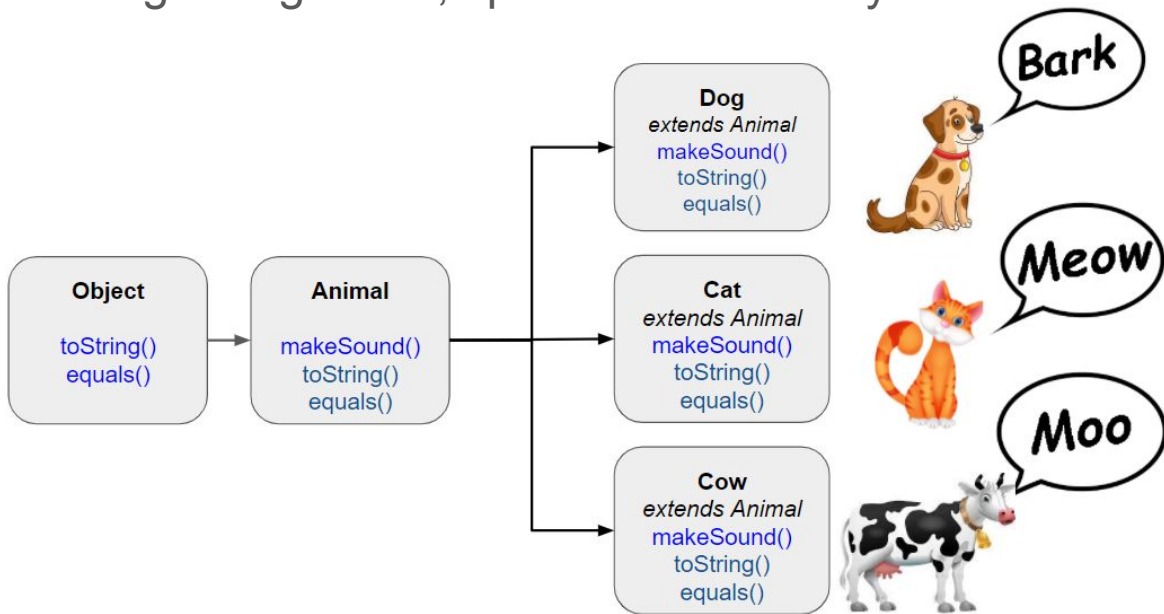


Pulse Survey is open

Polymorphism

Polymorphism is the ability for a subclass to be treated as its superclass, while still providing subclass specific responses. The subclass can be treated generically while still providing non-generic, specific functionality.

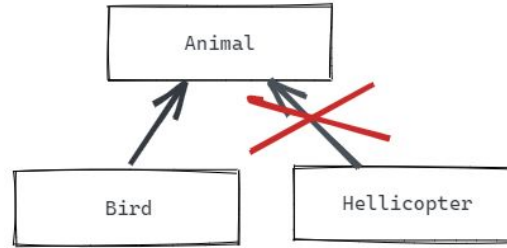


Object Context

An Object can have a relationship with another type in its hierarchy with an IS-A relationship.

A Bird IS-A Animal

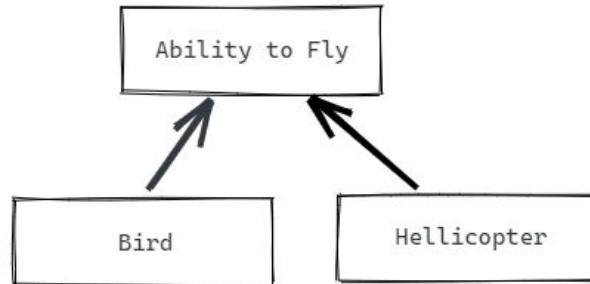
A Helicopter IS NOT A Animal



But we often want to group objects not by what they are, but what they can do, or by what abilities they have - a HAS-A relationship. This allows for grouping of unlike things with like abilities.

A Bird HAS-A ability to fly

A Helicopter HAS-A ability to fly



Interfaces

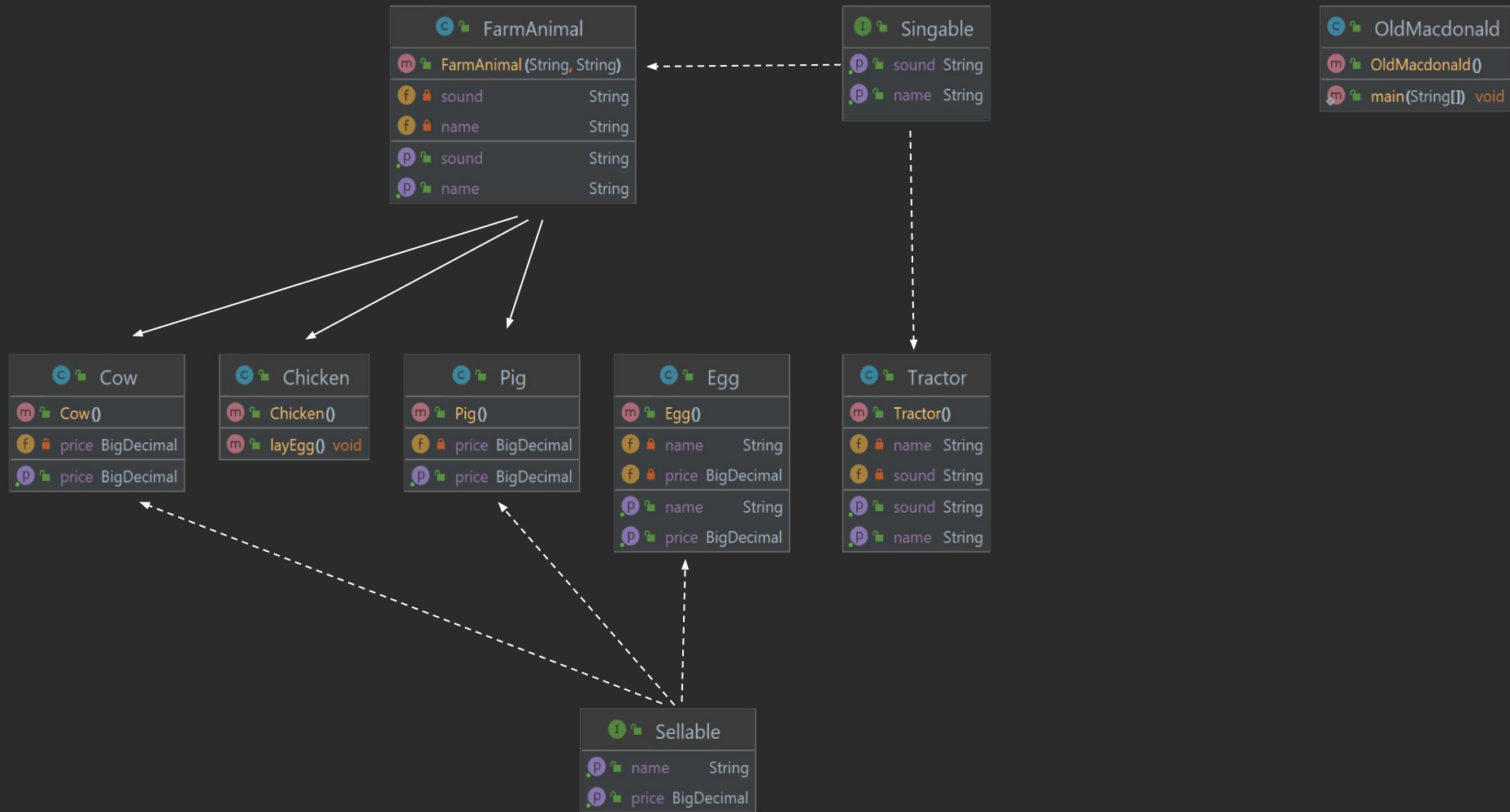
1. Defines what something can do or how it can be used, but not how it does it.
2. A contract that defines what methods an object must have to be of that type, but does not define how those methods will work.
3. Provides method signatures, but not implementation.
4. Transitive like inherited methods
 - a. If Class A implements interface B, then A “is-a” B (and so are A's children)
5. A single class may have multiple interfaces, allowing to be part of multiple groupings.

```
public interface Drivable {  
  
    void turn(String direction);  
    boolean accelerate();  
    boolean decelerate();  
  
}
```

Polymorphism using Interfaces

The interfaces create a data type to which the object can be cast. This allows objects with the same interfaces to be grouped generically, while still providing their specific response when a method is invoked.

```
List<Drivable> drivables = new ArrayList<Drivable>();  
drivables.add( new Car() );  
drivables.add( new Bicycle() );  
drivables.add( new HotAirBalloon() );  
  
for (Drivable d : drivables) {  
  
    d.turn( "Left" );  
    d.accelerate();  
  
}
```



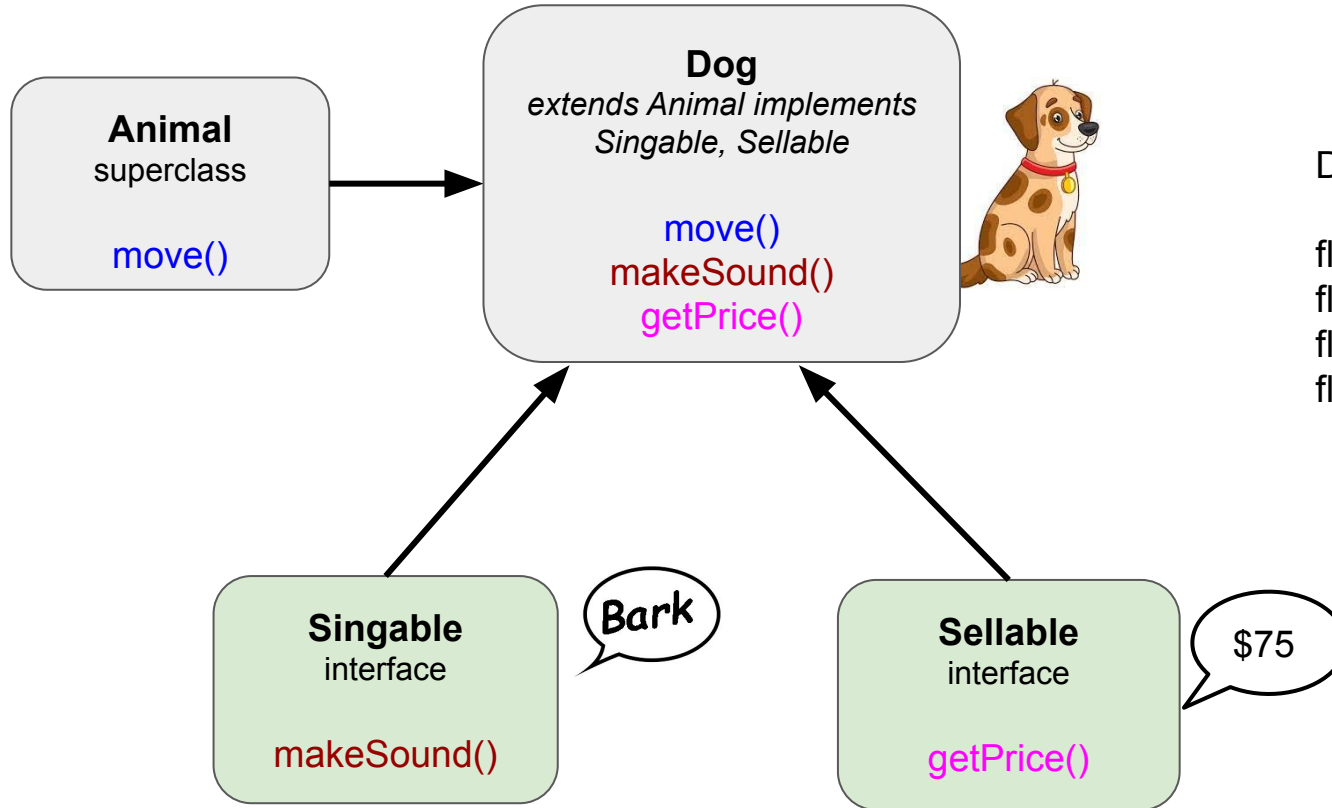
Managing Inheritance

Module 1: 13

Today's Objectives

1. instanceof Operator
2. Final Methods and Classes
3. Abstract Methods and Classes
4. Protected and Default Access Modifiers

Object instanceof



`Dog fluffer = new Dog();`

`fluffer instanceof Animal`
`fluffer instanceof Dog`
`fluffer instanceof Singable`
`fluffer instanceof Sellable`

An Object is an instanceof a data type if it can be safely cast to that data type.

instanceof operator

Since *downcasting* can only be done if the object is already internally the type it is being cast to, there is a boolean operator, ***instanceof***, that can check if the object can be downcast to the subclass type.

object ***instanceof*** class

```
public void convert(Calculator calculator) {  
  
    if (calculator instanceof ScientificCalculator) {  
        ScientificCalculator = (Scientific) calculator;  
    }  
  
}
```

instanceof should be used when a class is being downcast to a subclass, and it is not known what type the object is internally.

Final Methods and Classes

The **final** modifier can be used with methods and classes to control how they are used.

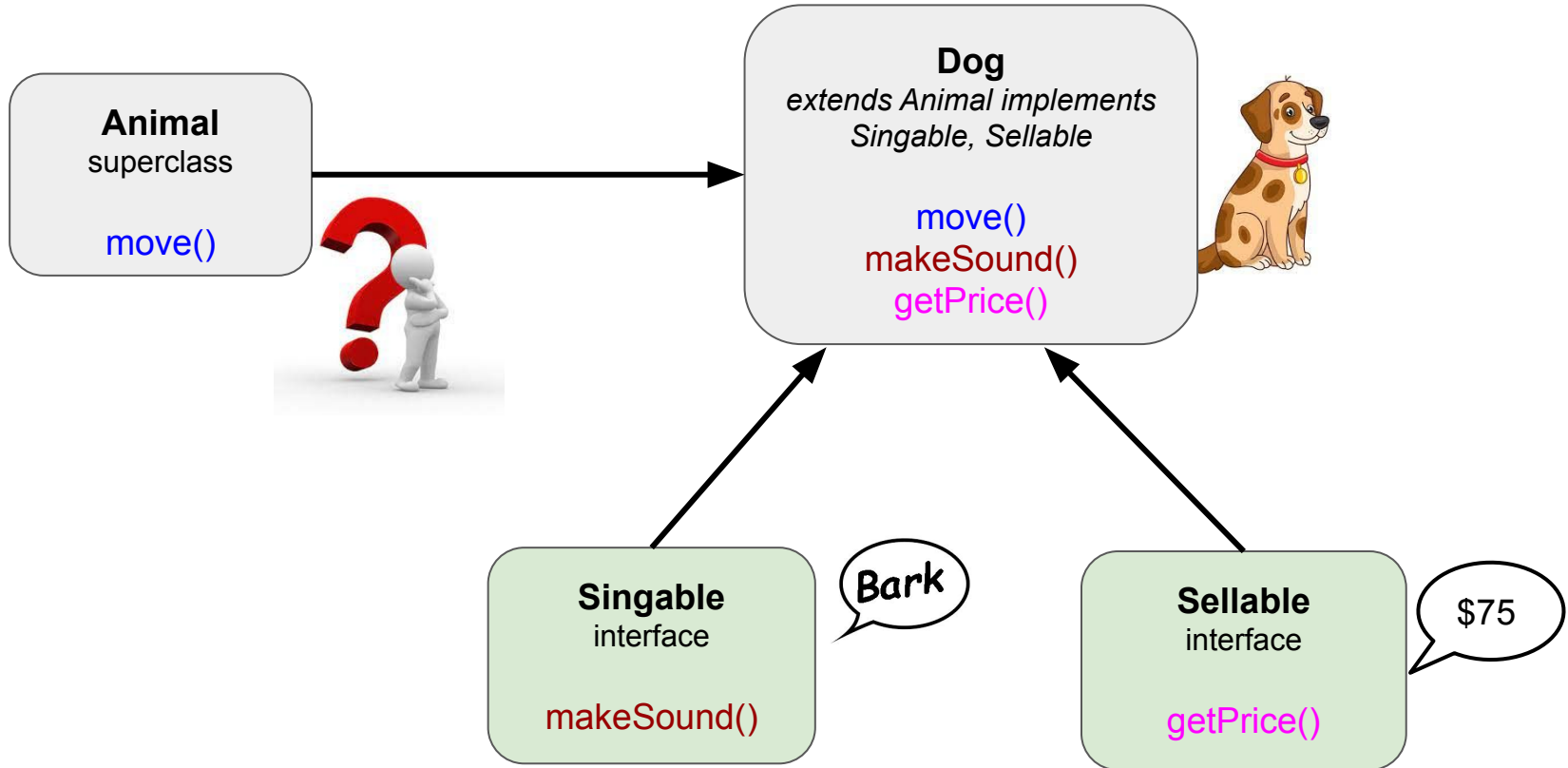
A final method cannot be Overridden.

```
public final String myMethod() { }
```

A final Class cannot have subclasses.

```
public final class myClass { }
```

Some classes don't make sense to be able to instantiate...



Abstract Class

An abstract class cannot be instantiated and exists solely for the purpose of inheritance and polymorphism.

Like a combination of an interface and a superclass.

1. Can extend it like a superclass
2. Can inherit implementation from it like a superclass
3. Can provide method signatures that must be implemented like an interface
4. A class can only extend either 1 abstract class or 1 superclass, but may implement multiple interfaces.

Abstract Method

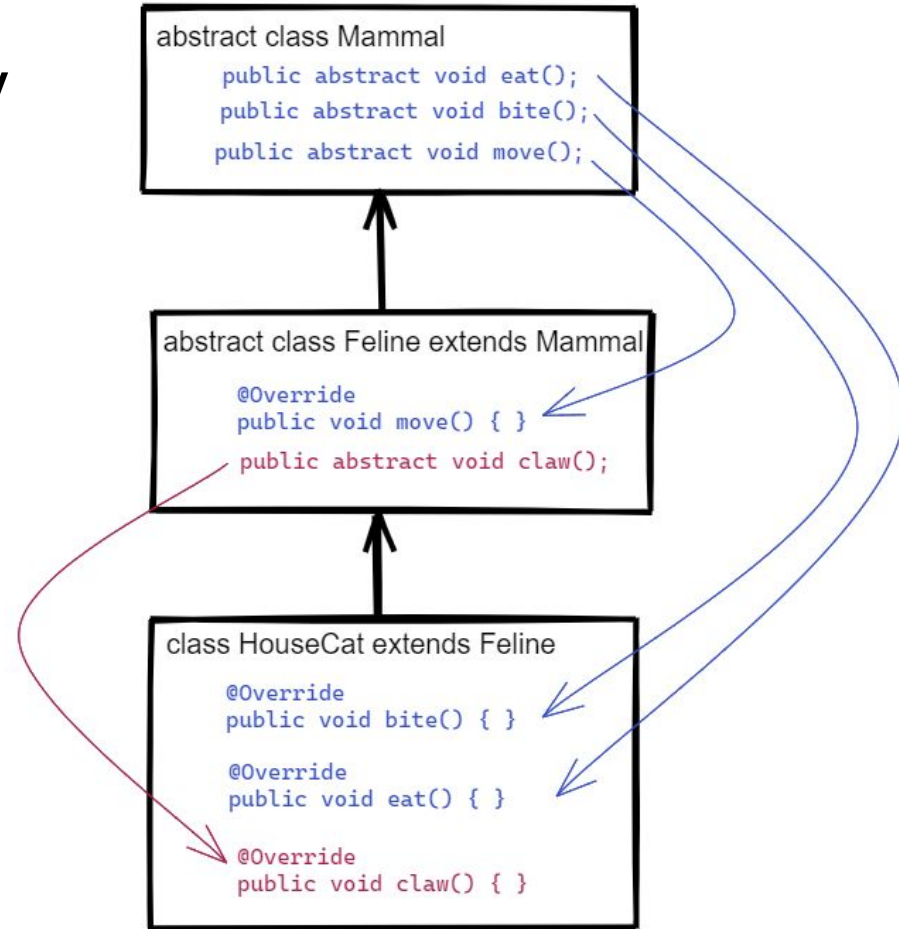
A method signature with the **abstract** modifier. Can only exist in an abstract class, and must be overridden in any implementation class that extends the abstract class.

```
public abstract int doScience(int x, int y)
```

Abstract Method Transitivity

If an abstract class extends another abstract class, then it can choose to implement the abstract method or do nothing and pass it on to the class that extends it.

All abstract methods must have implementation by the time extended by the first concrete implementation class.



Reasons to use an Abstract Class




























1. To prevent a superclass from being instantiated.
 - a. For example: Having a Generic “Feline” as an object may not make sense, so by making Feline abstract it forces the user of the class to instantiate the more concrete HouseCat or Lion objects that are subclasses of Feline.
2. When you need to have the ability to inherit functionality from a superclass and force a subclass to implement subclass specific methods.
 - a. ***Abstract classes should only represent a IS-A relationship*** (a Car IS-A Vehicle) and **never** a HAS-A relationship (a Car HAS-A Drivable)
 - b. This same need can also be accomplished by using a combination of a superclass and interfaces.

Superclass vs Interface vs Abstract

An **interface** provides method signatures without implementation that creates a contract of what must have a subclass specific implementation. *Can represent either an HAS-A or IS-A relationship.*

A **superclass** provides default implementation that can be inherited by a subclass, so it can guarantee a default implementation, but not a subclass specific one. *Can only represent an IS-A relationship.*

An **abstract class** allows for default implementation to be inherited and/or to provide method signatures for methods that must have a subclass specific implementation. *Can only represent an IS-A relationship.*

	Passes on implementation	Forces Override	Can be instantiated	IS-A	HAS-A	extends	implements	Subclass can have multiple	Defines a Data Type
Superclass									
Interface									
Abstract class									

Common Interview Question

What is the difference between an interface and an abstract class?

Protected and Default Access Modifiers

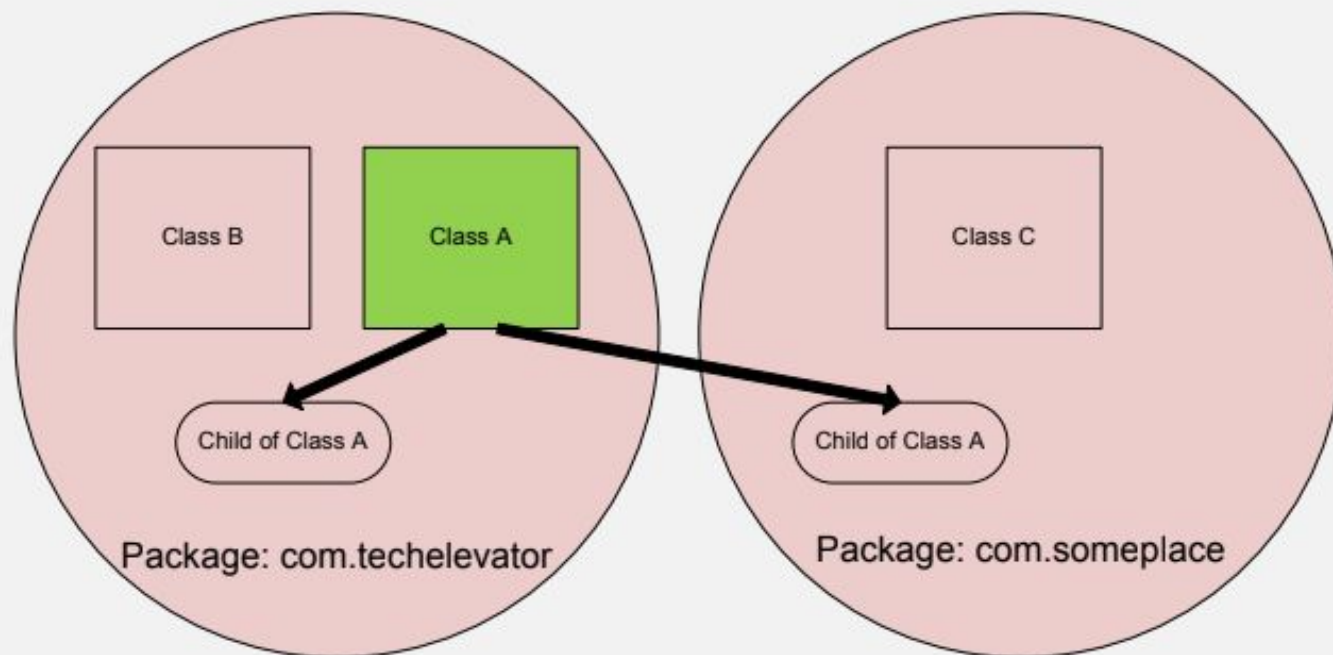
- **Private** - accessible only in the class
 - can be applied to methods and member variables
- **Protected** - accessible in the class and in any subclasses in the inheritance tree.
 - can be applied to methods and member variables
 - In Java, Protected is also available to any class in the same package, but this use is discouraged.
- **Default (no access modifier)** - accessible to any class or subclass in the same package.
 - can be applied to methods and member variables
- **Public** - accessible everywhere
 - can be applied to methods, member variables, classes, and interfaces)

Private in Class A

Key

Can Access

Cannot Access



Can be applied to:

- Methods
- Constructors
- Properties
- Inner Classes

Cannot be applied to:

- Classes
- Interfaces

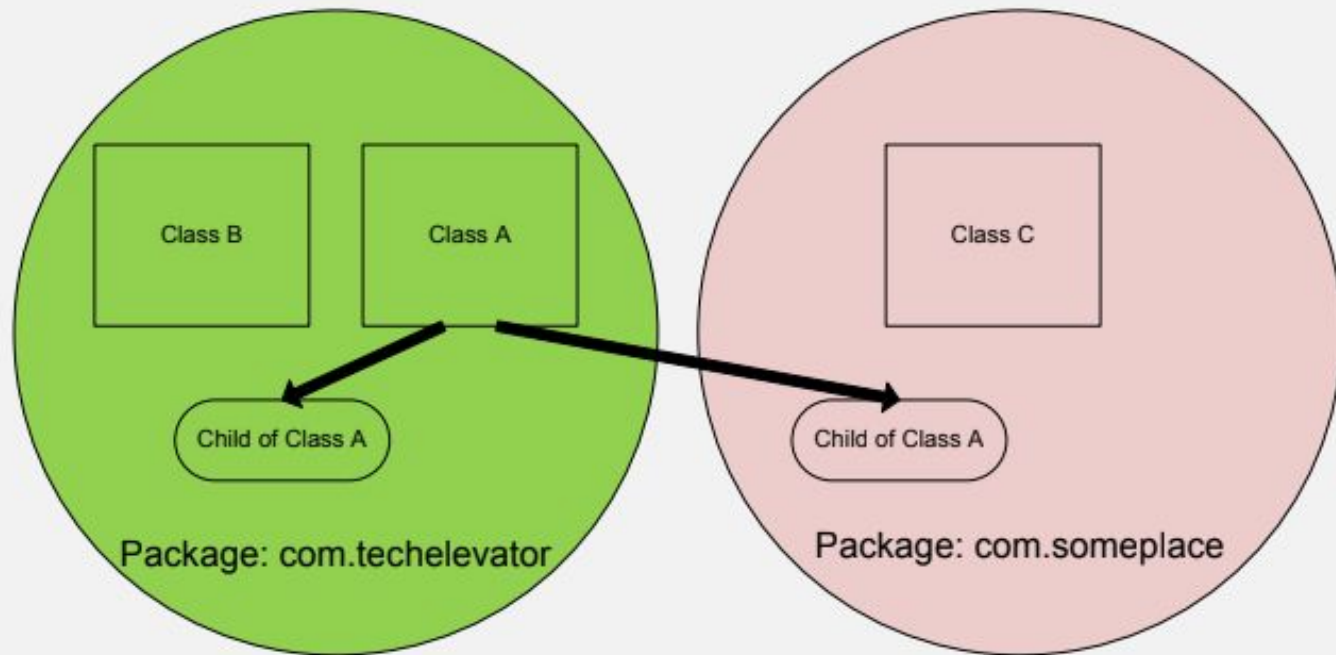
Private things can be accessed by: Only other things in the declaring class.

Default in Class A

Key

Can Access

Cannot Access

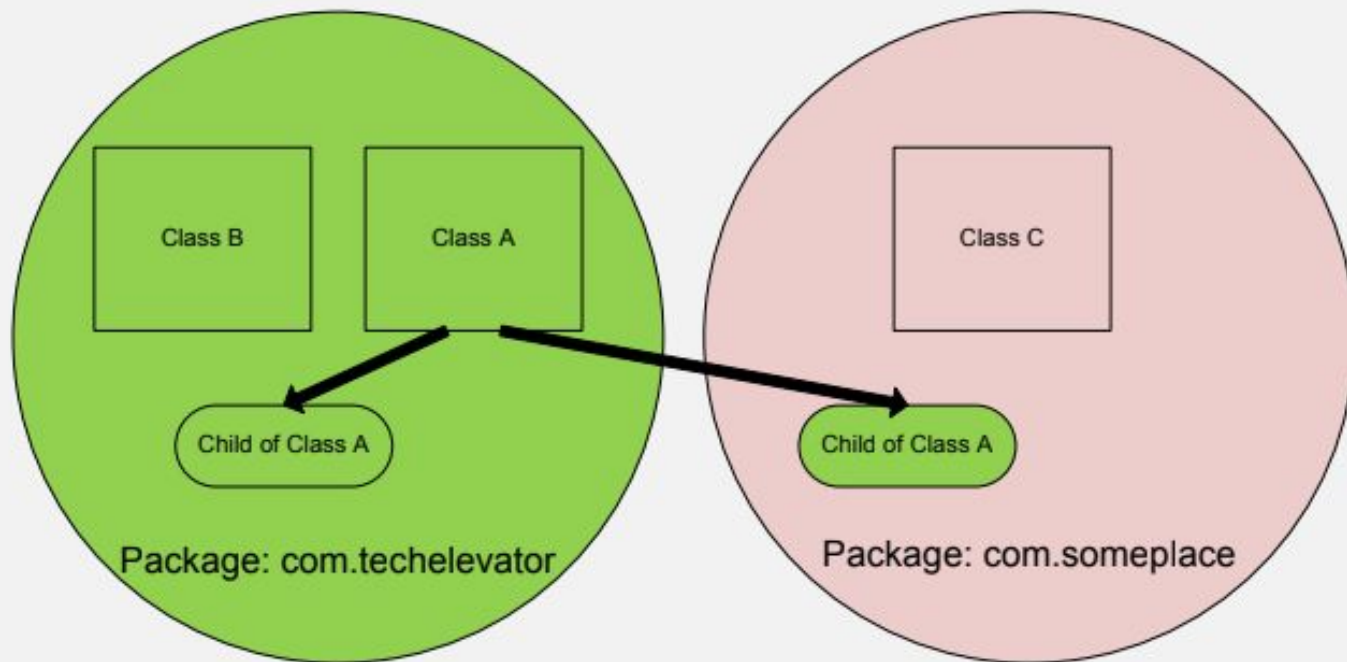


Can be applied to:
Methods, Constructors
Properties, Classes, Inner
Classes, and Interfaces

Default is what is
applied if no accessor
is explicitly given

**Default things can be
accessed by:** any class in the
same package

Protected in Class A



Can be applied to:

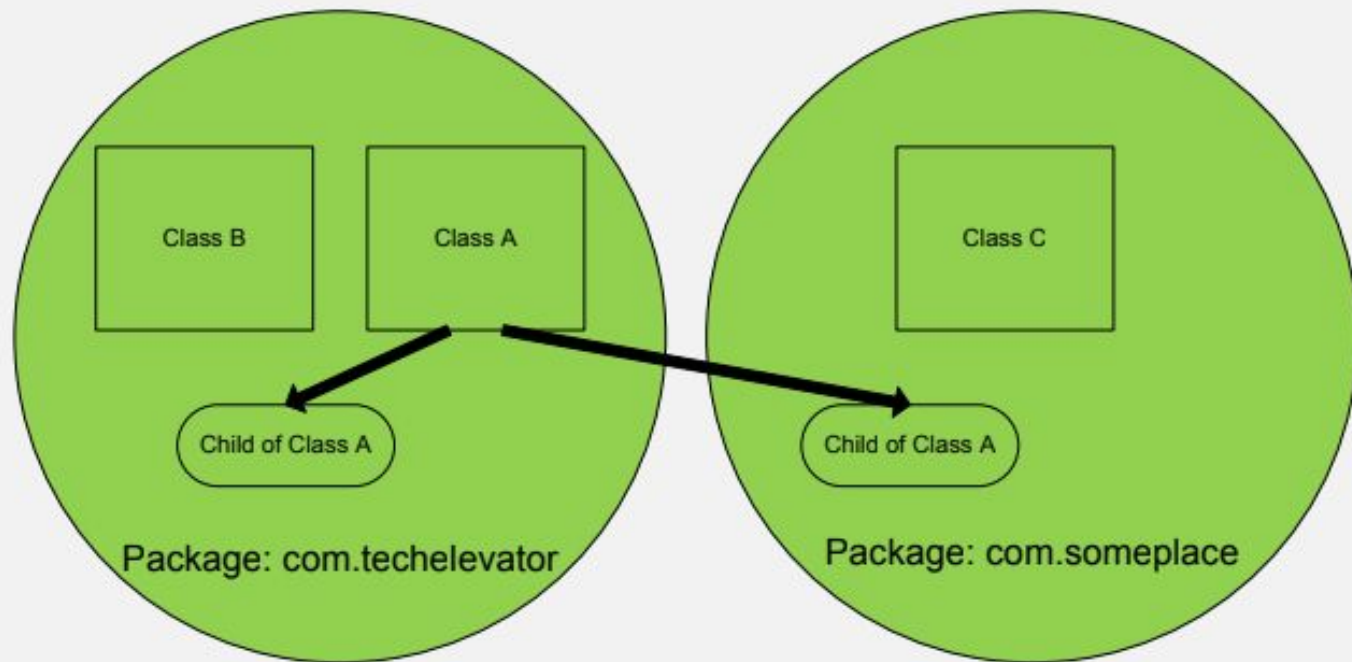
Methods
Constructors
Properties
Inner Classes

Cannot be applied to:

Classes
Interfaces

Protected things can be accessed by: any class in the same package and any subclass, even if in a different package

Public in Class A



Can be applied to:
Methods, Constructors
Properties, Classes, Inner
Classes, and Interfaces

**Public things can be
accessed by:** anything

When to use each accessor

Access	Visibility	Reason to use it
public	Everyone	for “set in stone” methods that you want other programmers to rely on to use your object. These create the behaviors of the object, but changing their method signatures may break other code that is using your object.
protected	Subclasses	for building connections between inherited classes. It lets you have methods in a superclass that are accessible to the subclasses, but does not allow access outside the hierarchy.
default	Package	for building cohesion between related classes in the same package. should generally be avoided.
private	Class	for unstable, worker methods that may change and are only for use inside the class itself.

Class design should include how others will use your object, the methods that allow that use should be public. All other methods and variables should be private, until needed in the hierarchy or publically.