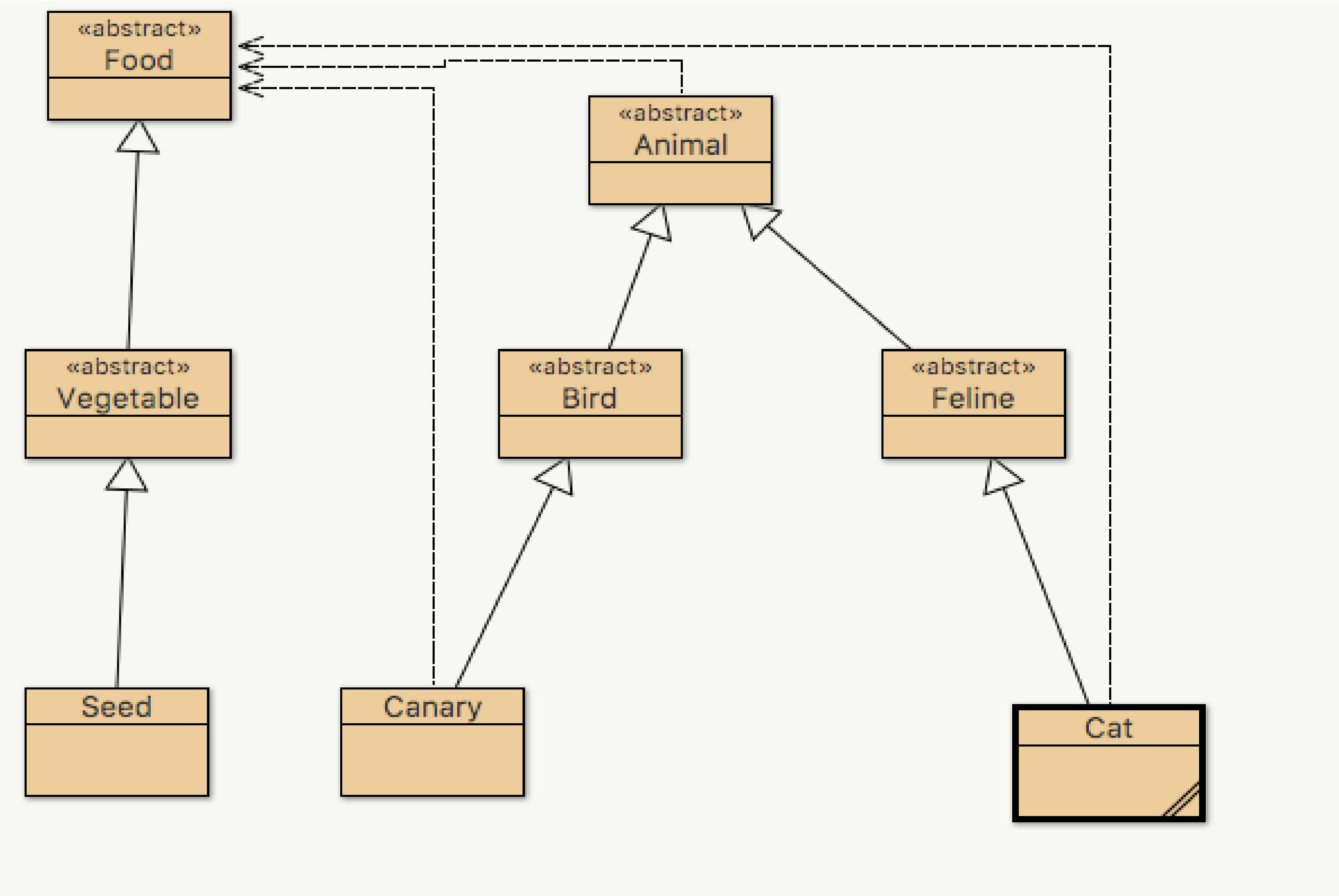


```java
public abstract class Feline extends Animal
{
    boolean hasFur = true;

    @Override
    public void move(int distance)
    {
        System.out.printf("I am a Feline and I leap %d metres, \n", distance);
    }

    public boolean hasFur(){
        return hasFur;
    }
}
```

Cat class

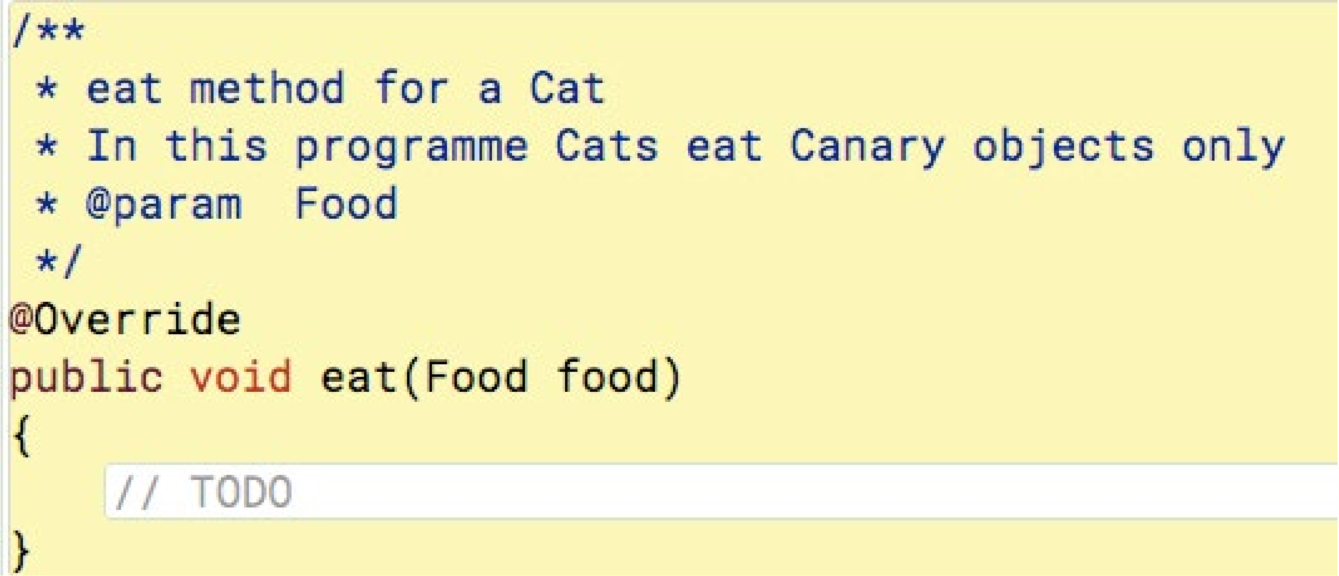


```
public class Cat extends Feline
{
    String name;
    /**
     * Constructor for objects of class Cat
     */
    public Cat(String name)
    {
        super();
        colour = "black"; // override default colour from Animal
        this.name = name;
    }

    /**
     * eat method
     * @param  Food food   : Cats eat Canaries
     * so the method has to make sure that food points to
     * a Canary object
     */
    @Override
    public boolean eat(Food food)
    {
        //TODO
        return false; // default return value
    }
}
```


«abstract»
Food
«abstract»
Animal
«abstract»
Vegetable
«abstract»
Bird
«abstract»
Feline
Seed
Canary
Cat

eat method of Cat

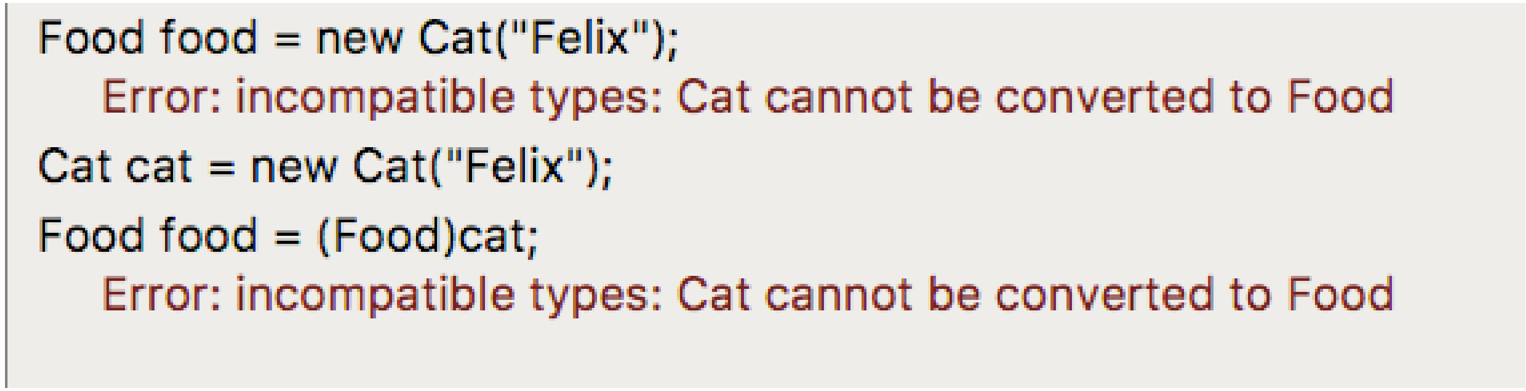
For this to work, a Canary **must** be a subclass of Food, just as Seed is
However, this is not the case.
Canary is a subclass of Animal



```
/**
 * eat method for a Cat
 * In this programme Cats eat Canary objects only
 * @param  Food
 */
@Override
public void eat(Food food)
{
    // TODO
}
```

A Canary is not a Food type

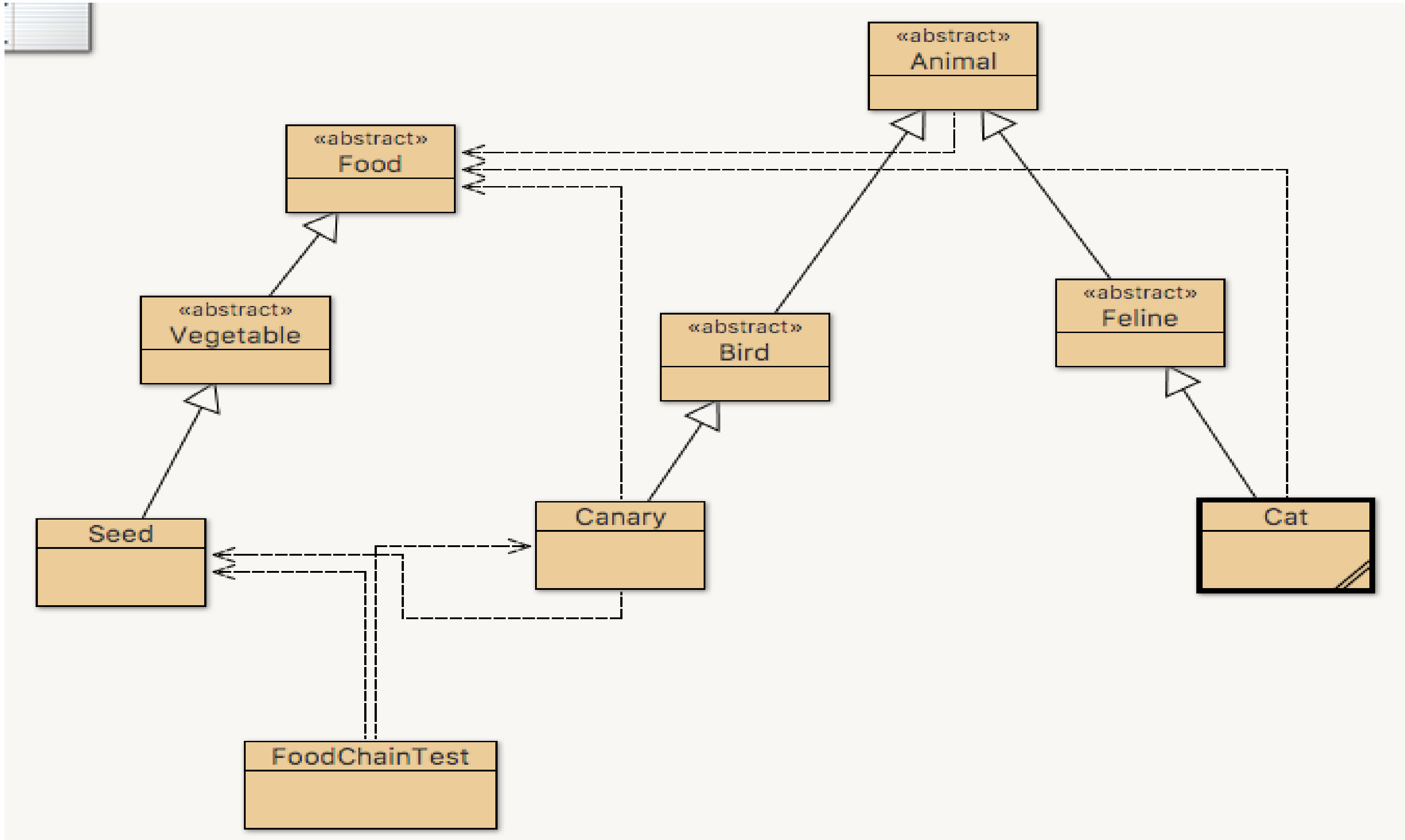
Furthermore, there is no way to cast a Canary object to Food
E.g. Try the following in code pad



```
Food food = new Cat("Felix");
    Error: incompatible types: Cat cannot be converted to Food
Cat cat = new Cat("Felix");
Food food = (Food)cat;
    Error: incompatible types: Cat cannot be converted to Food
```

For polymorphism to occur, Cat would have to be a subclass of Food

Arrange your classes to look like this

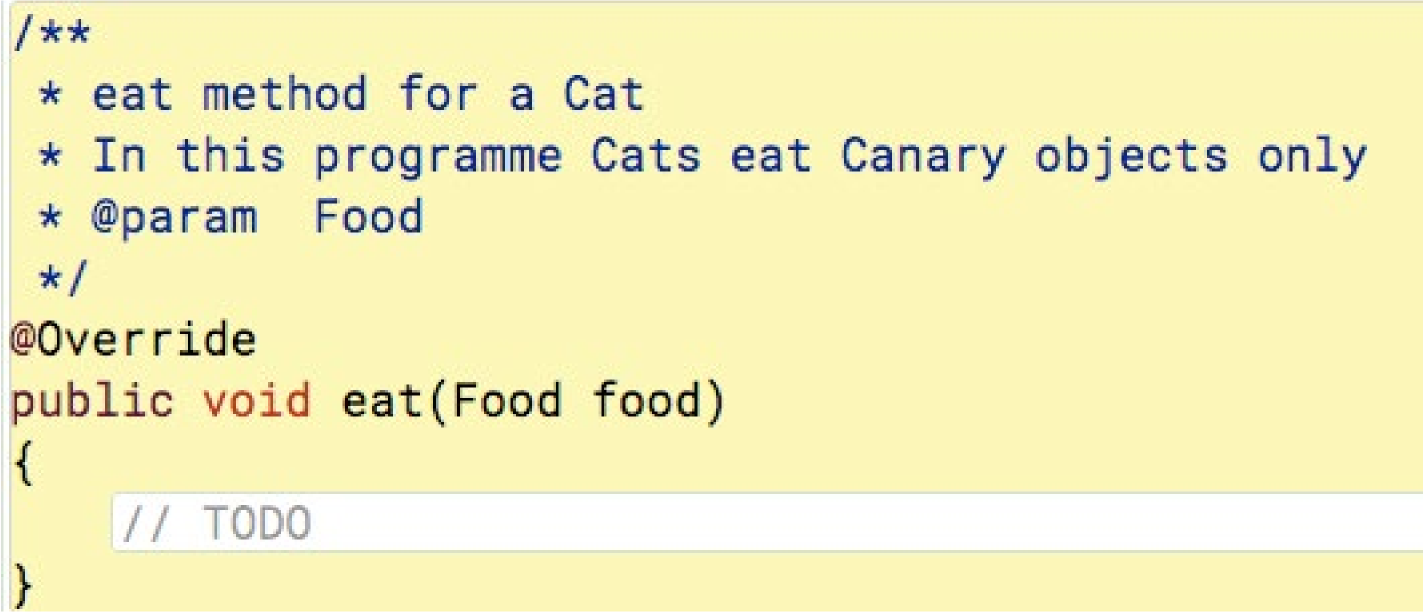


Now open the *eat* method of Cat

Copy and paste the body of the eat method in Canary into this method. Modify

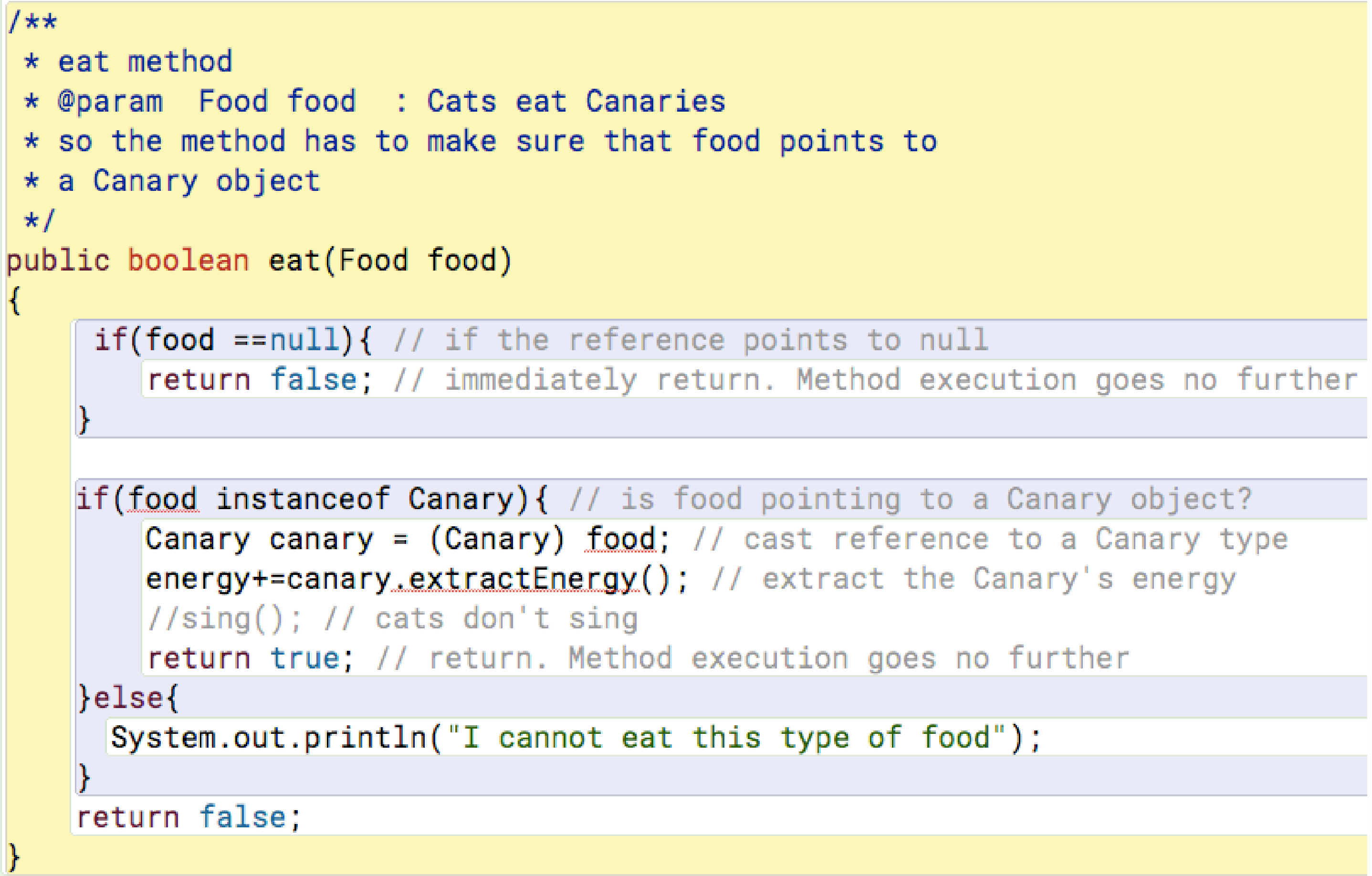
Remember a Cat can only eat a Canary

A Cat doesn’t sing



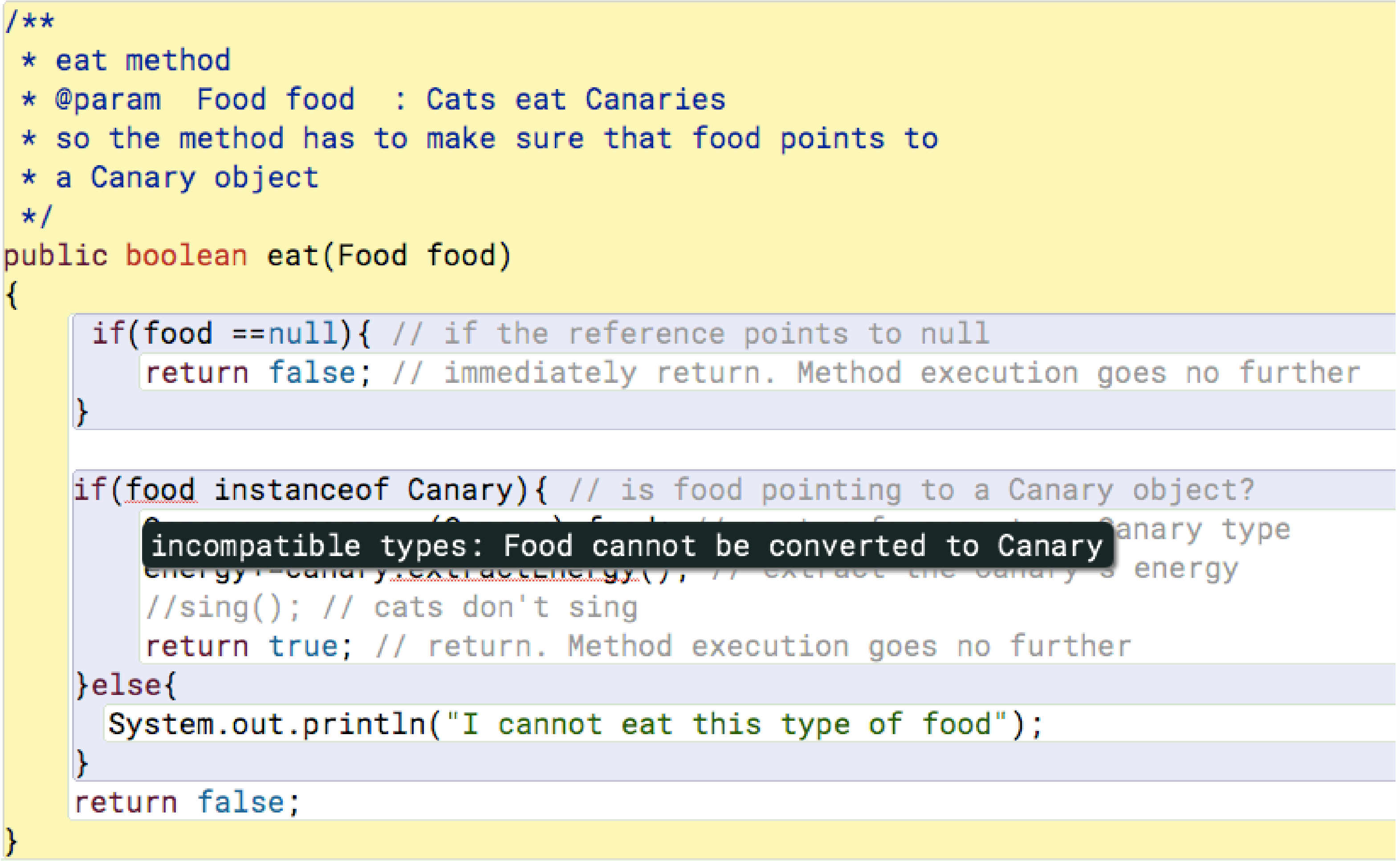
```
/**
 * eat method for a Cat
 * In this programme Cats eat Canary objects only
 * @param  Food
 */
@Override
public void eat(Food food)
{
    // TODO
}
```

What problems did you experience?



```
/**
 * eat method
 * @param  Food food   : Cats eat Canaries
 * so the method has to make sure that food points to
 * a Canary object
 */
public boolean eat(Food food)
{
     if(food ==null){ // if the reference points to null
        return false; // immediately return. Method execution goes no further
    }

    if(food instanceof Canary){ // is food pointing to a Canary object?
        Canary canary = (Canary) food; // cast reference to a Canary type
        energy+=canary.extractEnergy(); // extract the Canary's energy
        //sing(); // cats don't sing
        return true; // return. Method execution goes no further
    }else{
      System.out.println("I cannot eat this type of food");
    }
    return false;
}
```

Incompatible Types



```
/**
 * eat method
 * @param  Food food  : Cats eat Canaries
 * so the method has to make sure that food points to
 * a Canary object
 */
public boolean eat(Food food)
{
    if(food ==null){ // if the reference points to null
       return false; // immediately return. Method execution goes no further
    }

    if(food instanceof Canary){ // is food pointing to a Canary object?
       incompatible types: Food cannot be converted to Canary anary type
       energy+=canary.extractEnergy(); // extract the canary's energy
       //sing(); // cats don't sing
       return true; // return. Method execution goes no further
    }else{
      System.out.println("I cannot eat this type of food");
    }
    return false;
}
```

eat method of Cat

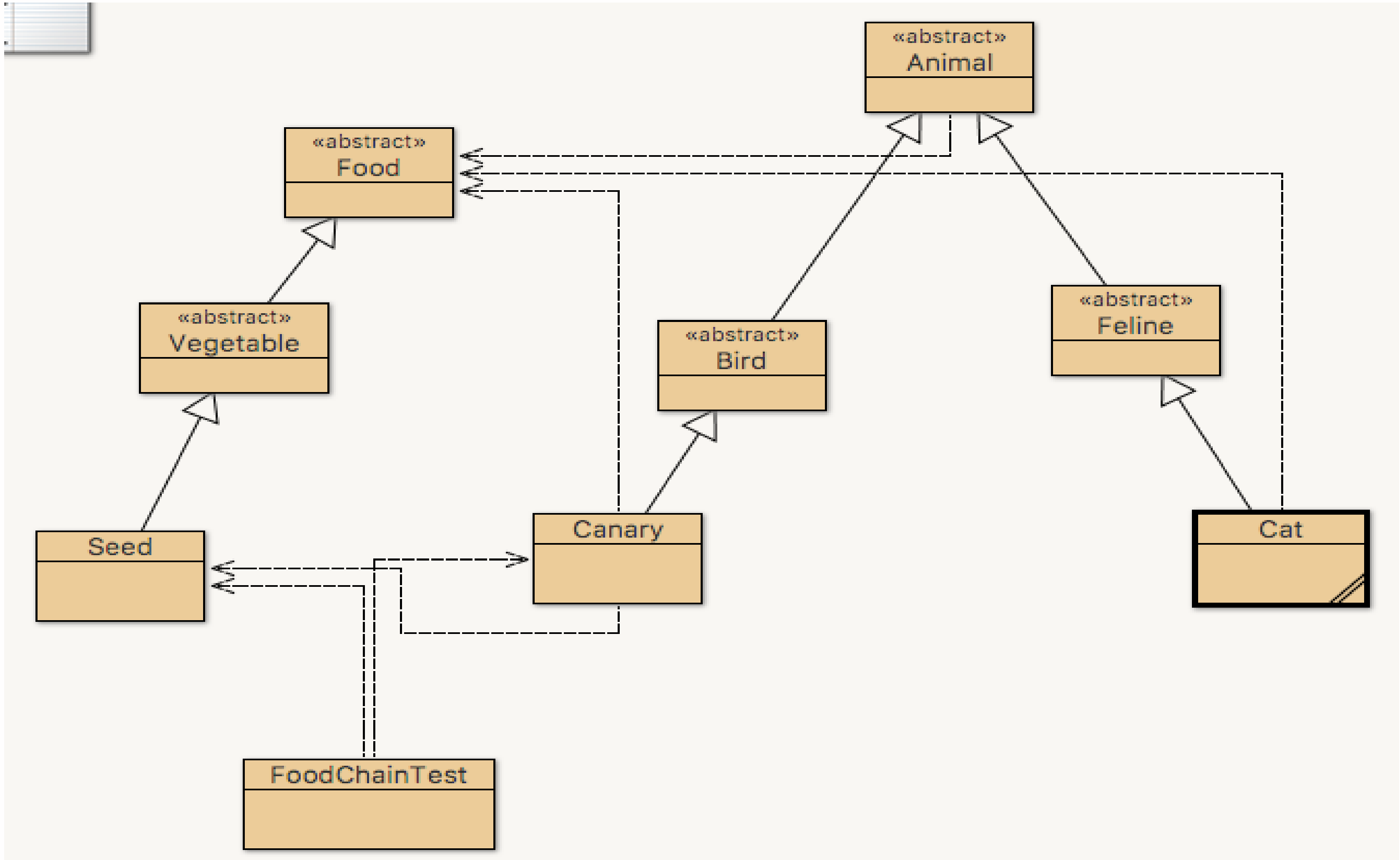
Big Problem! Food cannot be converted to Canary

However, the *eat* method only takes a Food reference as an input

In order to convert the Food reference to a Canary reference, Canary **must** be a subclass of Food, just as Seed was

But Canary is a subclass of Animal

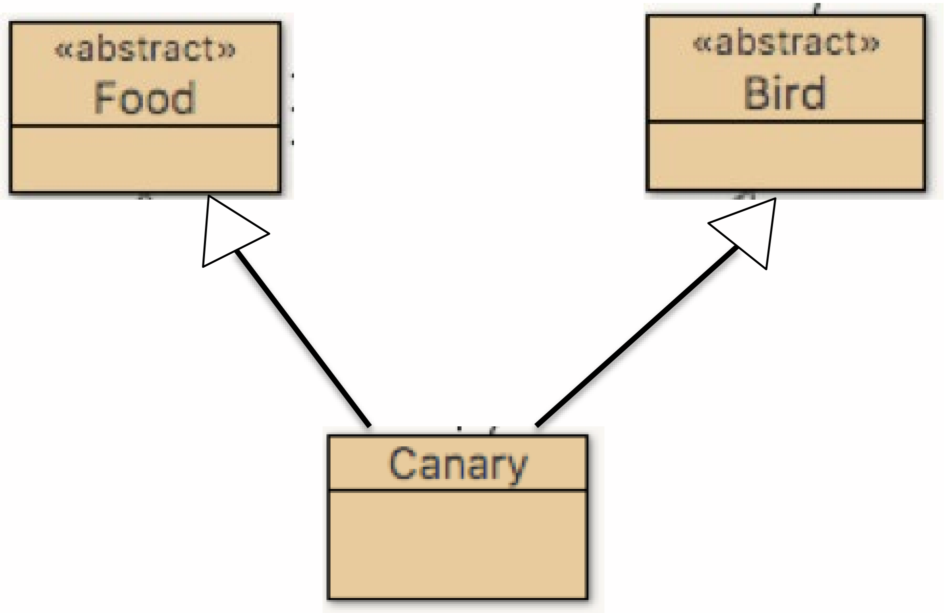
A Canary is not a Food Type



Ollscoil na Gaillimhe
University of Galway

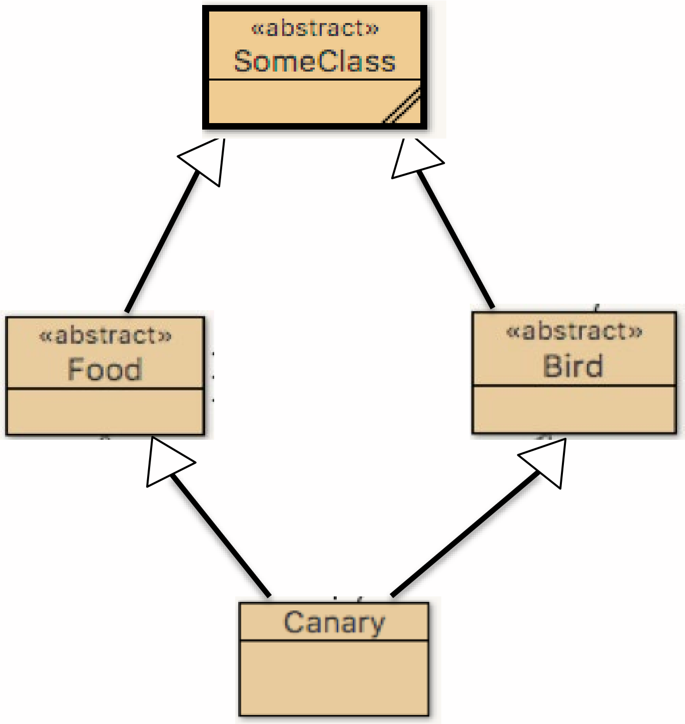
Multiple Inheritance

This problem could be solved using **multiple inheritance** – where a class can have multiple simultaneous superclasses



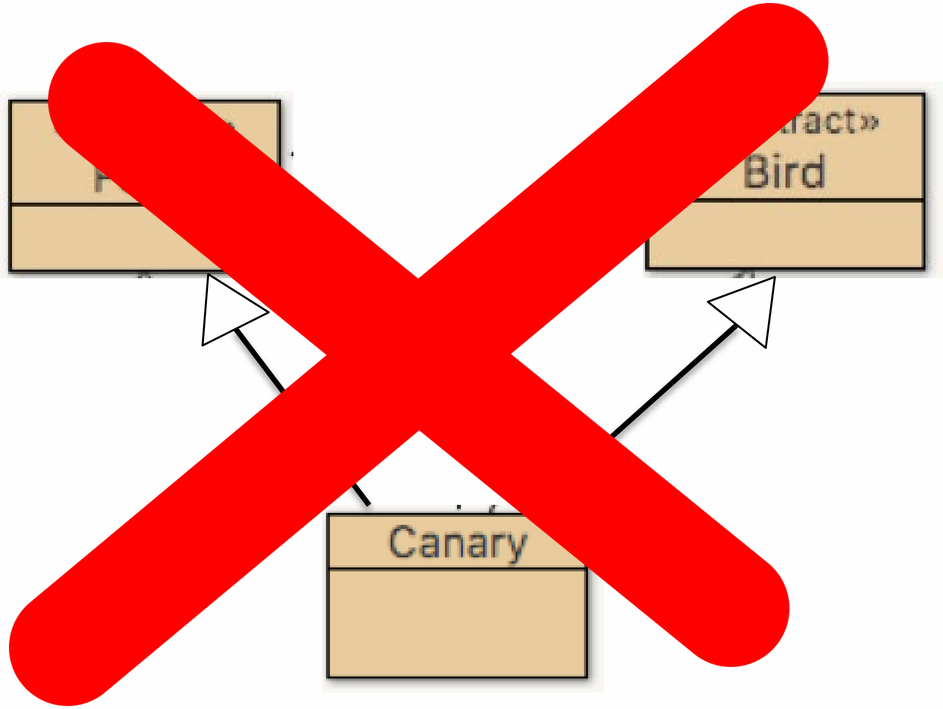
Multiple Inheritance

However, in OOP multiple inheritance has led to **major problems** due to conflicting field and method implementations inherited from superclasses



Multiple Inheritance

Java does not support multiple inheritance



Interface

Java uses a structure called an **interface** to achieve a form of multiple inheritance

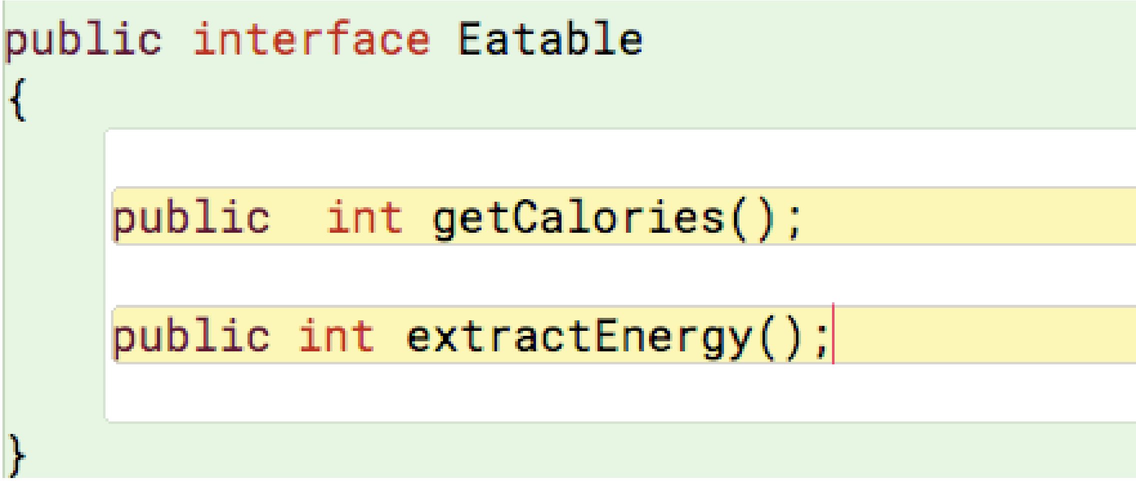
An interface is **like a class** – but it is really more like an outline of what methods a class should have

Just like a class an interface can be used **as a type**

Interface names often end in – **able** - simply by convention

Interface example

Compare and Contrast with a class definition

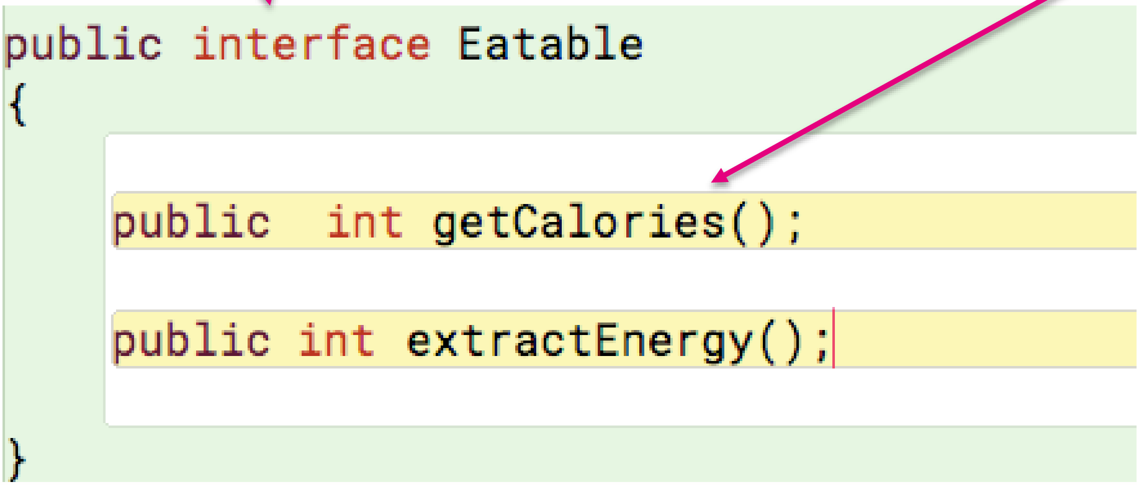


```
public interface Eatable
{

        public  int getCalories();

        public int extractEnergy();

}
```

Interface example

Note interface not class

- Note method definitions have no body



```
public interface Eatable
{

    public  int getCalories();

    public int extractEnergy();

}
```

Eatable interface

What does it mean?

1. Any class that implements Eatable can be treated as an Eatable type (Polymorphism)
2. Any class that implements Eatable must provide **concrete implementations** of its method

Implementing an interface

While a class can only extend one superclass (direct inheritance)
It can implement **multiple** interfaces

Food as an interface

What does it mean?

1. Any class that **implements** Food can be treated as a Food type (Polymorphism)
2. Any class that implements Food must provide **concrete implementations** of its method

Implementing an interface

A class can only extend one superclass (direct inheritance)
A class can implement **multiple** interfaces
the following class declaration is valid:

```
public class Canary extends Bird implements Food, Comparable{
…
}
```

“A Canary is a subclass of Bird and implements the interfaces Food and Comparable”

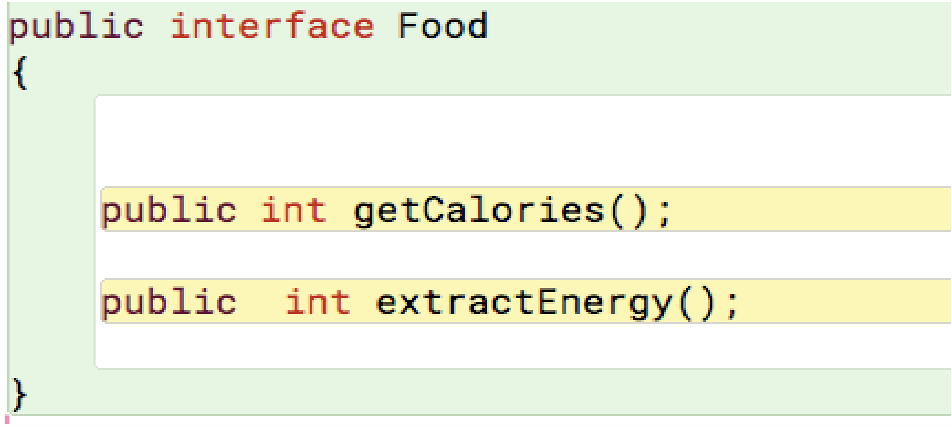
Solving the Cat’s eating problem

We are going to make the Food class into an interface

Any object that is edible (in our domain) will be required to implement the Food interface.

Step 1:

- Change Food to be an interface



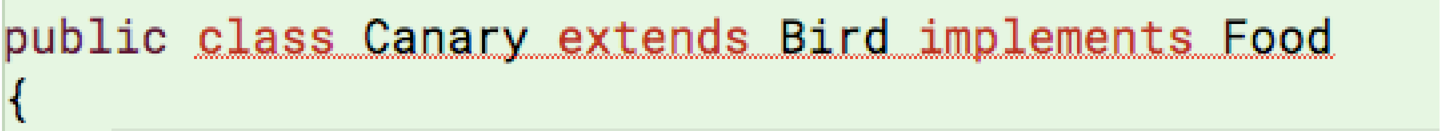
```
public interface Food
{


    public int getCalories();

    public  int extractEnergy();

}
```

- This also will require Vegetable to **implement** the Food interface
- Seed will need to have its own version of the *calories* field

Step 2

We want Canary to be considered a type of Food

Therefore, Canary should implement the Food Interface



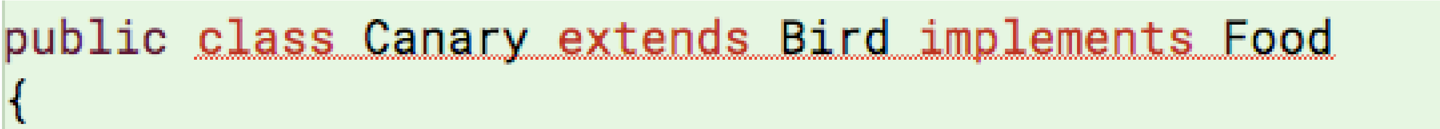
```
public class Canary extends Bird implements Food
{
```

Canary will be required to implement the Food interface’s two methods

- getCalories
- extractFood

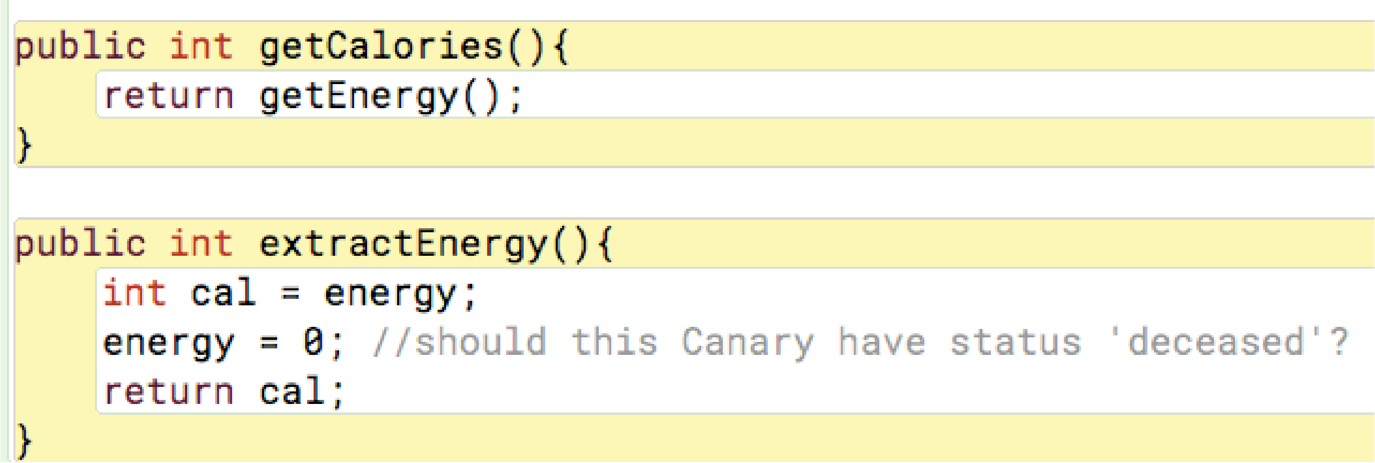
Step 2

Canary should implement Food



```java
public class Canary extends Bird implements Food
{
```

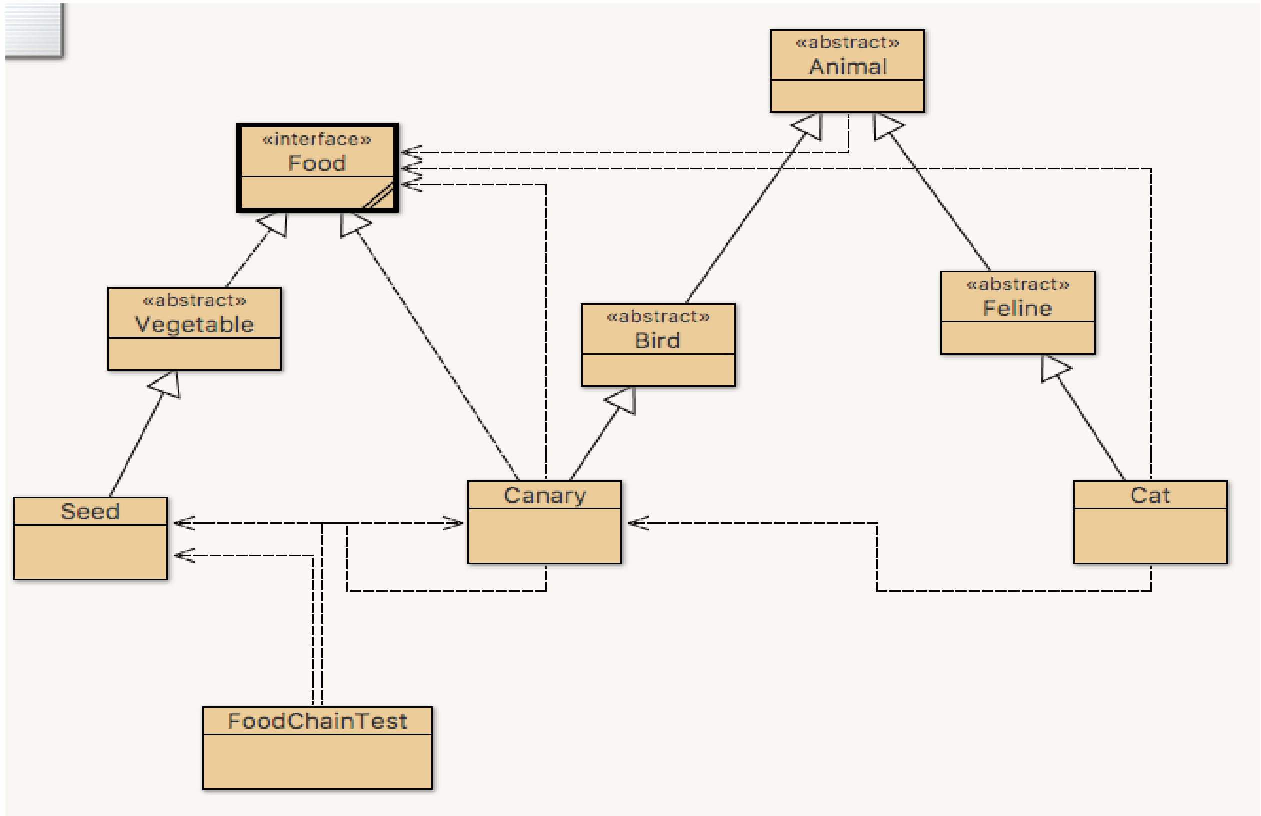
Canary will also be required to implement Foods two methods



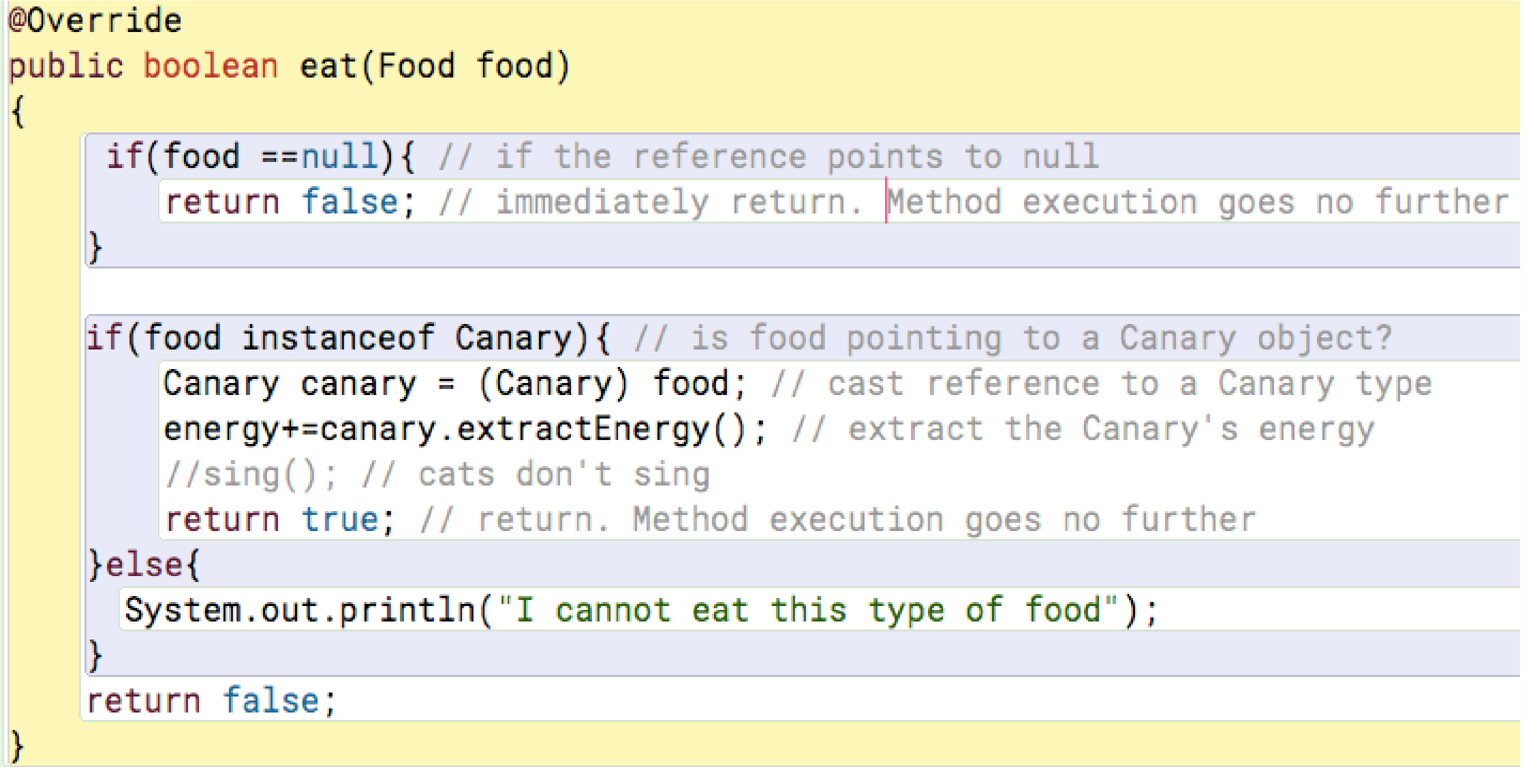
```java
public int getCalories(){
    return getEnergy();
}

public int extractEnergy(){
    int cal = energy;
    energy = 0; //should this Canary have status 'deceased'?
    return cal;
}
```

If you've followed these instructions, you should find that the *eat* method of Cat now compiles

A Canary is now a Food type as it implements the Food interface



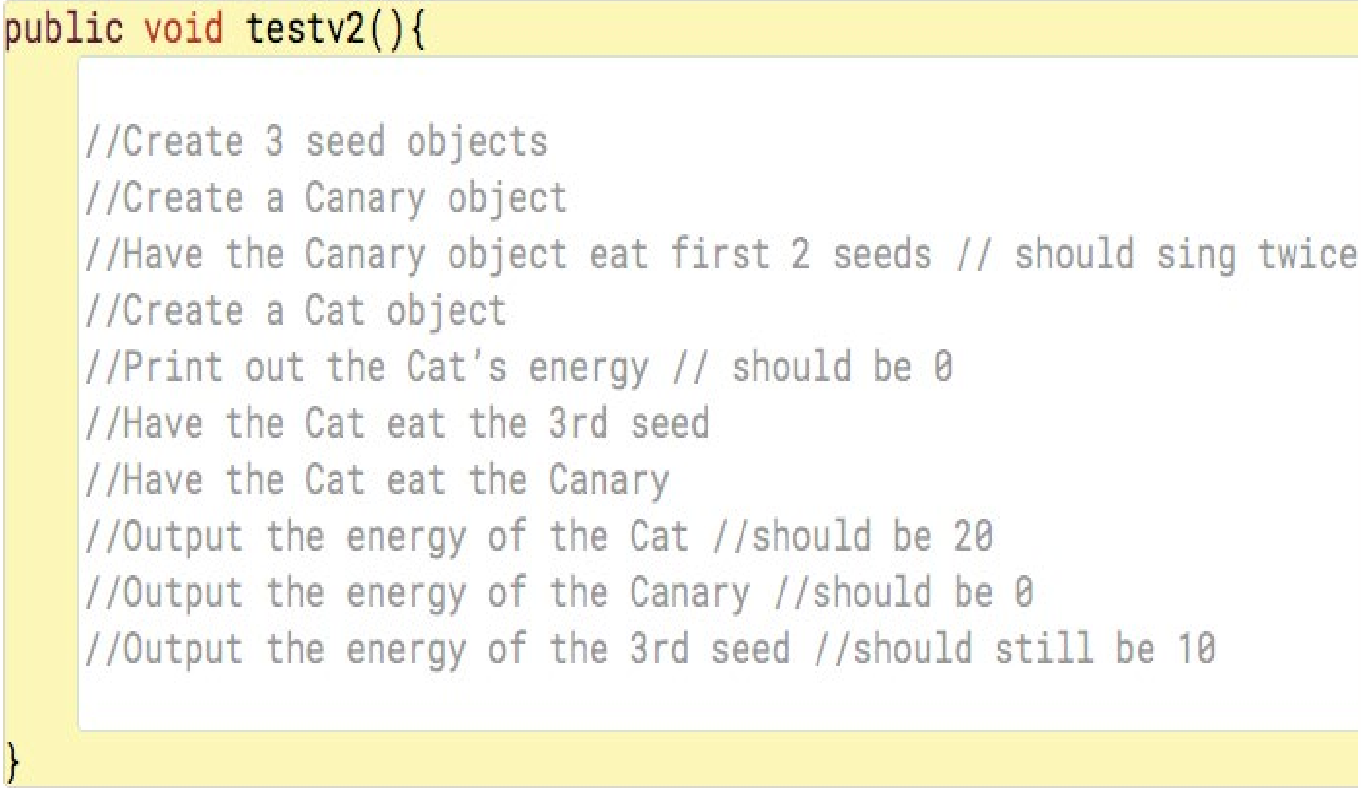
Cat’s eating problem solved



```java
@Override
public boolean eat(Food food)
{
     if(food ==null){ // if the reference points to null
        return false; // immediately return. Method execution goes no further
    }

    if(food instanceof Canary){ // is food pointing to a Canary object?
        Canary canary = (Canary) food; // cast reference to a Canary type
        energy+=canary.extractEnergy(); // extract the Canary's energy
        //sing(); // cats don't sing
        return true; // return. Method execution goes no further
    }else{
      System.out.println("I cannot eat this type of food");
    }
    return false;
}
```

Test your code

- Write a new test method in the FoodChainTest class
- Call it testv2
- Write Code to execute the code instructions in the comments below (Reuse some of the code in the testv1 method)
- Execute the method in the main method
- Check that the output is as expected



```
public void testv2(){

        //Create 3 seed objects
        //Create a Canary object
        //Have the Canary object eat first 2 seeds // should sing twice
        //Create a Cat object
        //Print out the Cat’s energy // should be 0
        //Have the Cat eat the 3rd seed
        //Have the Cat eat the Canary
        //Output the energy of the Cat //should be 20
        //Output the energy of the Canary //should be 0
        //Output the energy of the 3rd seed //should still be 10

}
```

Interface vs Abstract class: **Similarities**

Similarities:

- Both can be used to provide '*templates*' for what subclasses can implement
- An abstract method plays the same role as an interface method – Both *must* be implemented in concrete form by a subclass
- An abstract class and an Interface can be used as the **type** for a reference variable.

  E.g. `Food tasty = new Canary(“tasty”);`
- This code works **if** Food is an abstract class or Interface

Interface vs Abstract class: **Differences**

Differences:

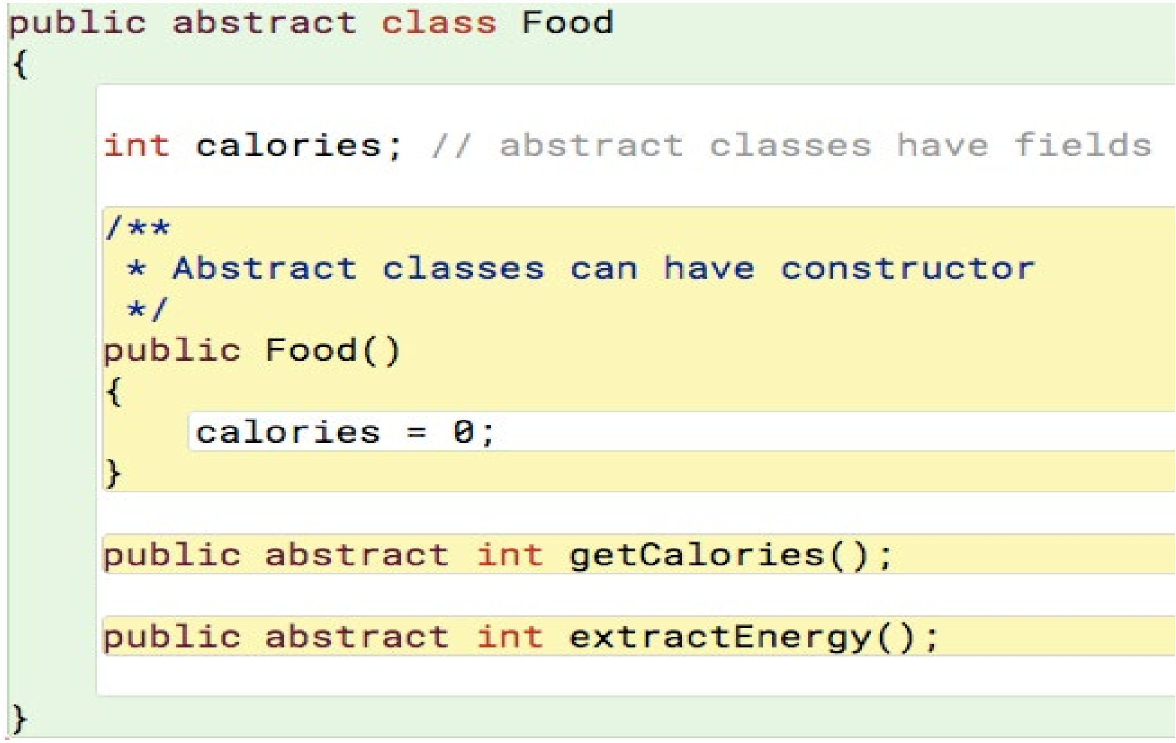
- An abstract class is used for classic inheritance purposes – providing an abstract structure that subclasses inherit. The subclasses have a lot *in common*.
- E.g. the abstract class Bird provides common functionality for all feathered, winged animals

```
Bird canary = new Canary(“mary”);
Bird ossie = new Ostrich(“ossie”);
```

- However, an interface is often used to impose common functionality on classes that have nothing in common.
- E.g. The interface Food imposes common (Food) functionality on two quite different classes : Seed and Canary

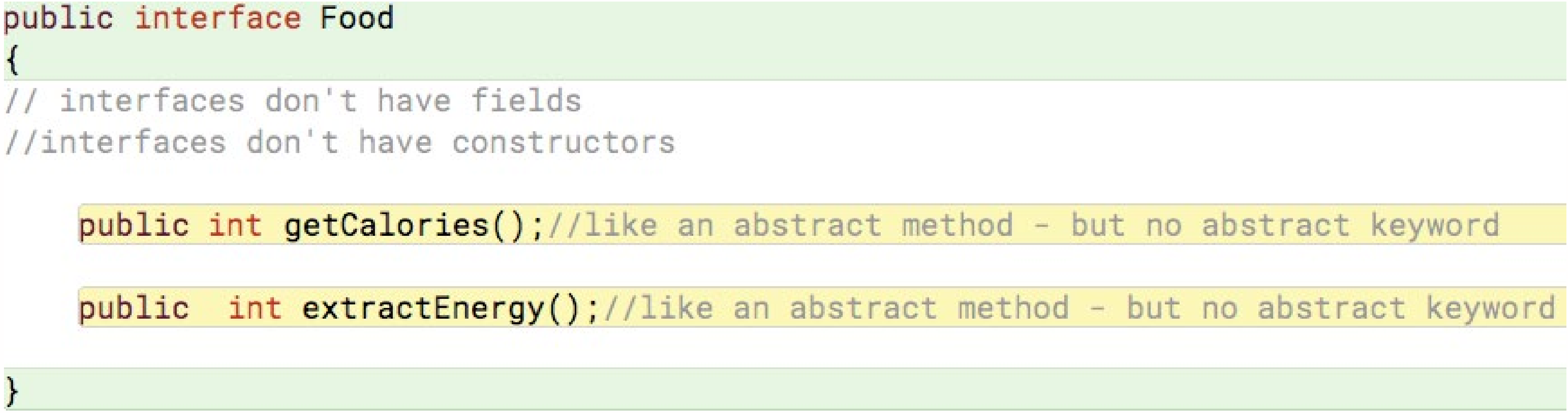
```
Food tasty = new Canary(“tasty”);
Food sunflower = new Seed();
```

On the next slide, we compare the similarities and differences between the abstract class and interface versions of Food



```java
public abstract class Food
{

    int calories; // abstract classes have fields

    /**
     * Abstract classes can have constructor
     */
    public Food()
    {
        calories = 0;
    }

    public abstract int getCalories();

    public abstract int extractEnergy();

}
```

VS



```java
public interface Food
{
// interfaces don't have fields
//interfaces don't have constructors

    public int getCalories();//like an abstract method - but no abstract keyword

    public  int extractEnergy();//like an abstract method - but no abstract keyword

}
```

Differences/Similarities: Syntax

- An abstract class has the term **abstract class** in its class declaration
- An interface has the term **interface** in its declaration
- An abstract class may have fields; an interface usually will not*
- An abstract class may have a constructor; an interface will not
- A class will use the keyword **extends** in its class declaration when inheriting from an abstract class
- A class will use the keyword **implements** in its class declaration to indicate that it will implement an interface
- A class can only extend one superclass (abstract or concrete). However, it can implement **multiple** interfaces
- An abstract class may have a concrete method; an interface will not
- An abstract method has the **abstract** keyword in its method declaration; an interface method does not
- An interface method and an abstract method do not have a method body

***When fields are declared in an Interface, they are public, static and, final by default**

We will not be covering examples with fields declared in Interfaces