COMP 110/L Lecture 21

Mahdi Ebrahimi

Slides adapted from Dr. Kyle Dewey

Outline

- `this`
- `instanceof`
- `Casting`
- `equals()`
- `protected`
- `interface`

this

`this`

Refers to whatever instance the given instance method is called on.

`this`

Refers to whatever instance the given instance method is called on.

```
public class Foo {
  public Foo returnMyself() {
    return this;
  }
}
```

Example:
`ThisExample.java`

Name Clashes

`this` can be used to refer to instance variables which have the same name as normal variables

Name Clashes

`this` can be used to refer to instance variables which have the same name as normal variables

```
public class NameClash {
  private int x;
  public NameClash(int x) {
    this.x = x;
  }
}
```

Example:
`NameClash.java`

`instanceof`

`instanceof`

Returns a boolean indicating if a given instance was made from or inherited from a given class

instanceof

Returns a boolean indicating if a given instance was made from or inherited from a given class

```
public class InstanceOf {
  public static void main(String[] a) {
    InstanceOf i = new InstanceOf();
    if (i instanceof InstanceOf &&
        i instanceof Object) {
      // code reaches this point
    }
  }
}
```

Example:
`InstanceOfExample.java`

Casting

Casting

Converts a value of one type into another.
Not always possible to perform.

Casting

Converts a value of one type into another.
Not always possible to perform.

```
int myInt0 = 16.0;
```

Casting

Converts a value of one type into another.
Not always possible to perform.

```
int myInt0 = 16.0;
```

Does not compile

Casting

Converts a value of one type into another.
Not always possible to perform.

```
int myInt0 = 16.0;

int myInt1 = (int)16.0;
```

Casting

Converts a value of one type into another.
Not always possible to perform.

```
int myInt0 = 16.0;

int myInt1 = (int)16.0;
```

`myInt1` holds 16

Casting

Converts a value of one type into another.
Not always possible to perform.

```
int myInt0 = 16.0;

int myInt1 = (int)16.0;


int myInt2 = (int)16.5;
```

Casting

Converts a value of one type into another.
Not always possible to perform.

```
int myInt0 = 16.0;

int myInt1 = (int)16.0;

int myInt2 = (int)16.5;
```

`myInt2` holds 16

Casting

Converts a value of one type into another.

Not always possible to perform.

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
...
Foo f = new Foo();
```

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
...
Foo f = new Foo();
Object o = f;
```

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
...
Foo f = new Foo();
Object o = f;
Foo g = o;
```

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
...
Foo f = new Foo();
Object o = f;
~~Foo g = o;~~
```

Does not compile

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
...
Foo f = new Foo();
Object o = f;
Foo g = (Foo)o;
```

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
...
Foo f = new Foo();
Object o = f;
Foo g = (Foo)o;
```

Compiles and runs ok

Casting

Converts a value of one type into another.

Not always possible to perform.

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
public class Bar { ... }
...
```

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
public class Bar { ... }
...
Foo f = new Foo();
Bar b = new Bar();
```

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
public class Bar { ... }
...
Foo f = new Foo();
Bar b = new Bar();
f = b;
```

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
public class Bar { ... }
...
Foo f = new Foo();
Bar b = new Bar();
~~f = b;~~
```

Does not compile

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
public class Bar { ... }
...
Foo f = new Foo();
Bar b = new Bar();
f = (Foo)b;
```

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
public class Bar { ... }
...
Foo f = new Foo();
Bar b = new Bar();
f = (Foo)b;
```

Compiles, but doesn't run correctly
(gives a `ClassCastException`)

`equals()`

`equals()`

Used to determine if two arbitrary objects are equal. Defined in `Object`.

`equals()`

Used to determine if two arbitrary objects are equal. Defined in `Object`.

```
“foo”.equals(“foo”)
```

Returns `true`

`equals()`

Used to determine if two arbitrary objects are equal. Defined in `Object`.

```
“foo”.equals(“foo”)
```

Returns `true`

```
“foo”.equals(“bar”)
```

`equals()`

Used to determine if two arbitrary objects are equal. Defined in `Object`.

```
“foo”.equals(“foo”)
```

Returns `true`

```
“foo”.equals(“bar”)
```

Returns `false`

`equals()` **vs.** `==`

- With `equals()`, we test *object equality*, AKA *deep equality*
  - Look at the inside of the object
- With `==`, we test *reference equality*, AKA *shallow equality*
  - Return `true` if two references refer to the exact same object

Example:
`StringEquals.java`

Defining Your Own `equals()`

- Usual pattern: see if the given thing is an instance of my class
  - If `true`, cast it to the class, and do some deep comparisons
  - If `false`, return `false`
- Anything is possible

Example: `CustomEquals.java`

`protected`

`protected`

Somewhere between `private` and `public`.

Like `private`, but subclasses can access it.

`protected`

Somewhere between `private` and `public`.

Like `private`, but subclasses can access it.

```
public class HasPrivate {
  private int x;
}
```

`protected`

Somewhere between `private` and `public`.

Like `private`, but subclasses can access it.

```
public class HasPrivate {
  private int x;
}
public class Sub extends HasPrivate {
  ...x...
}
```

`protected`

Somewhere between `private` and `public`.

Like `private`, but subclasses can access it.

```
public class HasPrivate {
  private int x;
}
```

~~`public class Sub extends HasPrivate {`~~

~~`  ...x...`~~

~~`}`~~

Not permitted - `x` is `private` in `HasPrivate`

protected

Somewhere between `private` and `public`.

Like `private`, but subclasses can access it.

```
public class HasPrivate {
  private int x;
}
public class Sub extends HasPrivate {
  ...x...
}
```

```
public class HasProt {
  protected int x;
}
public class Sub extends HasProt {
  ...x...
}
```

`protected`

Somewhere between `private` and `public`.

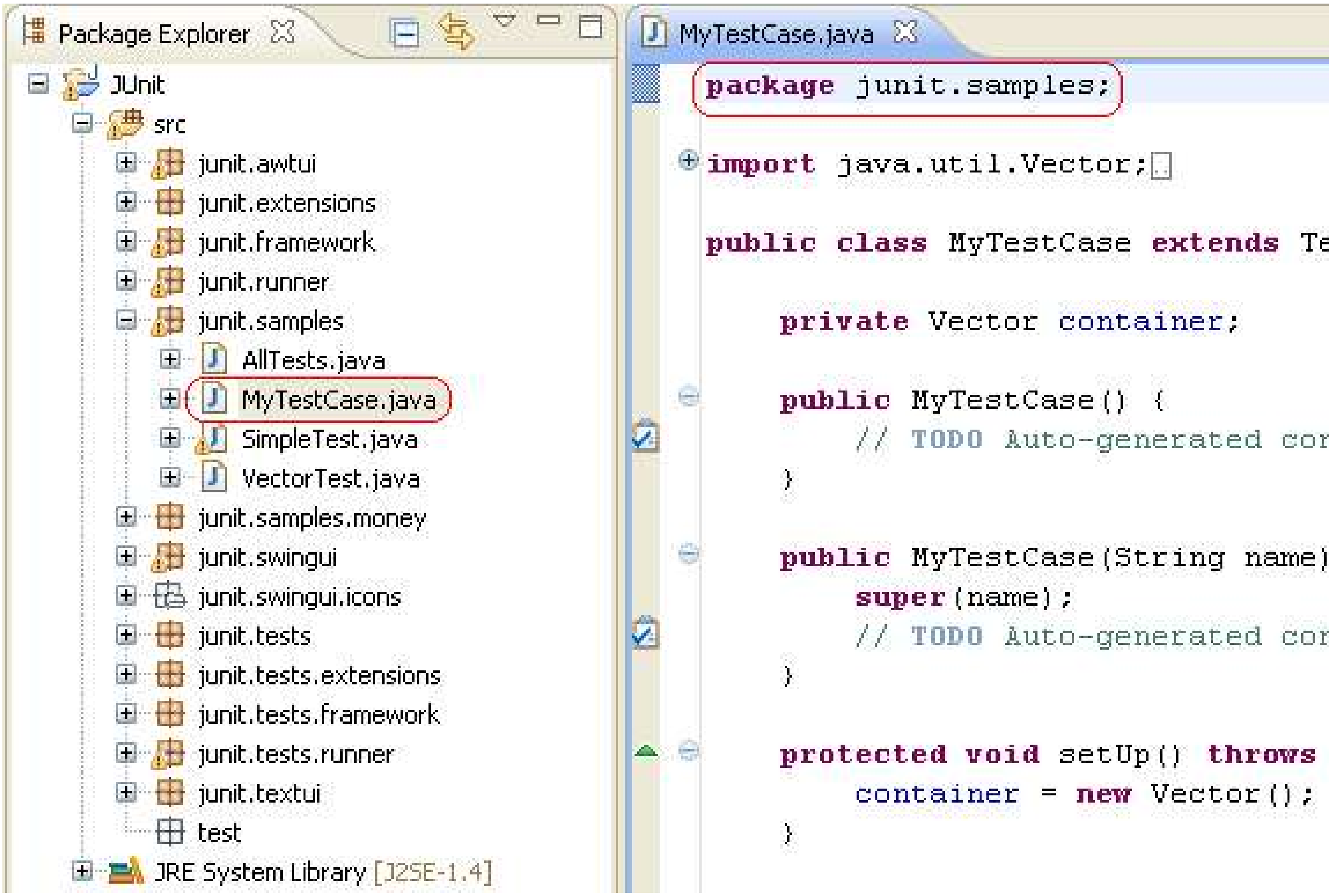
Like `private`, but subclasses can access it.

```
public class HasPrivate {
  private int x;
}
public class Sub extends HasPrivate {
  ...x...
}
```

```
public class HasProt {
  protected int x;
}
public class Sub extends HasProt {
  ...x...
}
```

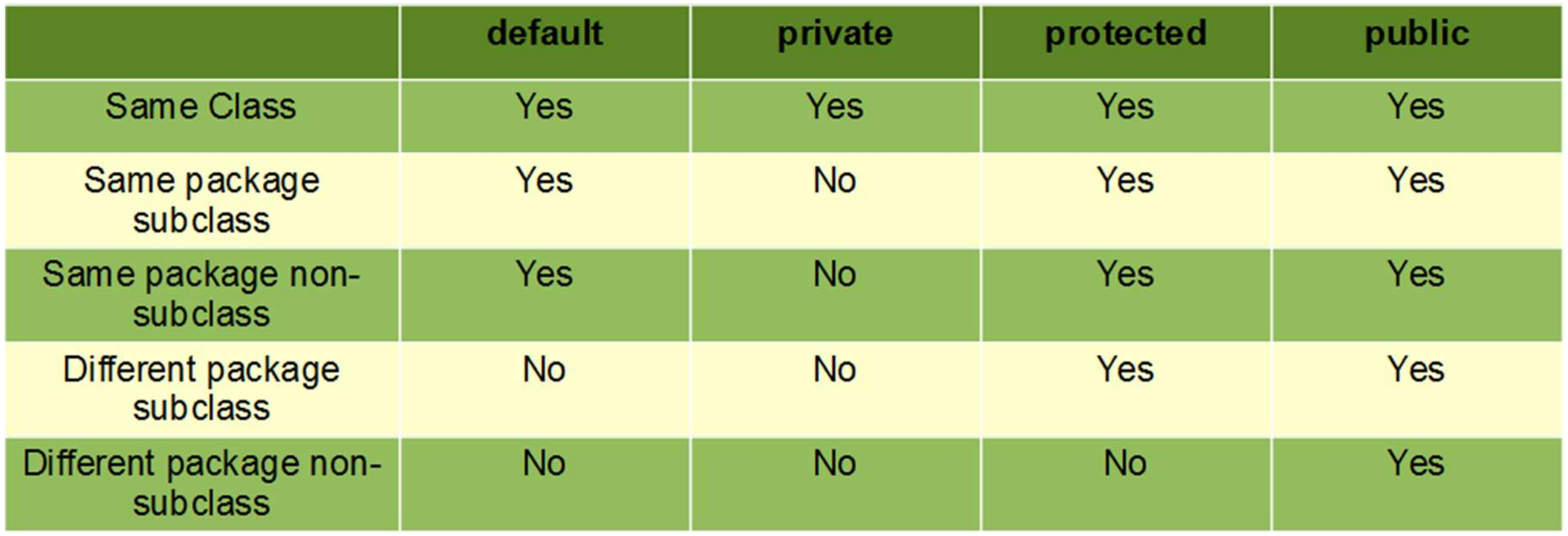
OK: `Sub` is a subclass of `HasProt`

A package in Java is used to group related classes. Think of it as **a folder in a file directory**. We use packages to avoid name conflicts, and to write a better maintainable code.



```
package junit.samples;

import java.util.Vector;

public class MyTestCase extends Te

    private Vector container;

    public MyTestCase() {
        // TODO Auto-generated cor
    }

    public MyTestCase(String name)
        super(name);
        // TODO Auto-generated cor
    }

    protected void setUp() throws
        container = new Vector();
    }
```

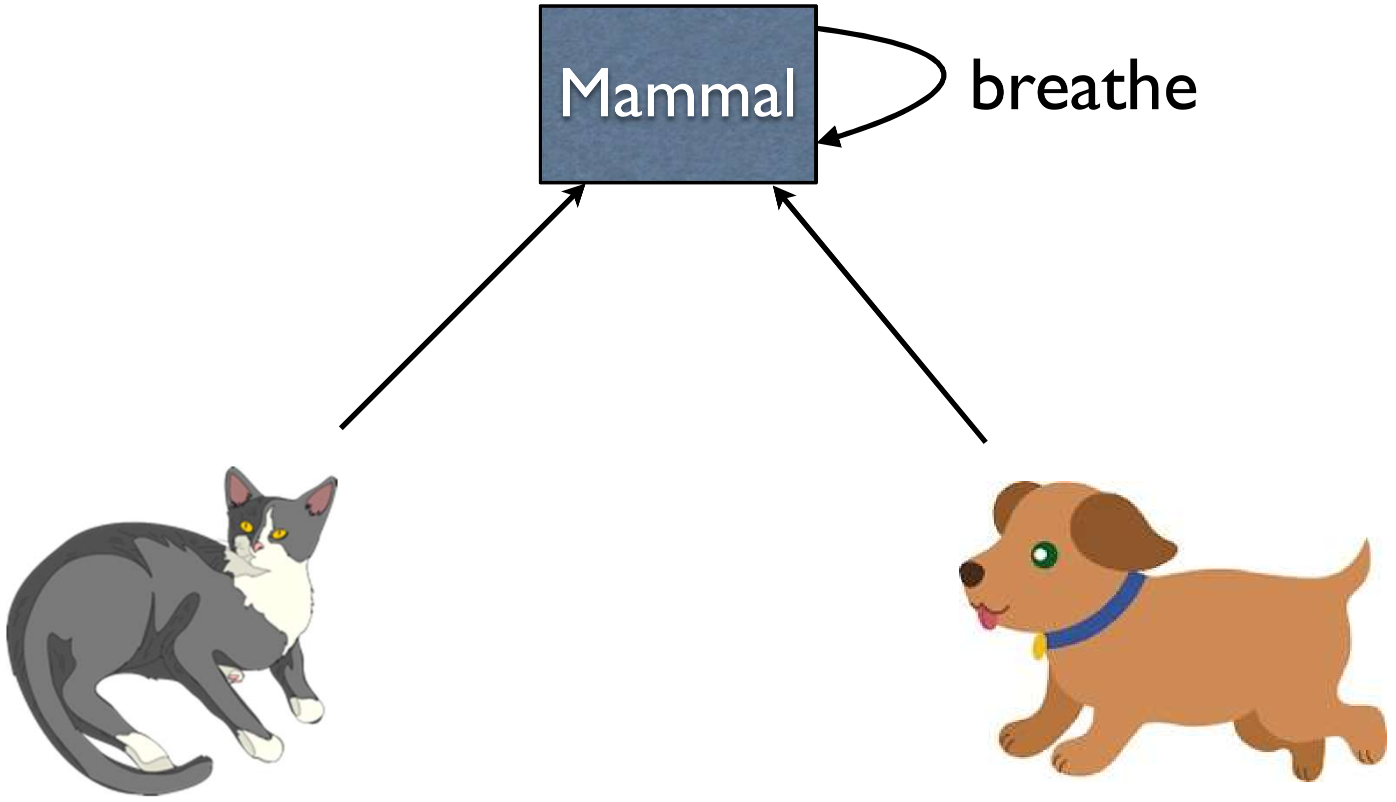
https://www.w3schools.com/java/java_packages.asp

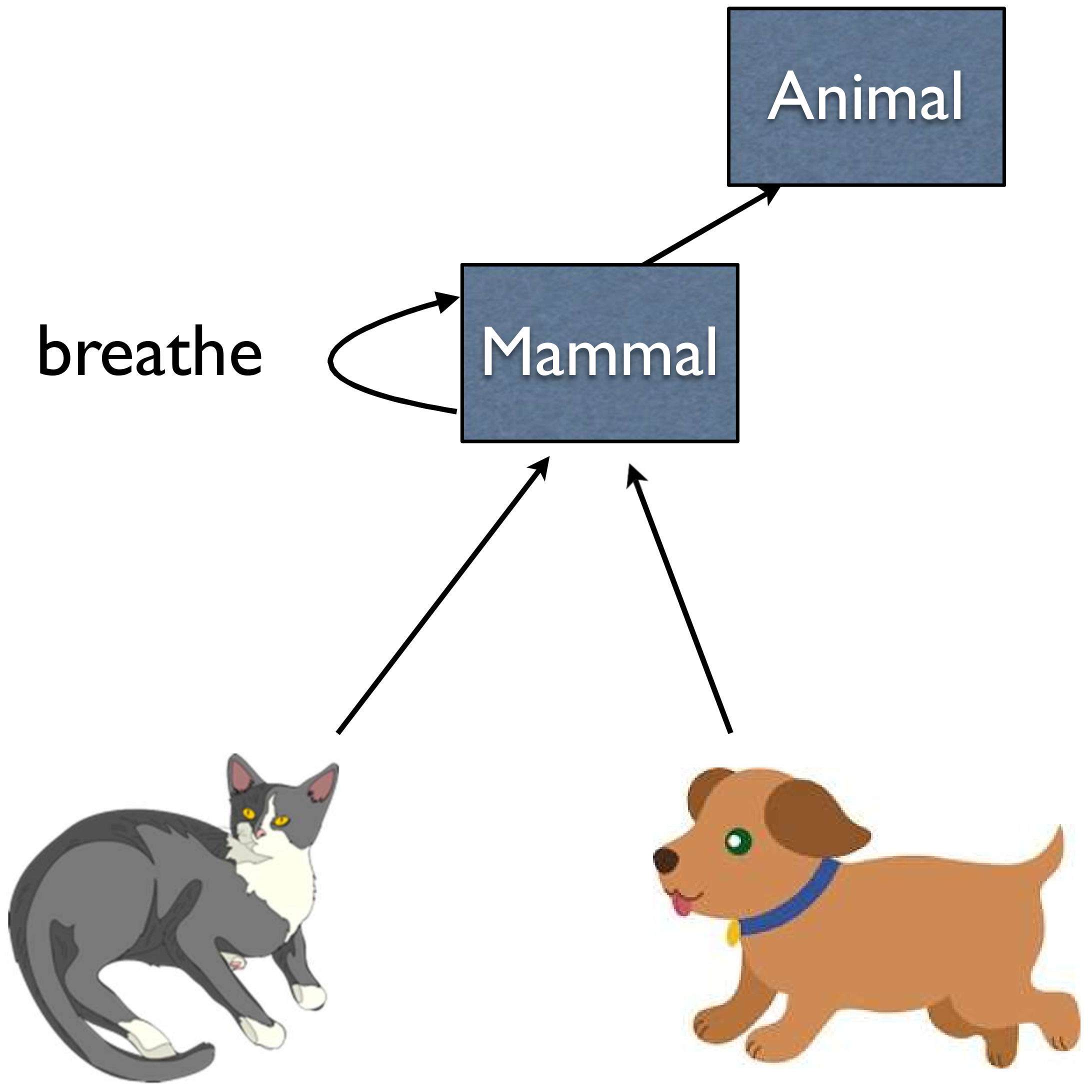


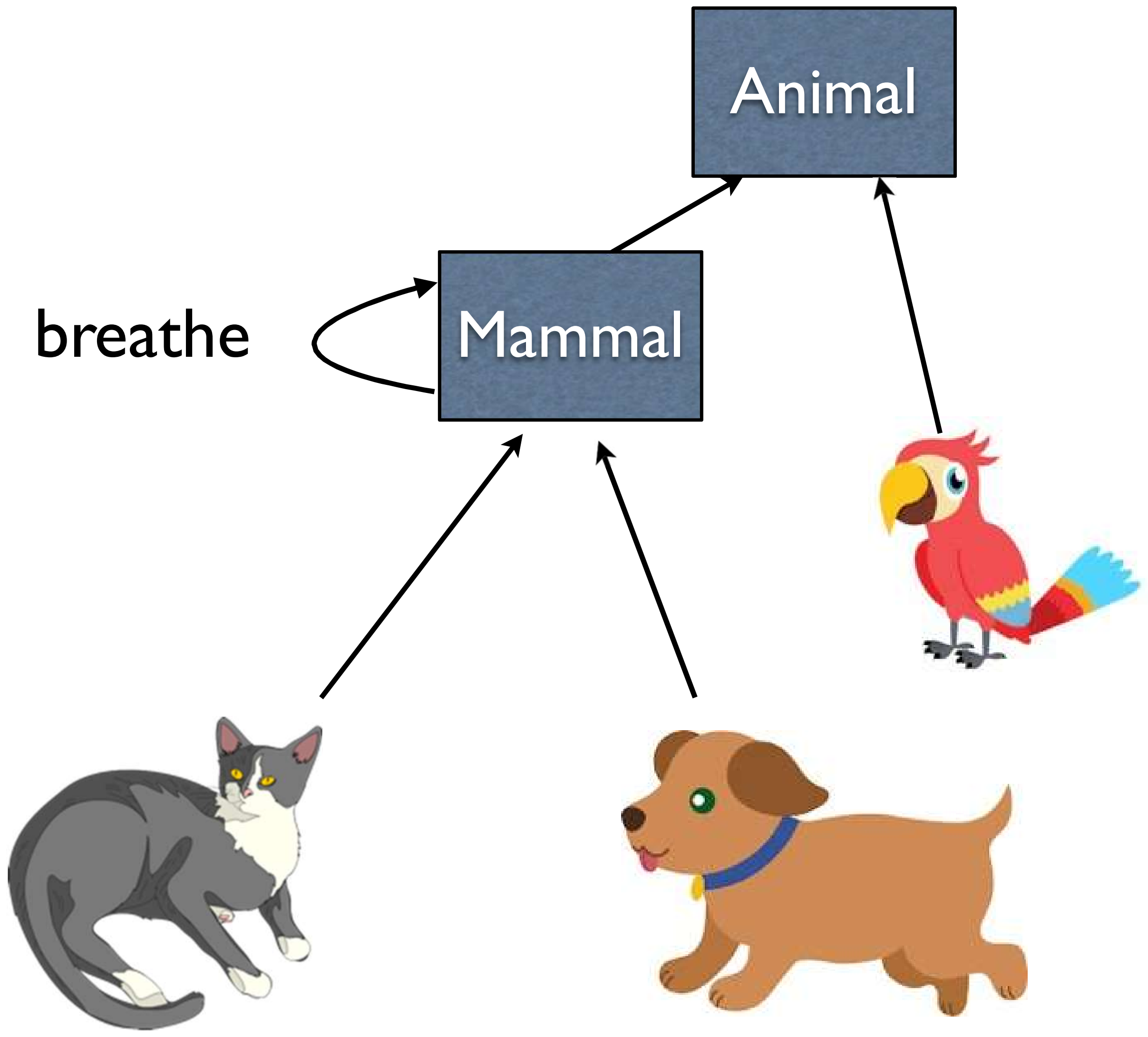
	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

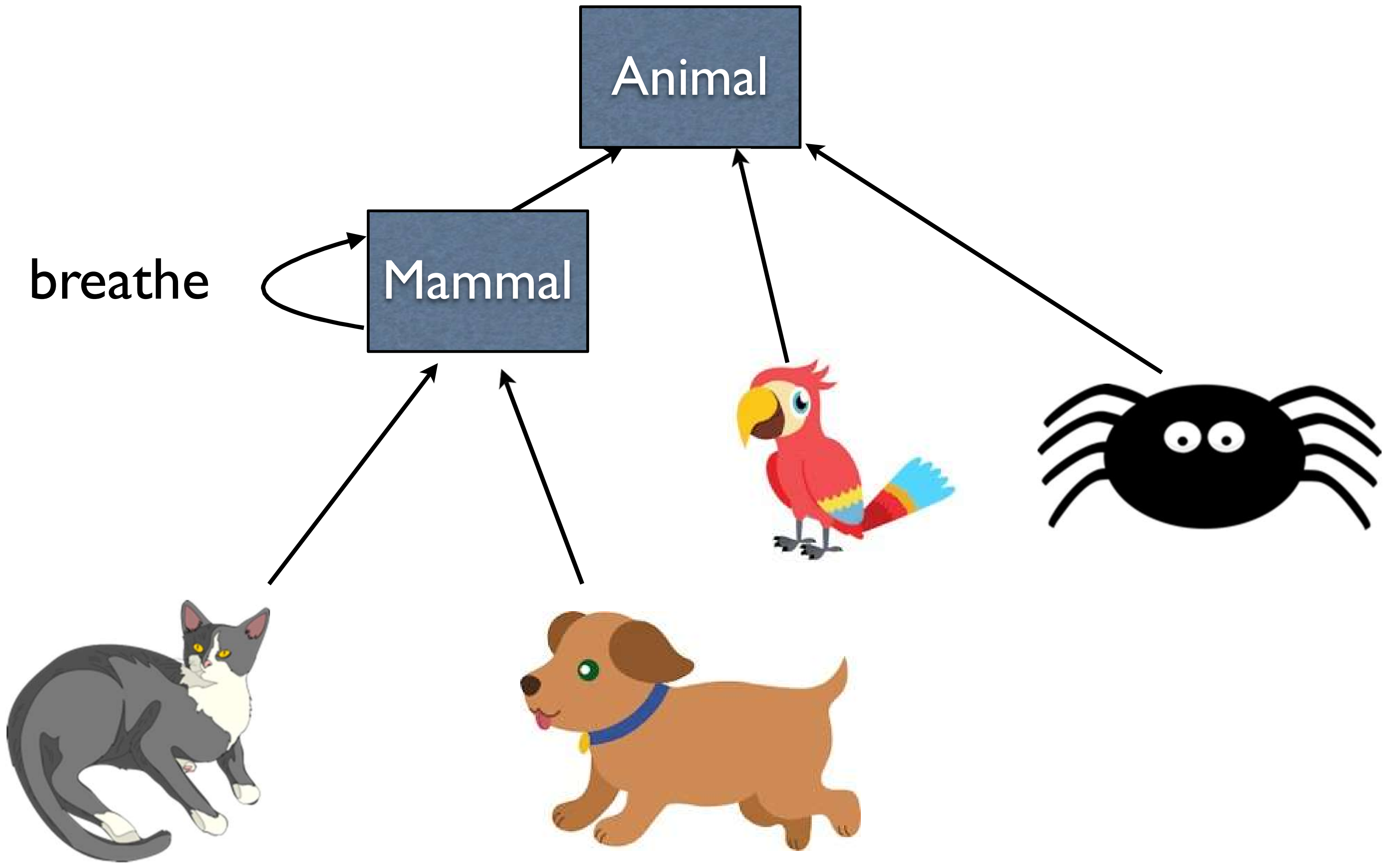
https://www.geeksforgeeks.org/access-modifiers-java/

interface


Mammal
breathe


Animal
Mammal
breathe


Animal
Mammal
breathe


Animal
Mammal
breathe

`interface`

- Like an abstract class with the following restrictions:
  - Cannot have constructors
  - Cannot have instance variables
- However, we can inherit from them anywhere, and we can inherit from multiple interfaces

Using `interfaces`

```
public interface CanBreathe {
  public void breathe();
}
```

Using `interfaces`

```
public interface CanBreathe {
  public void breathe();
}
```

```
public class Foo extends Bar
implements CanBreathe {
  public void breathe() { ... }
}
```

Using `interfaces`

```
public interface CanBreathe {
  public void breathe();
}
```

```
public class Foo extends Bar
implements CanBreathe {
  public void breathe() { ... }
}
```

```
public class Multi extends Alpha
implements Beta, Gamma, Delta { ... }
```

Example

- `Animal.java`
- `CanBreathe.java`
- `Mammal.java`
- `Dog.java`
- `Cat.java`
- `CanFly.java`
- `Parrot.java`
- `Bat.java`
- `Spider.java`
- `AnimalMain.java`