LEC 02

CSE 123

Abstract Classes

Questions during Class?

Raise hand or send here

sli.do #cse123

BEFORE WE START

Talk to your neighbors:

Coffee or tea? Or something else?

Instructor: Ziao Yin

Nichole Trien Packard Eeshani Chris

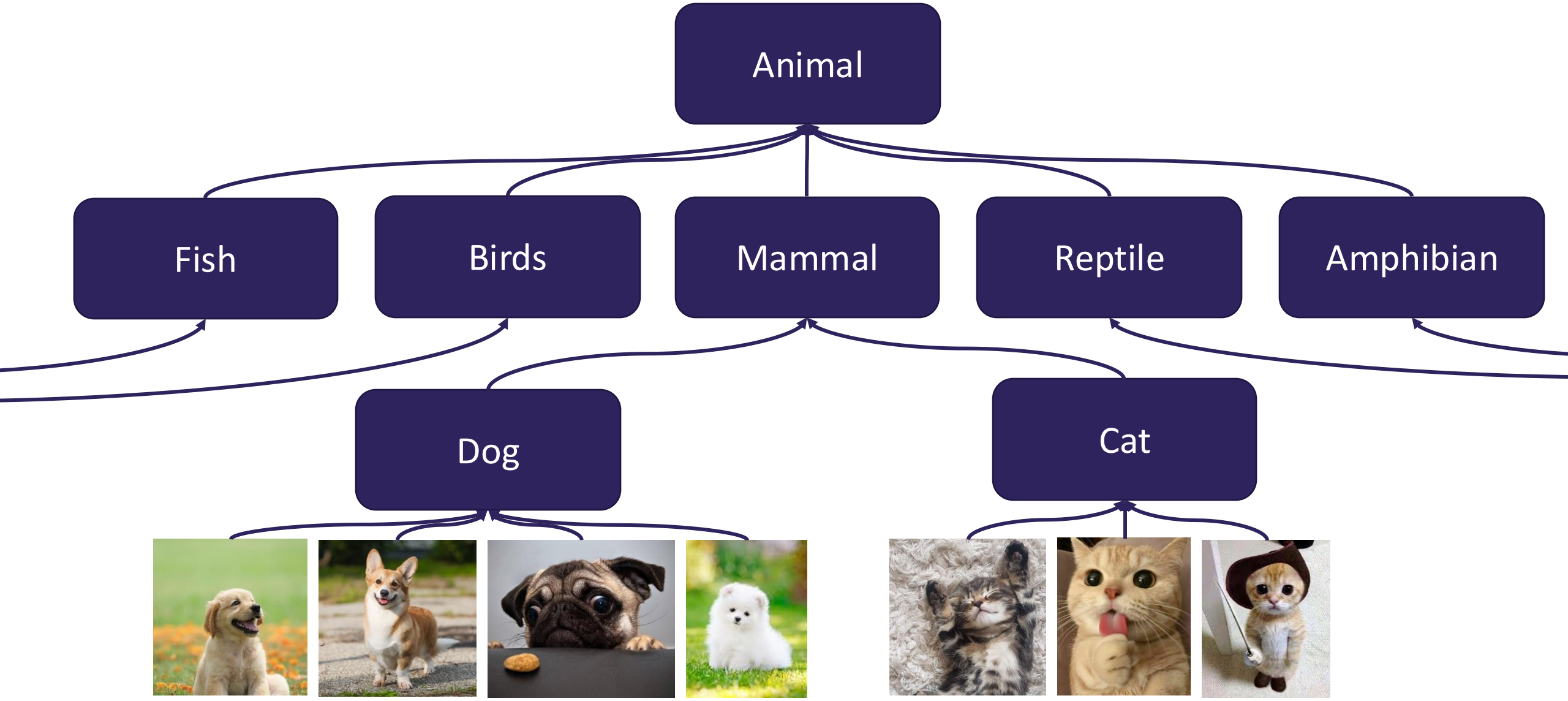
Announcements

- Creative Project 0 due tonight, Wed Jul 2 at 11:59pm!
  - See generic Creative Project rubric posted on website
- Programming Assignment 0 will be released tomorrow, Thurs Jul 3
  - Focused on inheritance and abstract classes

Lecture Outline

- **Polymorphism Review**
  - Declared vs. Actual Type
  - Compiler vs. Runtime Errors
- Abstract Classes Review
- Pre/Post conditions and commenting

Review: Is-a Relationships



Polymorphism

- `DeclaredType x = new ActualType()`
  - All methods in `DeclaredType` can be called on x
  - We've seen this with interfaces (`List<String>` vs. `ArrayList<String>`)
  - Can also be to inheritance relationships

```
Animal[] arr = {new Dog(), new Cat(), new Bear()};
for (Animal a : arr) {
    a.feed();
}
```

Compiler vs. Runtime Errors

- `DeclaredType x = new ActualType()`
  - At compile time, Java only knows **DeclaredType**
  - Compiler error (CE): trying to call a method that isn't present

```
Animal a = new Dog();
a.bark();                    // No bark() -> CE
```

  - Can cast to change the **DeclaredType** of an object

```
((Dog) a).bark();            // No more CE
```

  - Runtime error (RE): attempting to cast to an invalid **DeclaredType***

```
Animal a = new Fish();
((Dog) a).bark();            // Can't cast -> RE
```

  - Order matters! Compilation before runtime

Declared Type and Actual Type

```
DeclaredType varName = new ActualType(…);
```

```
Animal bucky = new Dog("Bucky");
```

Declared Type: Animal
Actual Type: Dog

Can call methods that makes sense for EVERY Animal
If Dog overrides a method, uses the Dog version

```
Dog bucky = new Dog("Bucky");
```

Declared Type: Dog
Actual Type: Dog

Can call methods that makes sense for EVERY Dog
If Dog overrides a method, still uses the Dog version

Compiler vs. Runtime Errors

With the following declaration and initialization:

```
DeclaredType name = new ActualType();
```

If we call:

```
name.method();
```

Does **DeclaredType** contain **method**?

- Yes → Execute **method** of **ActualType**
- No → Compiler Error

```
((CastToType) name).method();
```

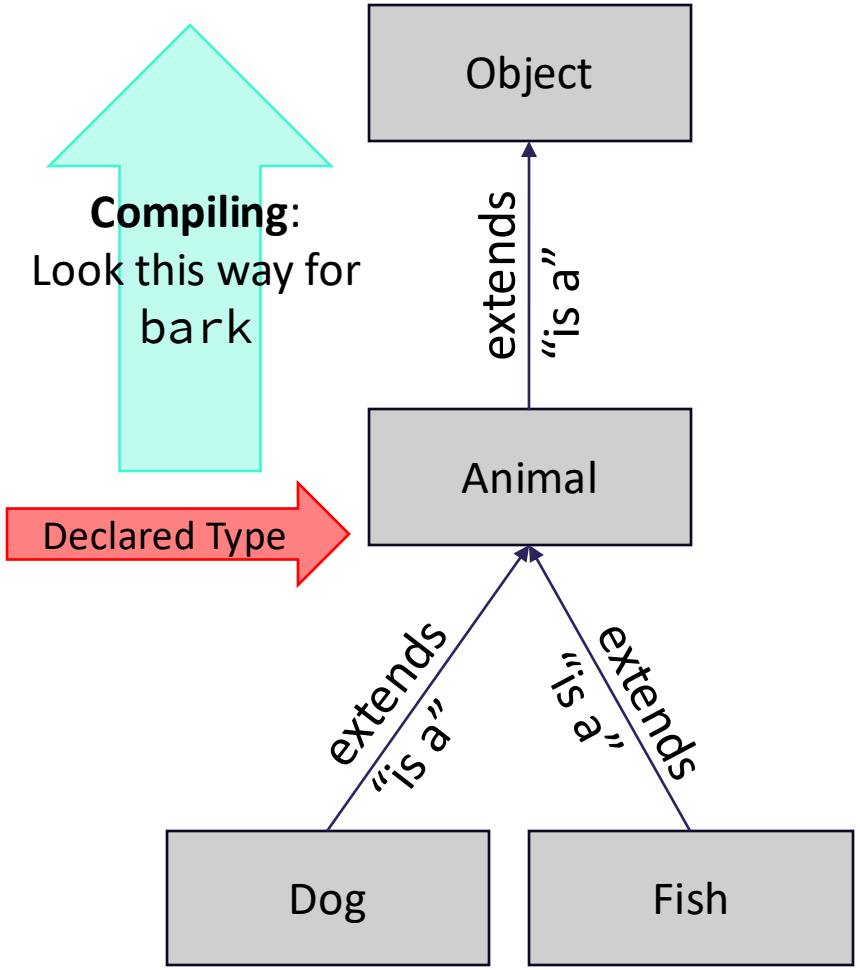
Is it conceivable that an instance of **DeclaredType** could be a **CastToType**? *

*This is true if either **DeclaredType** or at least one of its subtypes can substitute for a **CastToType**

- No → Compiler Error
- Yes → Does **CastToType** contain **method**?
  - No → Compiler Error
  - Yes → Does **ActualType** cast to **CastToType**?
    - No → Runtime Error
    - Yes → Execute **method** of **ActualType**

Inheritance and Method Calls

```
Animal bucky = new Dog();
bucky.bark();
```



When compiling:

Can we *guarantee* that the method exists for the declared type?

Does the declared type or one of its super classes contain a method of that name?

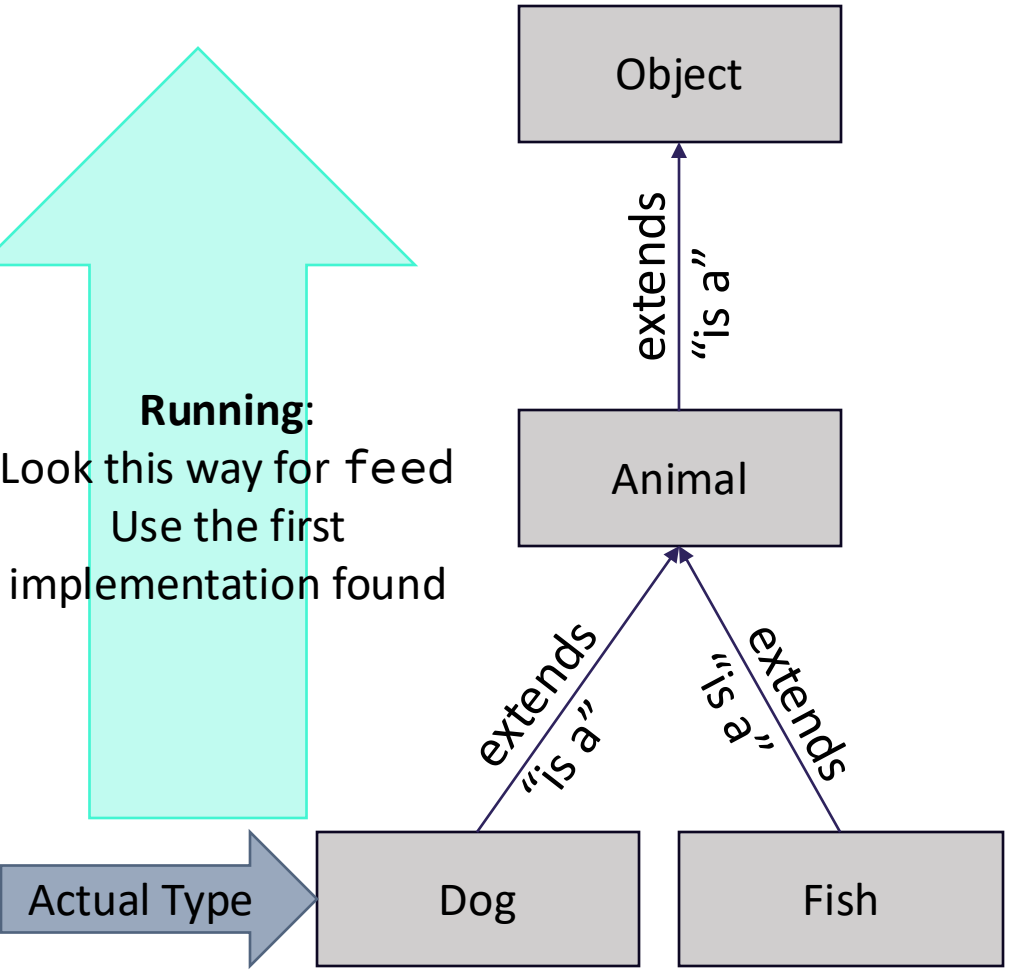
If not… Compile Error!

In this example:

When compiling, neither Animal nor Object have a bark method, so we have a compile error!

Overrides and Method Calls

```
Animal bucky = new Dog();
bucky.feed();
```



When running:

Use the *most specific* version of the method call starting from the actual type.

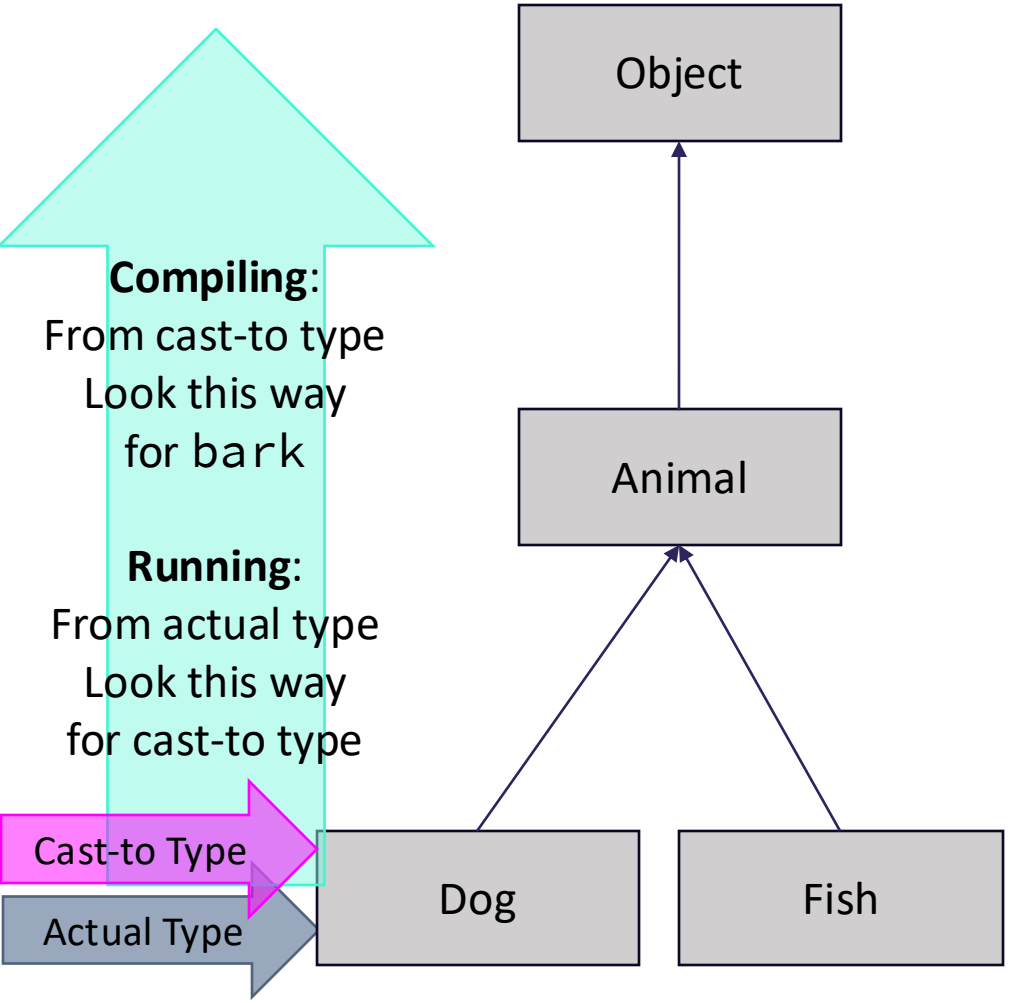
Start from the actual type, then go "up" to super classes until you find the method. Run that first-discovered version.

In this example:

If the Dog class overrides feed, then we'll use the implementation in Dog. Otherwise we'll use the one in Animal

Casting and Method Calls

```
Animal bucky = new Dog();
((Dog) bucky).bark();
```



When compiling:

Can we *guarantee* that the method exists for the Cast-to type?

Does the Cast-to type or one of its super classes contain a method of that name?

If not… Compile Error!

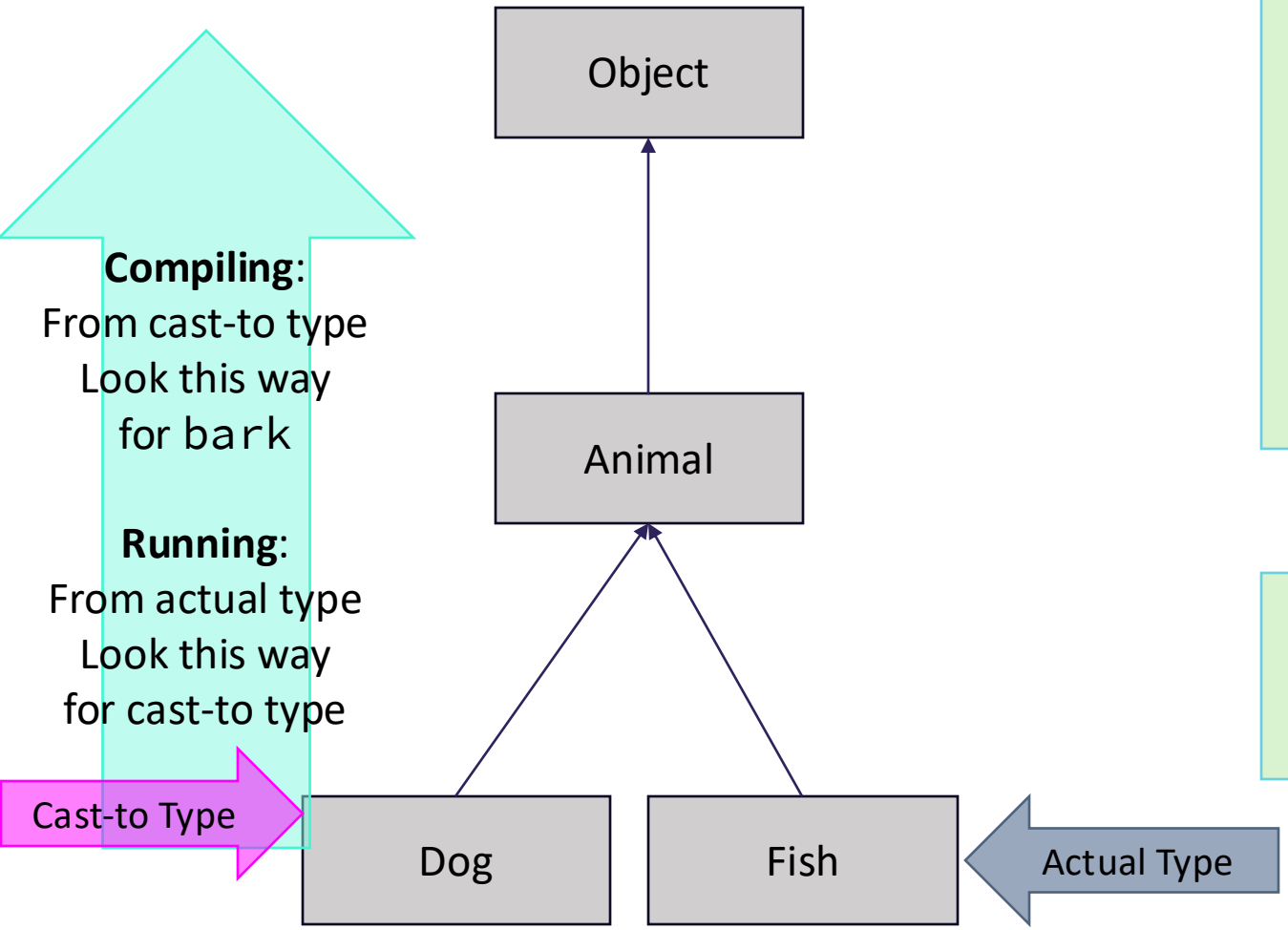
When Running:

Check that the Cast-to Type is either the Actual Type, or one of its super classes

This example has no error

Casting and Method Calls

```
Animal bucky = new Fish();
((Dog) bucky).bark();
```



When compiling:

Can we *guarantee* that the method exists for the Cast-to type?

Does the Cast-to type or one of its super classes contain a method of that name?

If not… Compile Error!

When Running:

Check that the Cast-to Type is either the Actual Type, or one of its super classes

This example has a runtime error

Practice

What results from the following code being executed? (1)

```
Animal bucky = new Dog();
bucky.bark();
```

A. Compiler Error

B. Runtime Error

C. Compiles and runs without error

Practice

What results from the following code being executed? (2)

```
Animal bucky = new Dog();
((Dog) bucky).bark();
```

A. Compiler Error

B. Runtime Error

C. Compiles and runs without error

Practice

What results from the following code being executed? (3)

```
Animal bucky = new Dog();
((String) bucky).meow();
```

A. Compiler Error

B. Runtime Error

C. Compiles and runs without error

Practice

What results from the following code being executed? (4)

```
Animal bucky = new Dog();
((Reptile) bucky).slither();
```

A. Compiler Error

B. Runtime Error

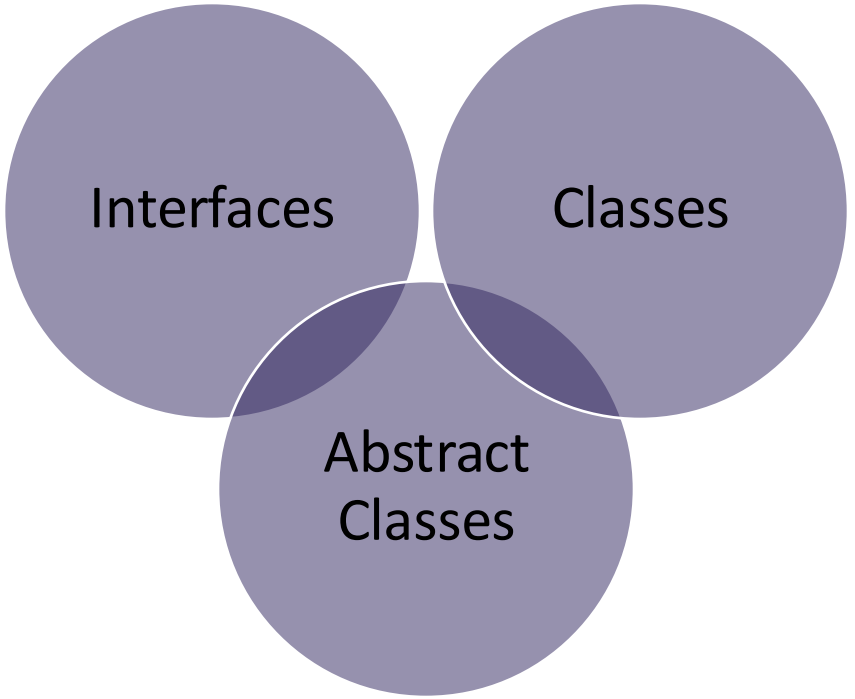
C. Compiles and runs without error

Lecture Outline

- Polymorphism Review
  - Declared vs. Actual Type
  - Compiler vs. Runtime Errors
- **Abstract Classes Review**
- Pre/Post conditions and commenting

Abstract Classes

- Mixture of `Interfaces` and `Classes`
  - `Interface` similarities:
    - Can contain (`abstract`) method declarations
    - Can't be instantiated
  - `Class` similarities:
    - Can contain method implementations
    - Can have fields
    - Can have constructors



- Is there identical / nearly similar behavior between classes that shouldn't inherit from one another?

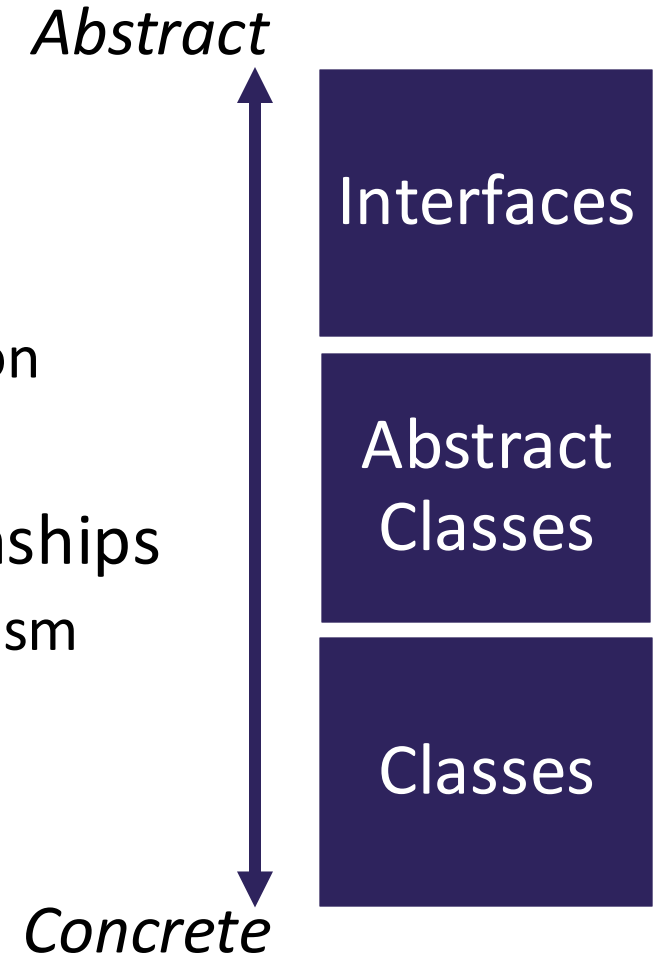
Shape / Square / Circle Example

The starter code contains `Shape`, `Square`, and `Circle` classes similar to the pre-class work, as well as a `Client` that prints out a couple of shapes.

- Add an abstract `getName` method to the `Shape`.
  - Add implementations of `getName` to `Square` and `Circle` that return "Square" and "Circle".
- Add a method `isEmpty` to `Shape` that tells you whether the shape is empty (has zero area) or not.
  - Hint: you will need to call `getArea`, but it may not immediately work…
- Implementing `isEmpty` by calling `getArea` works fine as is, but suppose we wanted to implement it in `Circle` directly. How could we do this just by looking at the fields of the `Circle`? Implement it this way by overriding `isEmpty` in `Circle`.
- Override `toString` in `Shape` to return a similar message to what the `Client` prints in the starter code:
  - Hint: your `toString` can call abstract methods!
- Rewrite the `Client` class to use the new `toString` on shapes.

Advanced OOP Summary

- Allow us to define differing levels of abstraction
  - Interfaces = high-level specification
    - What behavior should this type of class have
  - Abstract classes = shared behavior + high-level specification
  - Classes = individual behavior implementation
- Inheritance allows us to share code via “is-a” relationships
  - Reduce redundancy / repeated code & enable polymorphism
    - Still might not be the “best” decision!
  - Interfaces extend other interfaces
  - (abstract) classes extend other (abstract) classes



- You’re now capable of designing some pretty complex systems!

Design in the "real world"

- In this course, we'll always give you expected behavior of the classes you write
  - Often not the case when programming for real
  - Clients don't really know what they want (but programmers don't either)

- My advice:
  - Clarify assumptions before making them (do I really want this functionality?)
  - **There's no one right answer**
    - Weigh the options, make a decision, and provide explanation
    - Iterative development: make mistakes and learn from them
    - Be receptive to feedback and be willing to change your mind

Interface versus Implementation

- Interface: what something *should* do
- Implementation: *how* something is done
- These are different!
- Big theme of CSE 123:

    choose between different implementations of same interface