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

Lecture Topics

- Abstract classes and methods
- Polymorphism

Abstract

- It may not make sense to have an object of type superclass. E.g. Animal or Bird
- E.g. have you ever seen an Animal or Bird object walking/flying about?
- You've seen *specific* types of Animals and specific types of Birds
- Animal and Birds are **abstractions**

Abstraction

1. variable noun

An abstraction is a general idea rather than one relating to a particular object, person, or situation.

Collins English English Dictionary

https://www.collinsdictionary.com/dictionary/english/abstraction

Ollscoil na Gaillimhe
University of Galway

Abstract Keyword

- You can declare a class to be **abstract**

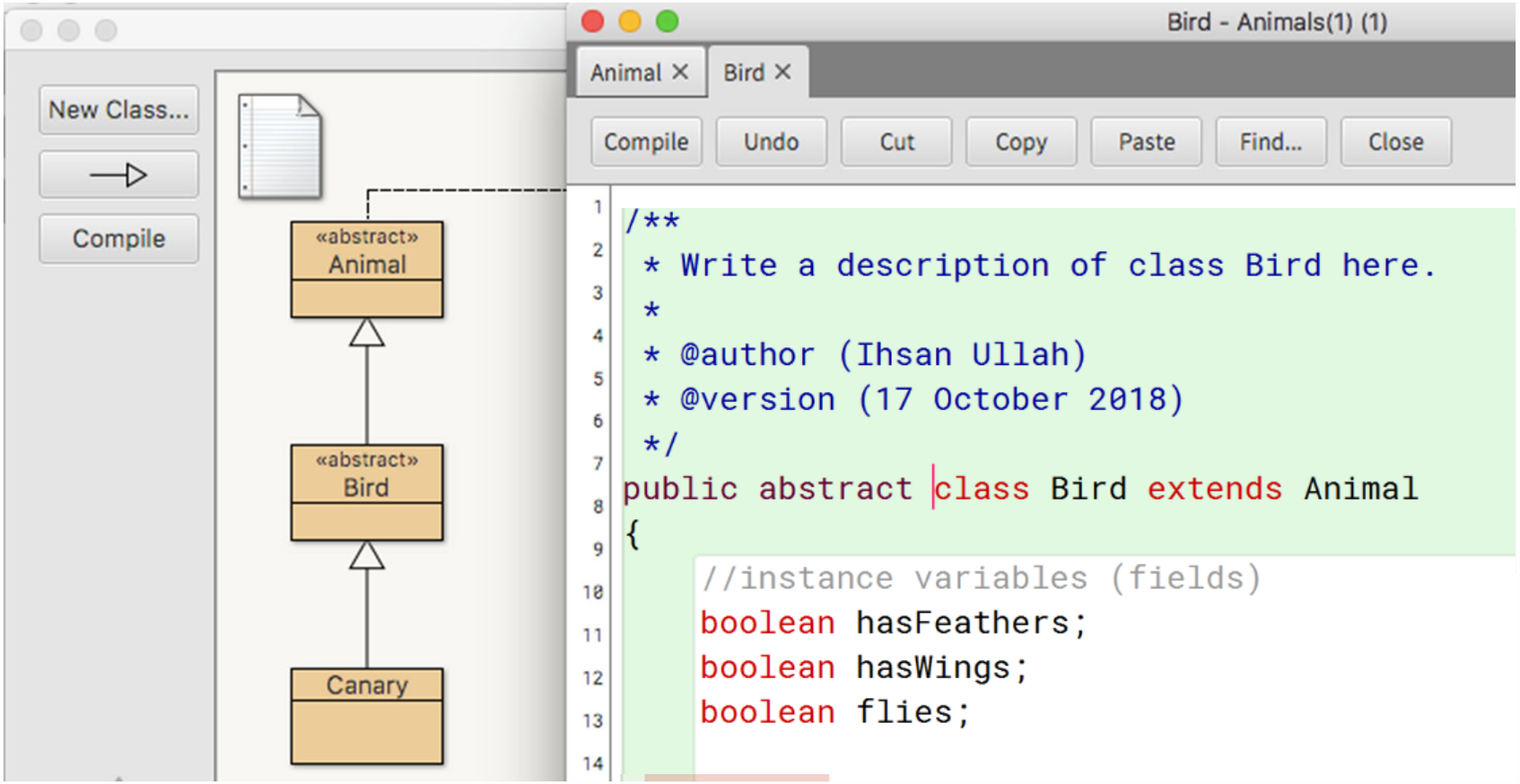
```
public abstract class Animal
{
```

```
public abstract class Bird extends Animal
{
```

- Java allows you to specify which classes can be made into objects
- ..and which are **abstract** and used just for inheritance purposes

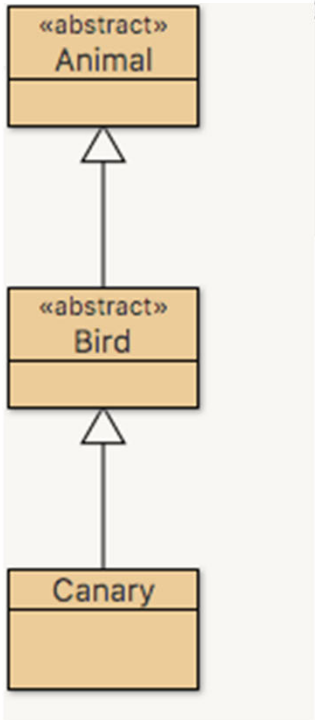
Code

In BlueJ Make the Animal and Bird classes **abstract**



abstract Keyword

Adding the word **abstract** to the class definition tells Java that it can't make objects from this class
Now, as you did before, try to create an Animal and Bird object




```
Animal animal = new Animal();
    Error: Animal is abstract; cannot be instantiated
Bird bird = new Bird();
    Error: Bird is abstract; cannot be instantiated
```

abstract

- First effect is that you no longer can create objects from the abstract class
- However, all the existing rules of inheritance still apply

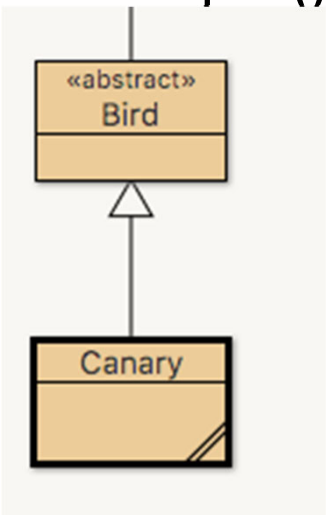
```
public abstract class Bird extends Animal
{
    //instance variables (fields)
    boolean hasFeathers;
    boolean hasWings;
    boolean flies;

    /**
     * Constructor for objects of class Bird
     */
    public Bird()
    {
        super(); //calls the constructor of its
        colour = "black"; //overrides the value
        hasFeathers = true; //all the subclasse
        hasWings = true; //all the subclasses o
        flies = true; //all the subclasses of B
    }
```

- Sub-classes of Bird inherit its non-private fields

abstract

Even though Bird is declared as an abstract class a subclass (e.g. Canary) still has to invoke super()

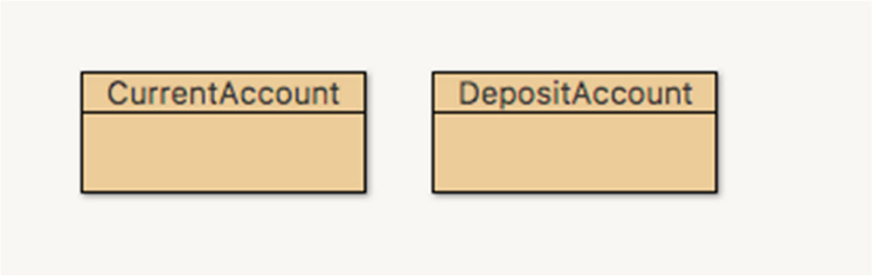



```
/**
 * Constructor for objects of class Canary
 */
public Canary(String name)
{
    super(); // call the constructor of the superclass Bird
    this.name = name;
    colour = "yellow"; // this overrides the value inherited from Bird

}
```

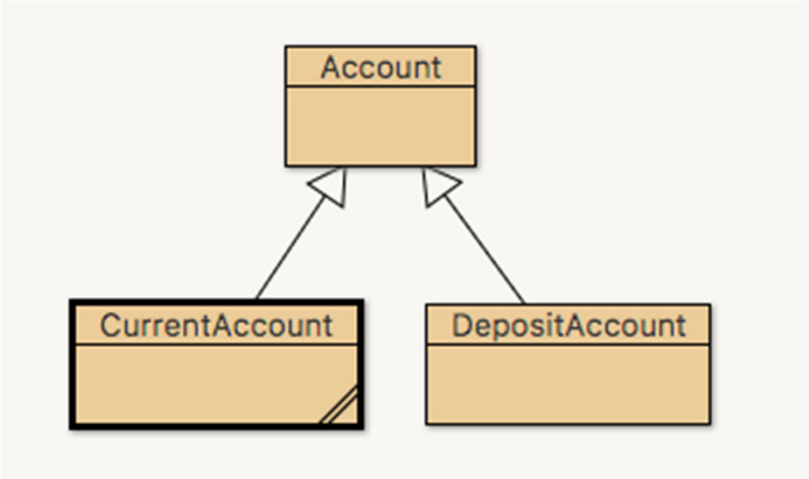
Why use an abstract class?

- In situations where you want to use inheritance but do not want another developer to create an object from the superclass.
- E.g a banking app has two bank account types :
- Current Account and Deposit Account



Why use abstract

- Both account types share many of the same fields and methods
- So the developer creates a superclass, Account, to hold all the shared fields and methods



Why use abstract

- However a trainee developer then writes the following line of code

```
Account account = new Account();
```

- This is a problem as there is no such thing in the Banking app as an Account.
- An account must either be a Current Account or a Deposit Account

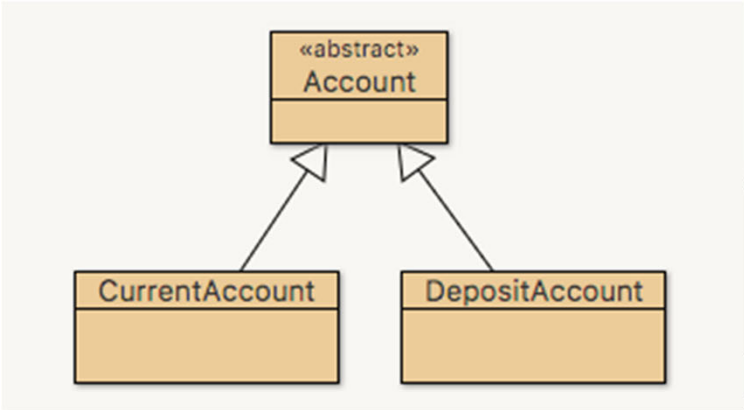
To prevent this happening, the senior developer declares the Account class abstract

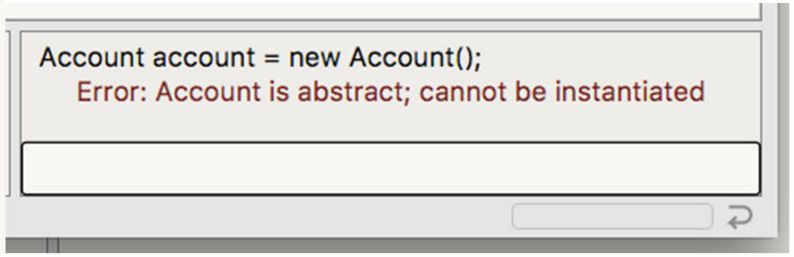
```
public abstract class Account
```

Why use abstract

As before, CurrentAccount and DepositAccount still inherit fields and methods from the abstract Account class

But Account itself cannot be *instantiated* (an object cannot be made of it)





Methods in an abstract class

As you've seen, an abstract class can have standard methods
These methods are inherited automatically by the subclass

```
/**
 * move method
 * param int distance - the distance the Animal should move
 * All subclasses inherit this method
 */
public void move(int distance){
    System.out.printf("I move %d metres \n", distance);
}
```

Ollscoil na Gaillimhe
University of Galway

Methods in an abstract class

As we've seen, a subclass can **override** (provide their own specific implementation) of the inherited methods

```
/**
 * the move method in Bird overrides the move method
 * inherited from superclass Animal
 */
@Override // good programming practice to use @Override to denote overridden methods
public void move(int distance){
    if(flies){
        System.out.printf("I fly %d metres \n", distance);
    }else{
        System.out.printf("I am a bird but cannot fly. I walk %d metres \n", distance);
    }
```

Ollscoil na Gaillimhe
University of Galway

e.g. this is the overriden move method in the Bird class

Abstract methods

- Abstract classes can also have **abstract methods**
- Abstract methods are methods **with no body**

E.g.
```
public abstract void sing();
```

- In other words, they do nothing
- So what are abstract methods used for?

Demonstration

- Open up the *Animal* class in BlueJ
- Go to the *move* method

```
public void move(int distance){
    System.out.printf("I move %d metres \n", distance);
}
```

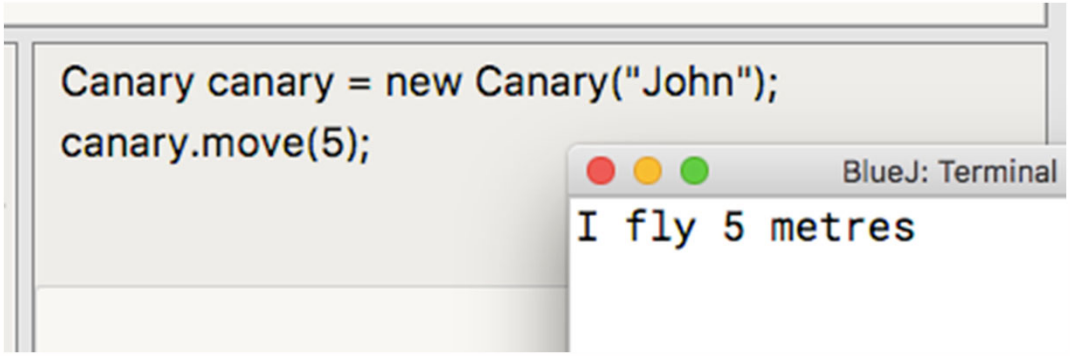
- Make it an **abstract method**
- This involves removing its body and simply keeping the method signature followed by a ';'

- Now compile the full project

```
public abstract void move(int distance);
```

Demonstration

- Your code still compiles
- In code pad, type the the following (hit return after each line)



- Where is the move functionality coming from?
- **From Bird's move method**

Demonstration

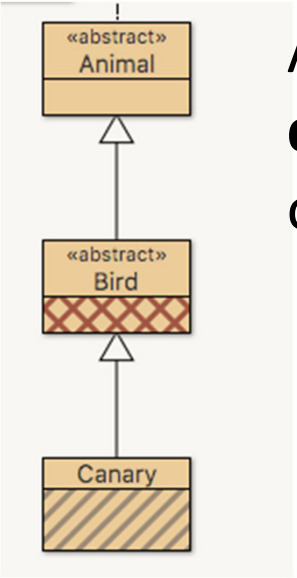
Canary's move functionality comes from Bird

Now delete (or comment out) the move method from Animal

```
/**
 * move method
 * param int distance - the distance the Ani
 * All subclasses inherit this method
 */
//public abstract void move(int distance);
```

Recompile your project

Now Bird won't compile
Check what the error is
So what is the role of *move* in Animal ?



As an abstract method, it provides the **definition** of a method that at least one of its subclasses **must implement**

The meaning of the the abstract method *move* in the Animal class:

*"All animals **must move**, but it is up to each specific animal to decide how it moves"*

Concrete

- The adjective **concrete** is often used in OOP to denote a class or method that <u>is not</u> abstract
- i.e. The class or method is fully implemented
- In our example, Canary is a concrete class
- The move method in Bird is a concrete method

Reference Type

An abstract class is **often** used as the type of a reference variable
Try this in code pad

```
Bird bird = new Canary("John");
Animal animal = new Canary("Mary");
```

Here we have two concrete objects referenced by variables whose type is an abstract class
Very common approach in OOP

abstract class and method summary

- The **abstract** keyword allows you to represent a class that should not be instantiated (made an object of)
- Inheritance from the abstract class happens the same as before
- An abstract class may have concrete and **abstract method**s
- An an abstract method does not have a method body
- It is there to provide a definition of a method that at least one of its subclasses must implement (make concrete)
- In our case – having an abstract method move is like saying "All animals **must move**, but it is up to each animal to decide how it moves"

Polymorphism

Ollscoil na Gaillimhe
University of Galway

Polymorphism

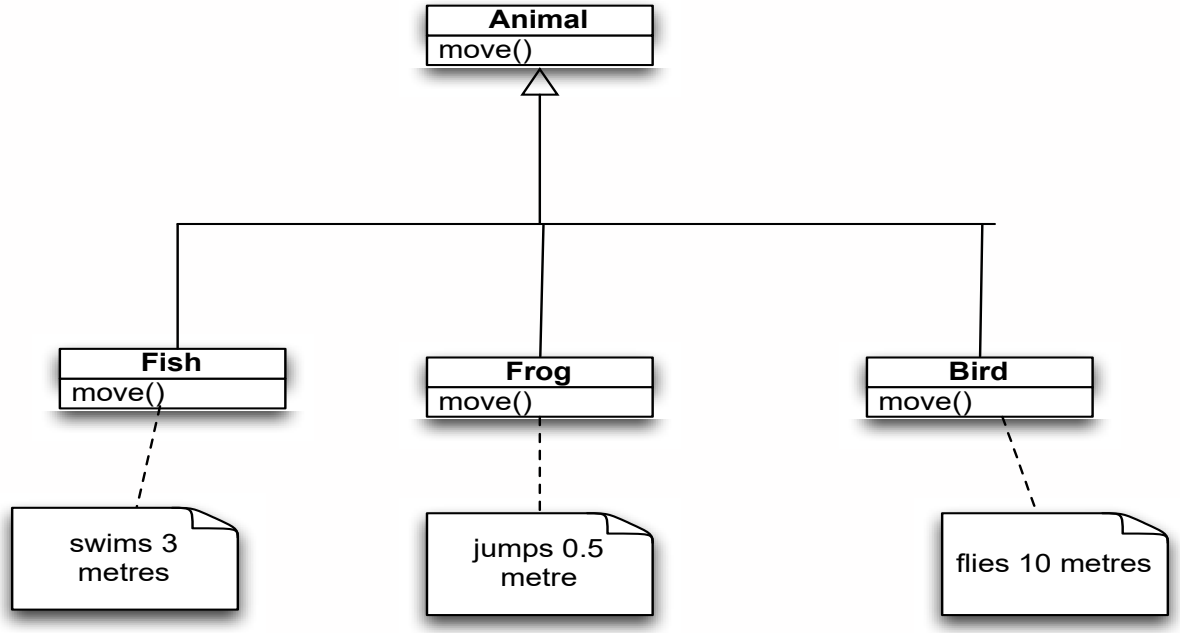
- **Polymorphism** (from Greek *polys*, "many, much" and *morphē*, "form, shape")
- Polymorphism refers to how an object can be treated as belonging to several types as long as those types **are higher** than the object's type in the class hierarchy
- Thus, In the code snippet below, a Canary can be treated as a **Bird** type and as an **Animal** type

```
Bird bird = new Canary("John");
Animal animal = new Canary("Mary");
```

Example

Open a new Project in Blue J, create an abstract class called Animal with one abstract method *move*

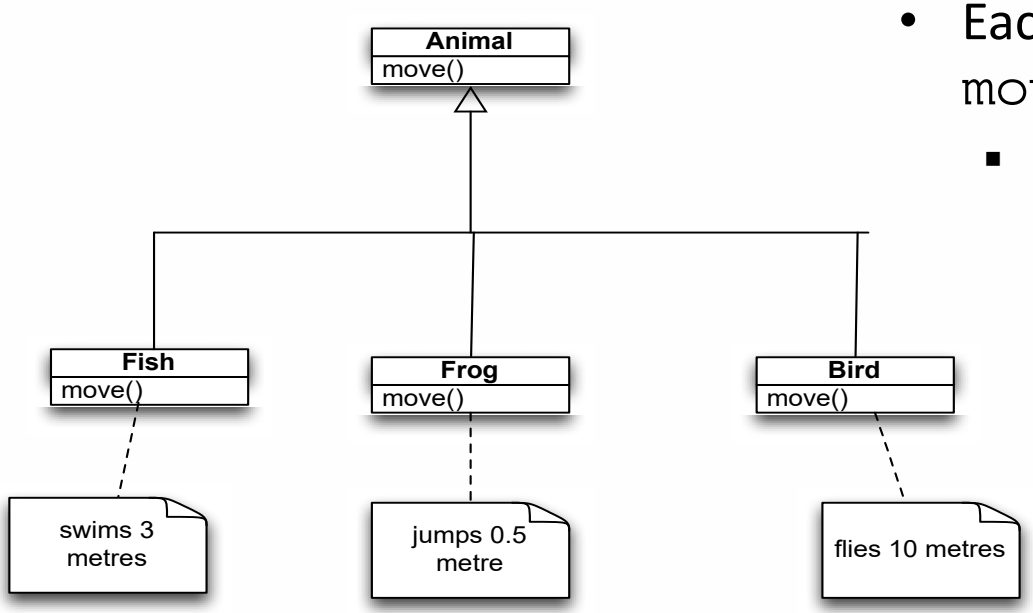
Write the code for three subclasses: Fish, Frog and Bird



Ollscoil na Gaillimhe
University of Galway

Example

- Open a new Project in Blue J, create an abstract class called Animal with one abstract method *move*
- Create three sub-classes of Animal: Fish, Frog, Bird



- Each inherits and **overrides** the `move()` method
  - A Fish swims, a Frog jump, a Bird Flies

Animal Code

```java
public abstract class Animal
{

    public abstract void move(int y);

}
```

```java
public class Bird extends Animal
{
    @Override
    public void move(int y)
    {
        System.out.printf("I fly %d metres", y);
    }
}
```

```java
public class Fish extends Animal
{
    @Override
    public void move(int y)
    {
        System.out.printf("I swim %d metres", y);
    }
}
```

```java
public class Frog extends Animal
{
    @Override
    public void move(int y)
    {
        System.out.printf("I hop %d metres", y);
    }
}
```

Polymorphism Key point

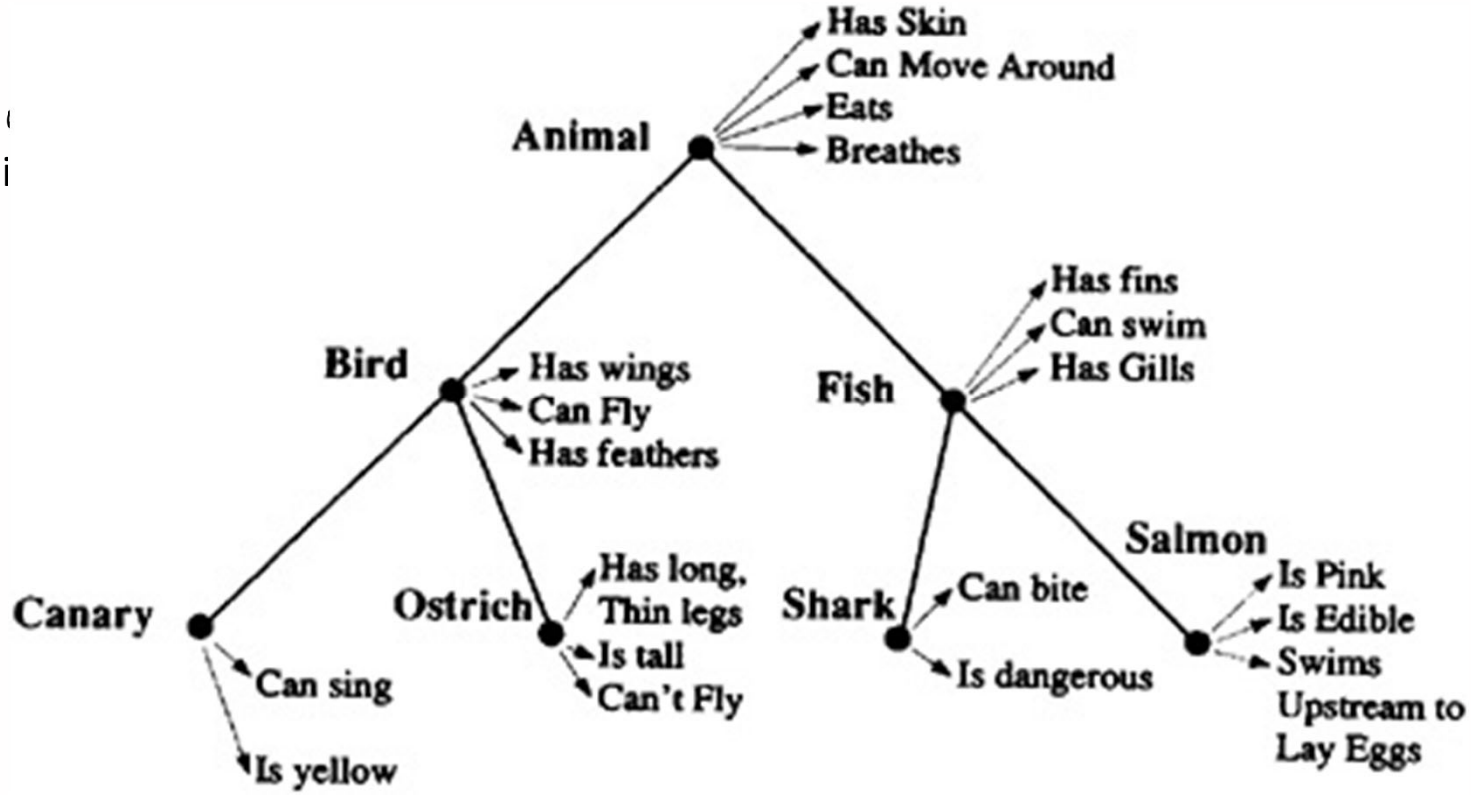
- In general, a variable of type X can point to any object that has an 'is-a' relationship to type X

```
    Animal bird1 = new Bird();
    Animal bird2 = new Bird();
    Animal frog1 = new Frog();
    Animal frog2 = new Frog();
    Animal fish1 = new Fish();
}
```

- A variable of type `Animal` can point to a `Bird`, `Frog` or `Fish` object
- `Bird`, `Frog` or `Fish` objects have an 'is-a' relationship to the `Animal` class

‘Is-a’ relationship

E.g. a variable
Animal can poi
objects of any
type **directly**
below it
in the
class
hierarchy



Codepad

Create an array of Animal references of size 6

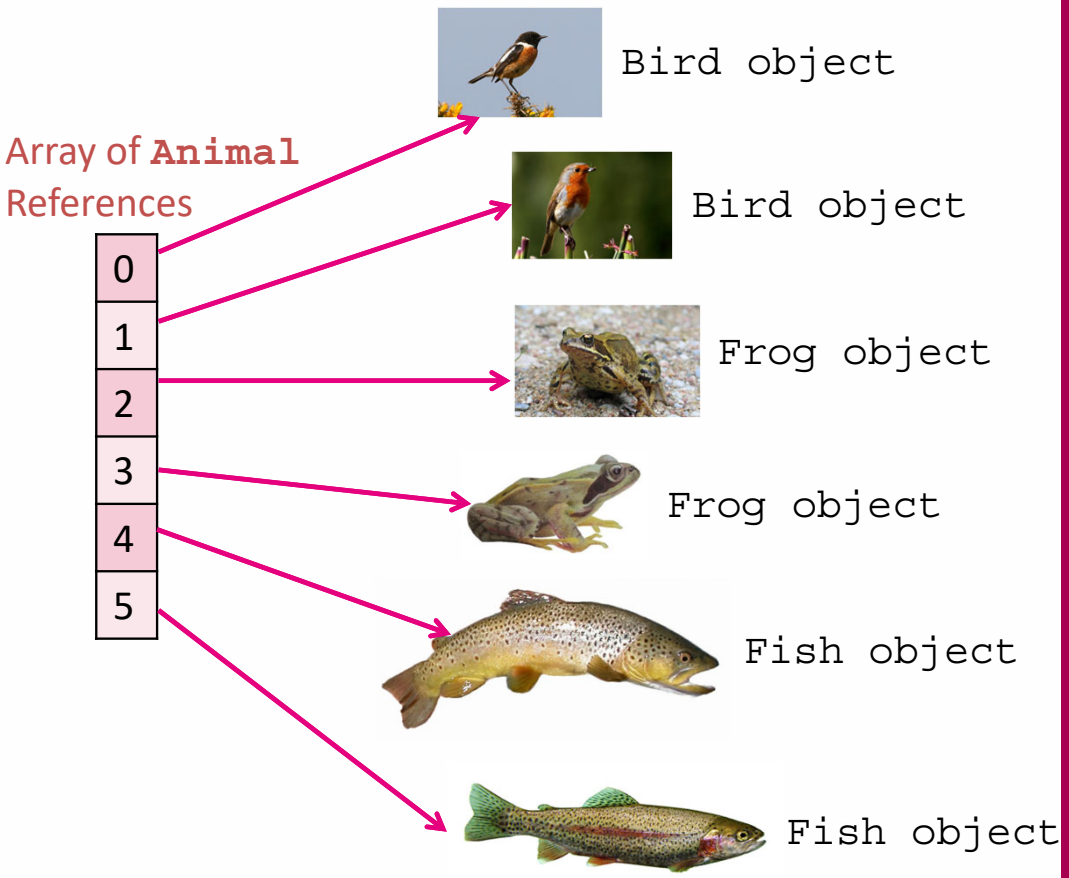
```
Animal[] animal = new Animal[6];
```

Even though Animal is an abstract class we can still create an array of Animal references

Write the code

Now write the code to add a reference to a different animal in each array location

E.g. a bird in the first location
A bird in the second location
A Frog in the third location
And so on



For tomorrow, write the code requested in the previous slide in a new Class with a main method.