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?

Instructors: Brett Wortzman
Miya Natsuhara

Arohan	Neha	Rushil	Johnathan	Nicholas
Sean	Hayden	Srihari	Benoit	Isayiah
Audrey	Chris	Andras	Jessica	Kavya
Cynthia	Shreya	Kieran	Rohan	Eeshani
Amy	Packard	Cora	Dixon	Nichole
Trien	Lawrence	Liza	Helena	

Music: CSE 123 25wi Lecture Tunes

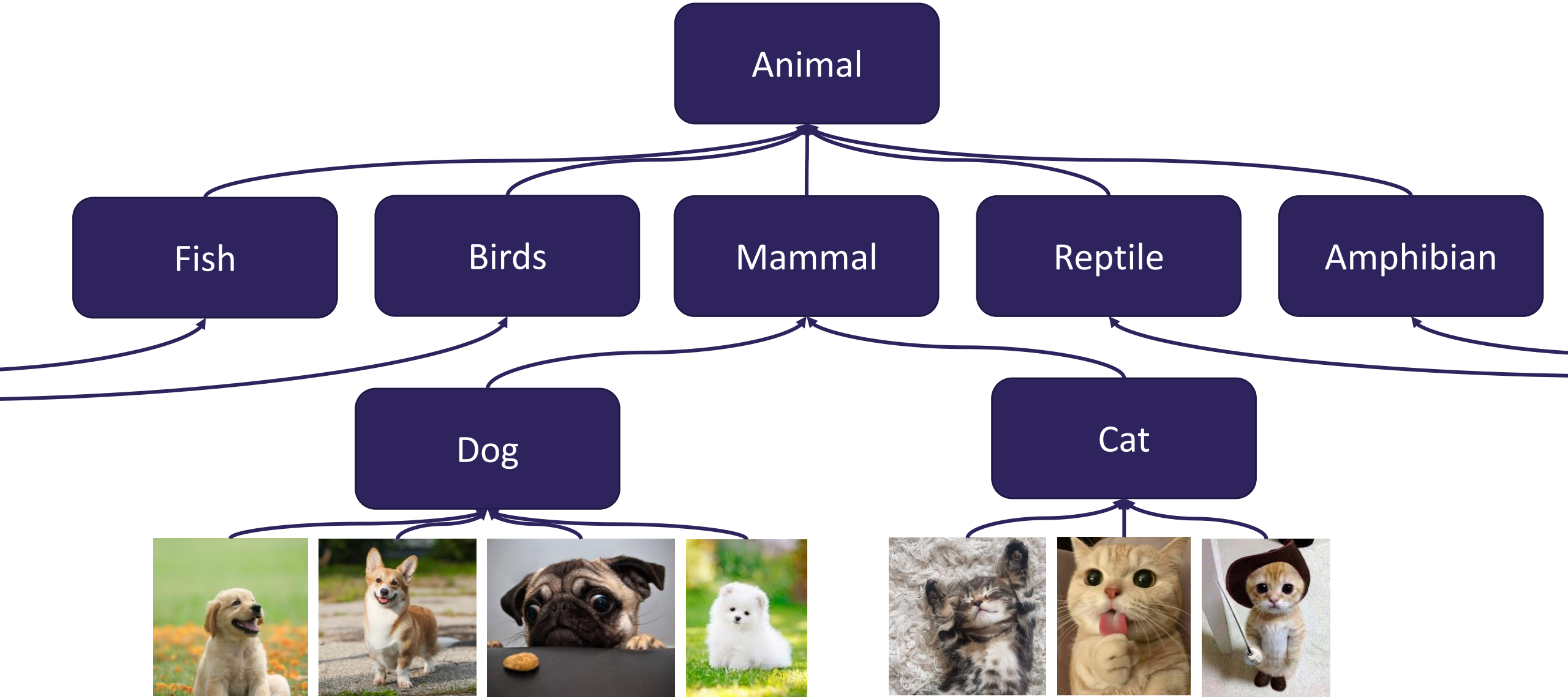
Announcements

- Review sessions held on Monday Jan 13
  - Recordings are linked from the course calendar
- Creative Project 0 due tonight, Wed Jan 15 at 11:59pm!
  - See generic Creative Project rubric posted on website
- Programming Assignment 0 will be released tomorrow, Thurs Jan 16
  - Focused on inheritance and abstract classes
- NOTE: Monday, Jan 20 is a university holiday (MLK Jr. day) so campus will be closed
  - Instructor office hours will be cancelled
  - IPL will be closed
  - Message board will still be available, but response time may vary

Lecture Outline

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

Review: Is-a Relationships



Review: 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

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**

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

Practice : Think

sli.do #cse122

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

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

A. Compiler Error

B. Runtime Error

C. Compiles and runs without error

Practice : Pair

sli.do #cse122

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

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

A. Compiler Error

B. Runtime Error

C. Compiles and runs without error

Practice : Think

sli.do #cse122

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

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

A. Compiler Error

B. Runtime Error

C. Compiles and runs without error

Practice : Pair

sli.do #cse122

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

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

A. Compiler Error

B. Runtime Error

C. Compiles and runs without error

Practice : Think

sli.do #cse122

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

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

A. Compiler Error

B. Runtime Error

C. Compiles and runs without error

Practice : Pair

sli.do #cse122

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

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

A. Compiler Error

B. Runtime Error

C. Compiles and runs without error

Practice : Think

sli.do #cse122

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

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

A. Compiler Error

B. Runtime Error

C. Compiles and runs without error

Practice : Pair

sli.do #cse122

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

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

A. Compiler Error

B. Runtime Error

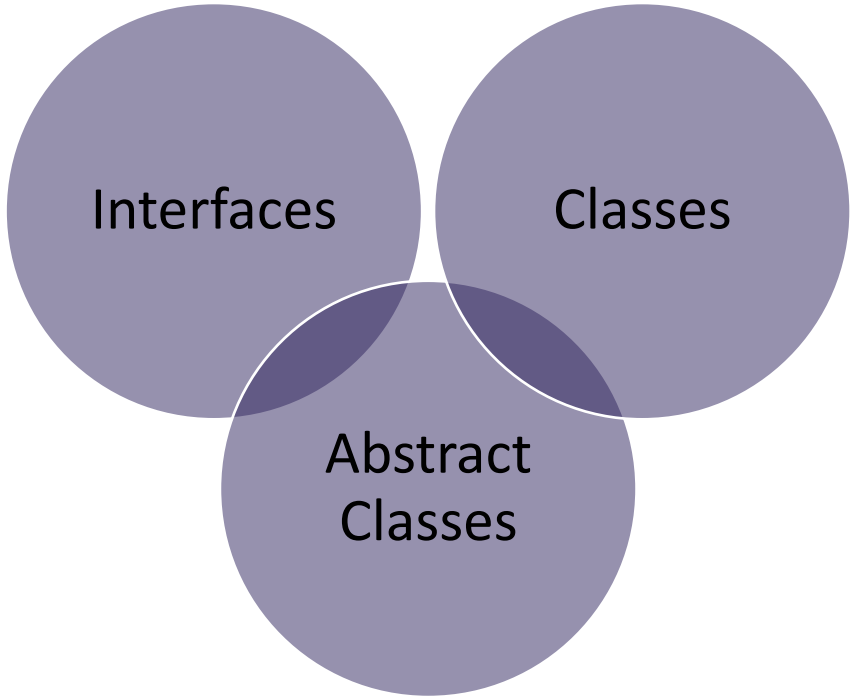
C. Compiles and runs without error

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



- 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