CISC 3120
C09: Interface, and Abstract Class and Method

Hui Chen

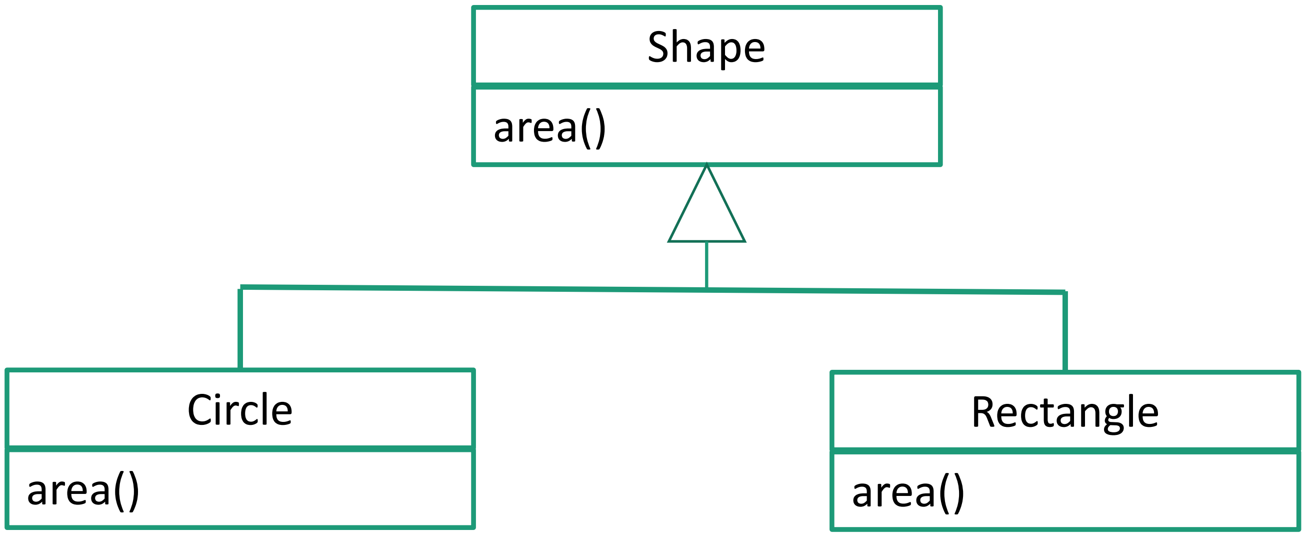
Department of Computer & Information Science

CUNY Brooklyn College

Outline

- Recap
  - Inheritance and polymorphism
  - Nested classes: inner class and static nested class
  - Assignments
- Abstract method
- Abstract class
- Interfaces
- The Object super class
- Determining object type
- Anonymous class, functional interface, and Lambda expression

Recap: Assignment Assignment W05-1_02-26



The Shape Class

- Do you like the area() method here?

```
public class Shape {                    …
  public double area() {
     System.out.println("This method is not supposed to be called.");
     return 0;
  }      …
}
```

- Remarks
  - We know semantically that each shape has a behavior to compute its area
  - However, we don't know the algorithm without knowing the actual shape
  - Do we really want to instantiate the Shape class?

Abstract Method

- An abstract method has no implementation

```
public abstract class Shape {
   public abstract double area();
}
```

In C++, sometimes called (pure) virtual function

- We shall discuss
  - Abstract class and method
  - Interface (prior to Java 8, equivalent to pure abstract class)

Abstract Class

- A class that is declared abstract
- Example

```
abstract class Animal {
…
}
```

- Abstract classes cannot be instantiated, but they can be subclassed.
- Any class that has an abstract method must be declared “abstract”

Subclass & Instantiation

- Abstract classes ***cannot*** be instantiated, but they ***can be*** subclassed.

```
abstract class Animal {

…

}
```

- How about these examples?

```
Animal animal = new Animal();
```

```
class Dog extends Animal {…}

Animal dog = new Dog();
```

Subclass & Instantiation

- Abstract classes ***cannot*** be instantiated, but they ***can be*** subclassed.

```
abstract class Animal {
...
}
```

```
Animal animal = new Animal();
```

✗

```
class Dog extends Animal {…}
Animal dog = new Dog();
```

✓

Abstract Method

- A method that is declared without an implementation

  ```
  abstract void makeNoise();
  ```

- A class that has an abstract method must be declared abstract
  - How about these examples?

```
class Animal {
  abstract void makeNoise();
}
```

```
abstract class Animal {
  abstract void makeNoise();
}
```

Class with Abstract Method

- A class that has an abstract method must be declared abstract

```
class Animal {
   abstract void makeNoise();
}
```

✗

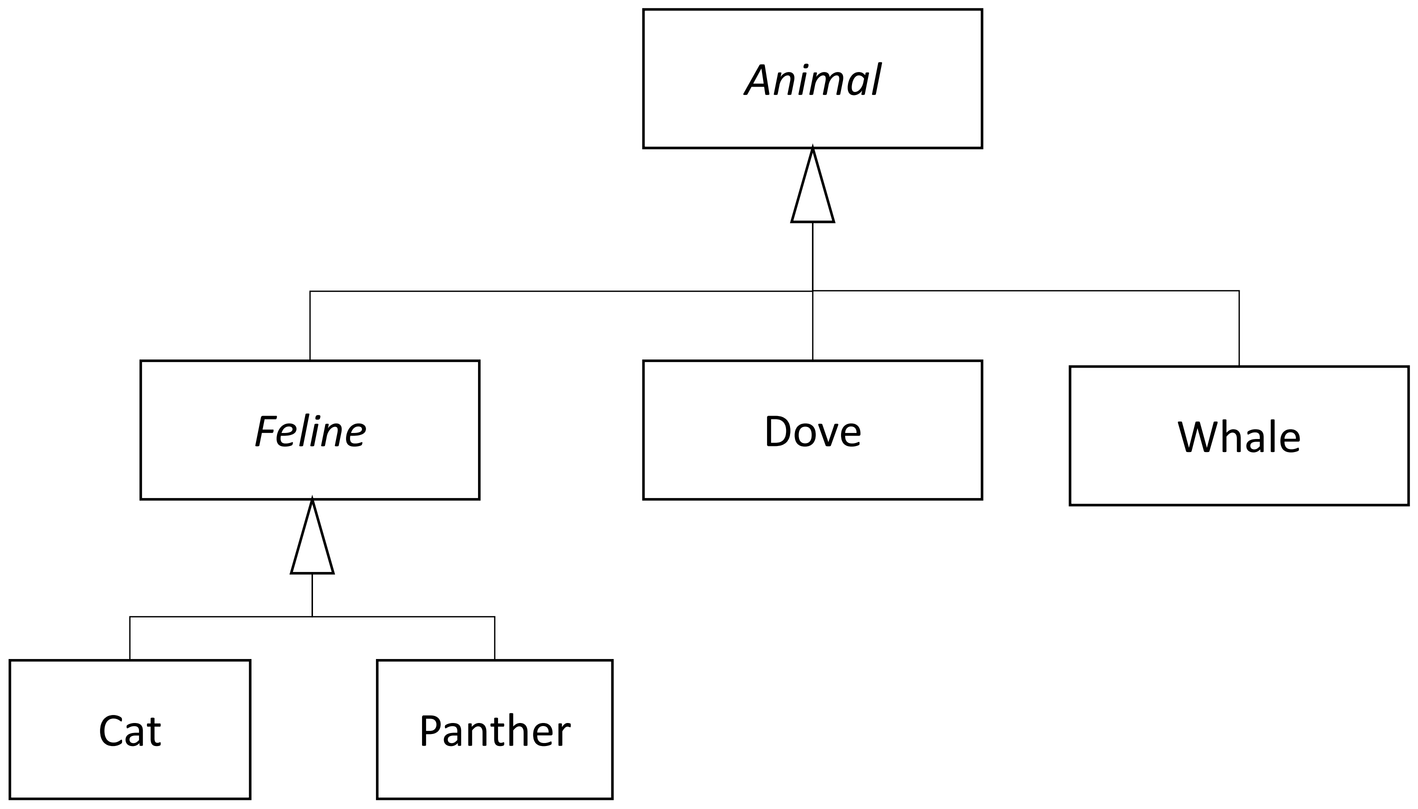
```
abstract class Animal {
   abstract void makeNoise();
}
```

✓

Subclass an Abstract Class

- Concrete subclass
  - A subclass may provide implementations for all of the abstract methods in its parent class.
- Abstract subclass
  - The subclass must also be declared abstract if it does not provide implementation of all of the abstract methods in its parent class.

Example: The Animal Kingdom



Questions?

- Abstract class
- Abstract method
- The “Animal Kingdom” example in the “Sample Programs” repository

Different Classes, Same Behaviors

- Different classes, although vastly different, may exhibit similar behavior
    - Any communication devices can “transmit” and “receive”
    - Any vehicles can “move”
    - Any objects can be “compared” to each other
    - …..

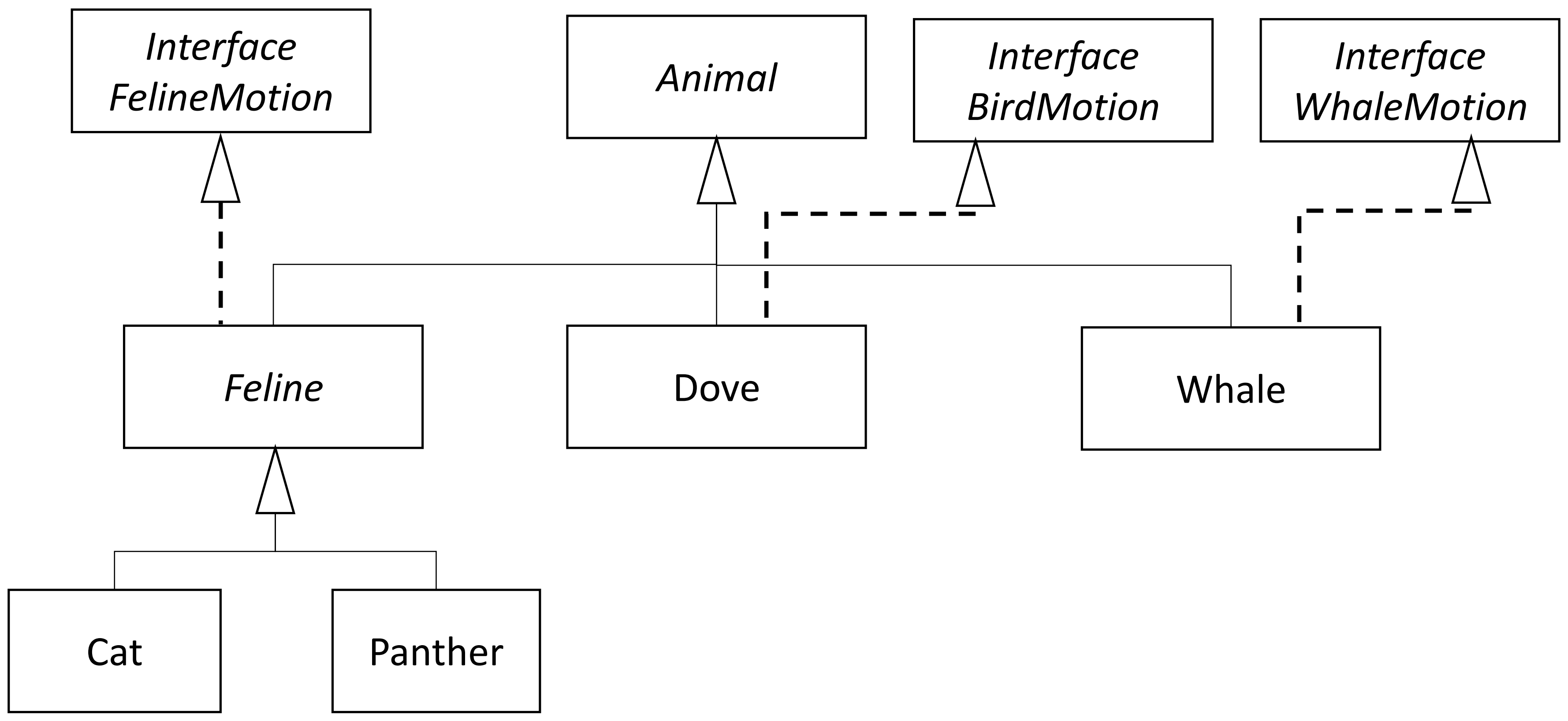
Interfaces

- Not the “interface” in “Graphical User Interface”
- Java has a reference type, called interface
  - Typically contain abstract methods only.
    - Java 8 introduces the concept of default methods and permits static methods (abstract methods with default implementation)
  - Interfaces are abstract classes, cannot be instantiated
    - can only be implemented by classes or extended by other interfaces
  - “implements” and “extends” are two distinct Java terms
    - A class “implements” an interface

Example: The Animal Kingdom Enhanced

- Different animals have different motions
  - Birds Fly
  - Whales Swim
  - And Cats ...

Example: The Animal Kingdom



Example: Birds Fly, Whales Swim, and Cats …

```
public interface BirdMotion {
        public void fly(Direction direction, double speed, double distance);
}

public interface WhaleMotion {
        public void swim(Direction direction, double speed, double distance);
}

public interface FelineMotion {
        public void walk(Direction direction, double speed, double distance);
        public void pounce(Animal prey);
}
```

Example: Implementing Interfaces

```
abstract class Feline implements FelineMotion {
…
        public void walk(Direction direction, double speed, double distance) { … }
         public void pounce(Animal prey) {  …   }
…
}


class Dove extends Animal implements BirdMotion { …
        public void fly(Direction direction, double speed, double distance) { … }
}
```

Questions?

- Interfaces
  - Why?
  - How?

Using Interface as Type

- Interfaces are data types

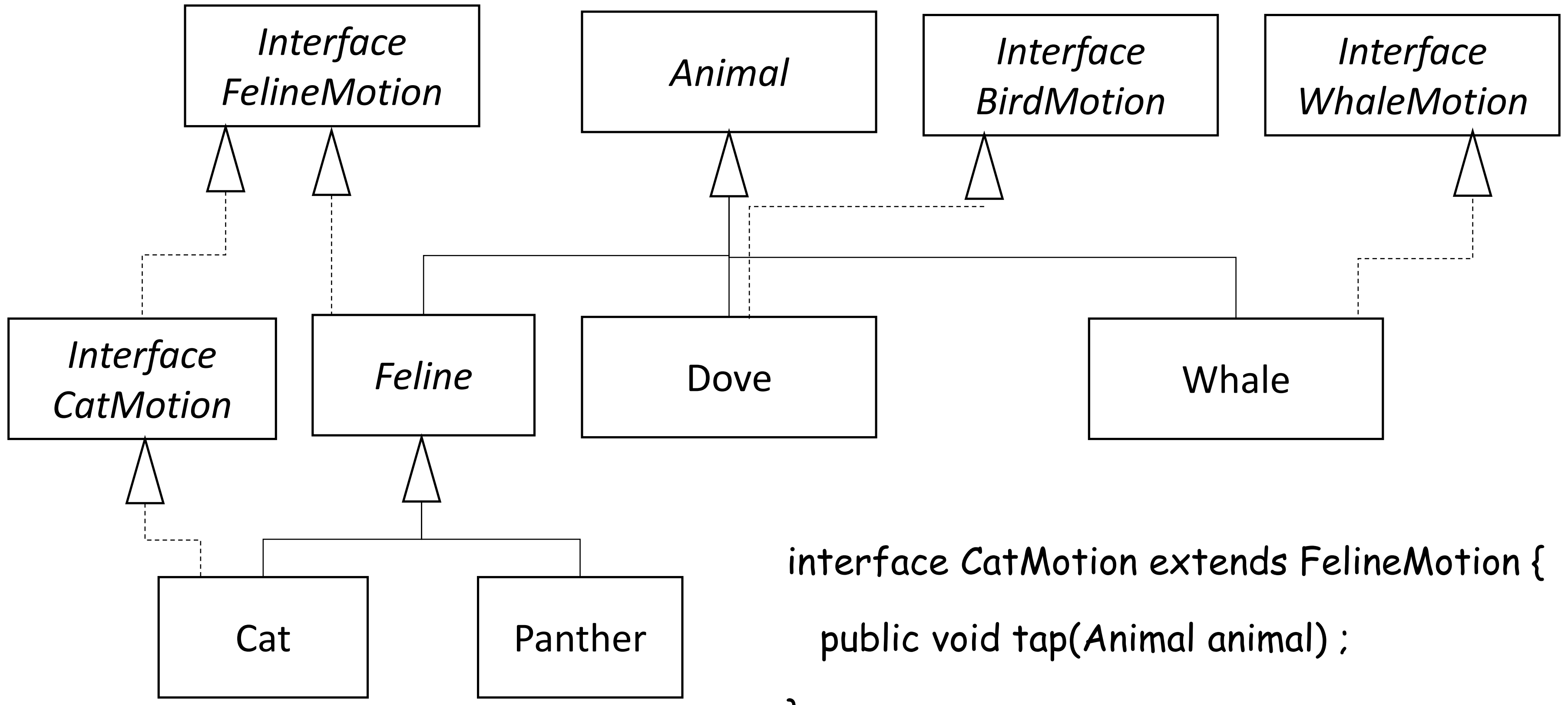
```
void flyAll(ArrayList<BirdMotion> flyingAnimals) {
…
}


Void moveBird(BirdMotion bird) {
}
```

Evolving Interfaces

- Interfaces can be extended (like classes)

```
interface CatMotion extends FelineMotion {
   public void tap(Animal animal) ;
}
```

Example: Extending FelineMotion



```
interface CatMotion extends FelineMotion {
   public void tap(Animal animal) ;
}
```

Implementing Multiple Interfaces

- A class **can** implement multiple interfaces
- But a class **cannot** extend multiple classes
- Which one of the following are is allowed in Java?

```
class FlyingCat extends
Cat, Dove {

…

}
```

```
class FlyingCat implements
BirdMotion, CatMotion {

   …

}
```

```
class FlyingCat extends
Feline implements
BirdMotion, CatMotion {

   …

}
```

Implementing Multiple Interfaces

- A class **can** implement multiple interfaces
- But a class **cannot** extend multiple classes

```
class FlyingCat extends
Cat, Dove {

…

}
```

✗

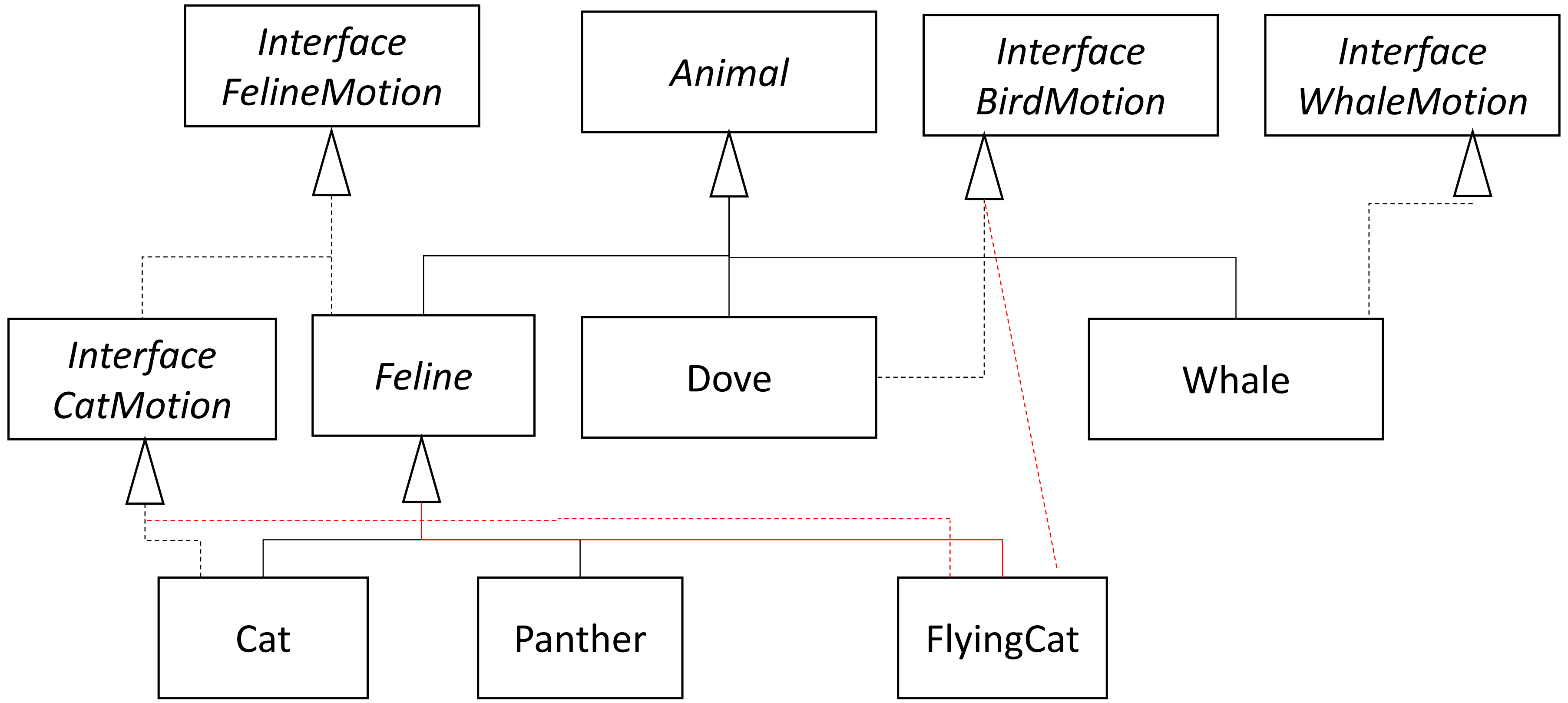
```
class FlyingCat implements
BirdMotion, CatMotion {

    …

}
```

✓

```
class FlyingCat extends
Feline implements
BirdMotion, CatMotion {

    …

}
```

✓

Example: Flying Cat in the Magic Kingdom



Questions

- Interfaces
  - Model common behaviors
  - Have only abstract methods
    - Since Java 8, can have default methods and static methods (virtual/abstract functions/methods with default implementations)
  - Are data types
  - Can be extended
  - Must be implemented
  - The “Animal Kingdom Enhanced” in the “Sample Programs” repo

What an object can do?

- The class hierarchy presents a problem
  - What data type are we dealing with?
- As a programmer how do we cope with it?
  - Use appropriate data types by design (preferred)

    ```
    void flyAll(ArrayList<BirdMotion> flyingAnimals) {
    …
    }
    ```
  - Check object type at runtime
    - Using instanceof
    - Using Class.isInstance()
    - Using Class.isAssignableFrom()

Operator instanceof

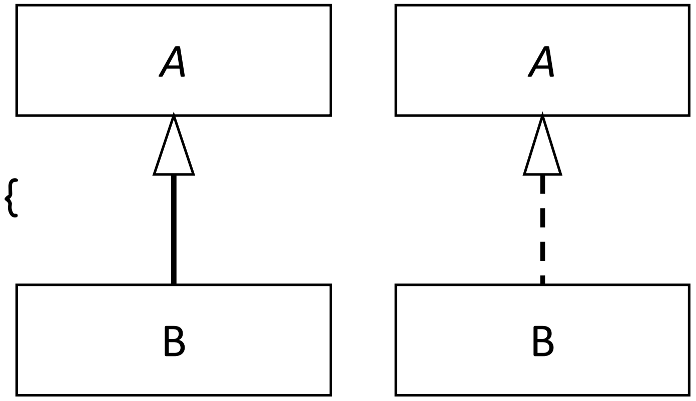
- Evaluates to true if the object is a given type; false otherwise
- Must know at compilation time the data type whether the object is instance of

```
void move(Animal animal) {
    if (animal instanceof Cat) {
        …
    }
}
```

Method Class.isInstance()

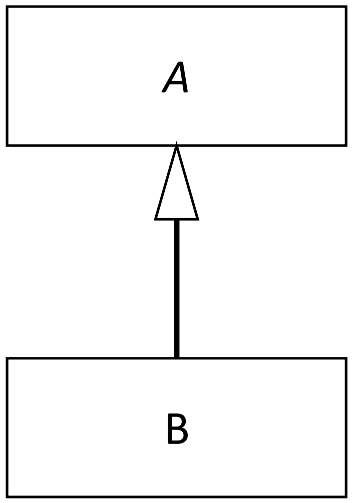
- Evaluates to true if the object is the data type of another object; false otherwise
  - A a = new A(); B b = new B(); a.getClass().isInstance(b)
  - is b an instance of A? True if any of the two scenarios in the graph
- Does not need to know at compilation time the data type whether the object is instance of

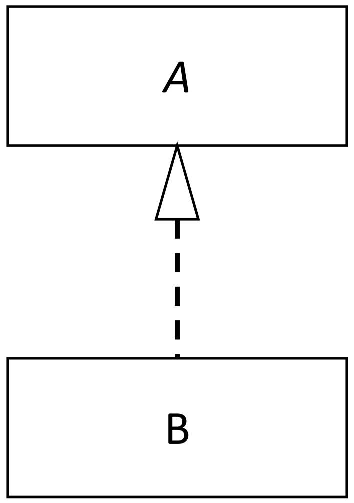
```
void move(Animal animal) {
        Cat garfield = new Cat();
        if (animal.getClass().isInstance(garfield)) {
                …
        }
}
```



Method Class.isAssignableFrom()

- A.isAssignableFrom(B)
  - where B is a class
  - Returns true if any of these two scenarios





Questions

- What is the object’s data type?
    - instanceof
    - Class.isInstance
    - Class.isAssignableFrom
- The “Vehicles” in the “Sample Programs“ repo

Nested Class

- Inner class (Non-static nested class)
  - Discussed in last class
- Static nested class
  - Discussed in last class
- Local class
- Anonymous class
  - Functional interface and Lamba expression

Local Class

- Classes defined within a block
  - What between a pair of balanced braces ({ … })
  - A block can be used anywhere a single statement is allowed.

```
class OuterClass {
    …

    { …
       class NestedClass {   …   }
       …
    }
    ….
}
```

Local Class: Characteristics

- Local classes are similar to inner classes
  - A local class has access to the members of its enclosing class.
- In addition, a local class has access to final or effectively final local variables
  - Final variables: e.g., final int a;
  - Effectively final, e.g., int a = 1; but variable "a" never changes after initialization
- It can access the method's parameters
- However,
  - cannot declare static initializers or member interfaces in a local class.
  - can only have static members only when they are constants (final static …)

Anonymous Class

- Essentially, a local class without a name
- Created by declaring and instantiating a class at the same time
- Use it when need a local class only once

```
class OuterClass {
    ...

    {  ...
       ParentClass a = new ParentClass() { ...}
    }
       ...
}
```

Anonymous Classes are Local Classes

- It has access to the members of its enclosing class.
- In addition, it has access to final or effectively final local variables
  - Final variables: e.g., final int a;
  - Effectively final, e.g., int a = 1; but variable “a” never changes after initialization
- It can access the method’s parameters
- However,
  - cannot declare static initializers or member interfaces in a local class.
  - can only have static members only when they are constants (final static …)

Nested Classes and Java API

- Many Java API methods have interface parameters
  - Comparators, Predicate, …
  - Commonly used with nested classes (most often, anonymous classes)
- Examples:
  - java.util.Arrays: binarySearch(T[] a, T key, Comparator<? super T> c)
  - java.util.ArrayList: sort(Comparator<? super E> c)
  - java.util.ArrayList: removeIf(Predicate<? super E> filter)
  - java.util.Collections: binarySearch(List<? extends T> list, T key, Comparator<? super T> c)
  - java.util.Collections: removeIf(Predicate<? super E> filter)

Functional Interface

- Any interface that contains only one abstract method
  - Since Java 8, a functional interface may contain one or more default methods or static methods

Use Functional Interface

- In your own design, sometime functional interface is better choice
- More often, you use functional interfaces because some Java API methods require them
  - Examples:
    - https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Functional Interface and Anonymous Class

- You can declare and instantiate a local class, or more often an anonymous class
- Example

```
ArrayList<Person> personList = new ArrayList<Person>();
Arrays.sort(personList, new Comparator<Person> {
   @Override
    public int compare(Person lhs, Person rhs) {
       // buggy (what if rhs is null?)
       return lhs.getName().compareTo(rhs.getName());
    }
}
```

Lambda Expression

- A simple way to declare and instantiate a class

```
ArrayList<Person> personList = new ArrayList<Person>();
Arrays.sort(personList, new Comparator<Person> {
  @Override
   public int compare(Person lhs, Person rhs) {
     // buggy (what if rhs is null?)
     return lhs.getName().compareTo(rhs.getName());
   }
}
```

```
ArrayList<Person> personList = new ArrayList<Person>();
Arrays.sort(personList, (lhs, rhs) -> lhs.getName().compareTo(rhs.getName())}
```

Questions

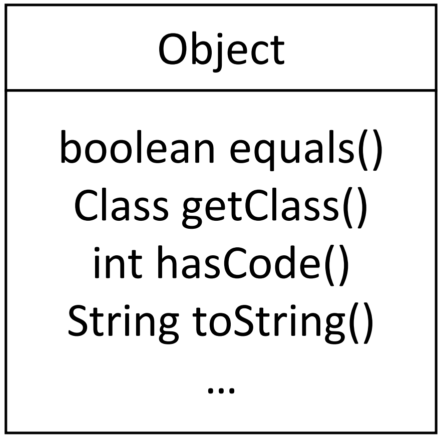
- Nested classes
- Nested classes in Java API
- Lambda expression
- The “Nested Class Example” in the “Sample Programs” repo

Inheritance, Generic Programming, and Java API

- Commonly seen these in Java API
  - <? extends E>
    - Any data type that is of data type E or a sub-type of E
  - <? super E>
    - Any data type that is of data type E or a super-type of E
- Discuss more in the future

Recall: The Object Super Class

- Java has a class called **Object**, like
- All classes are subclass of **Object** in Java



Questions

- A few items commonly seen in Java API
- The Java Object class

Assignment

- Practice Assignment
- CodeLab