LEC 01

CSE 123

Inheritance; Polymorphism; Comparable

Questions during Class?

Raise hand or send here

sli.do #cse123A

BEFORE WE START

Talk to your neighbors:

Plans for the weekend?

Instructors: Nathan Brunelle

TAs:

Arohan	Ashar	Neha	Rohini	Rushil
Ido	Zachary	Sebastian	Joshua	Sean
Hayden	Caleb	Justin	Heon	Rashad
Srihari	Benoit	Derek	Chris	Bhaumik
Kuhu	Kavya	Cynthia	Shreya	Ashley
Harshitha	Kieran	Marcus	Crystal	Eeshani
Prakshi	Packard	Cora	Dixon	Nichole
Niyati	Trien	Lawrence	Evan	Cady
Ziao				

Grading

Grades should reflect your proficiency in the course objectives

- All assignments will be graded **E (Excellent), S (Satisfactory), or N (Not yet)**
  - Under certain circumstances, a grade of U (Unassessable) may be assigned
- Final grades will be assigned based on the **amount of work at each level**

- See the syllabus for more details

Collaboration Policy

- When we assess your work in this class, we need to know that it's *yours*.
- Unless otherwise specified, **all graded work must be completed individually.**

Some specific rules to highlight:

- do not share your own solution code or view solution code from any source – including but not limited to other students, tutors, or the internet
- do not use AI tools (e.g. ChatGPT) on graded work in any capacity

See the syllabus for more details (this is *very* important to understand).

Coming up…

- ❓ Complete the Introductory Survey
  - This helps us gather data about the students taking our classes and their backgrounds, to inform future offerings.
- 🙋 Review Section0.5 in Ed
  - Includes material covered in cse121 and 122 to help review and jog your memory!
- 💻 The IPL opens Monday, April 7
  - Schedule posted soon
- 🔎 Creative Project 0: Search Engine out now
  - Due Wednesday, April 4, 11:59pm

Collaboration Policy

- When we assess your work in this class, we need to know that it's *yours*.
- Unless otherwise specified, **all graded work must be completed individually.**

Some specific rules to highlight:

- do not share your own solution code or view solution code from any source – including but not limited to other students, tutors, or the internet
- do not use AI tools (e.g. ChatGPT) on graded work in any capacity

See the syllabus for more details (this is *very* important to understand).

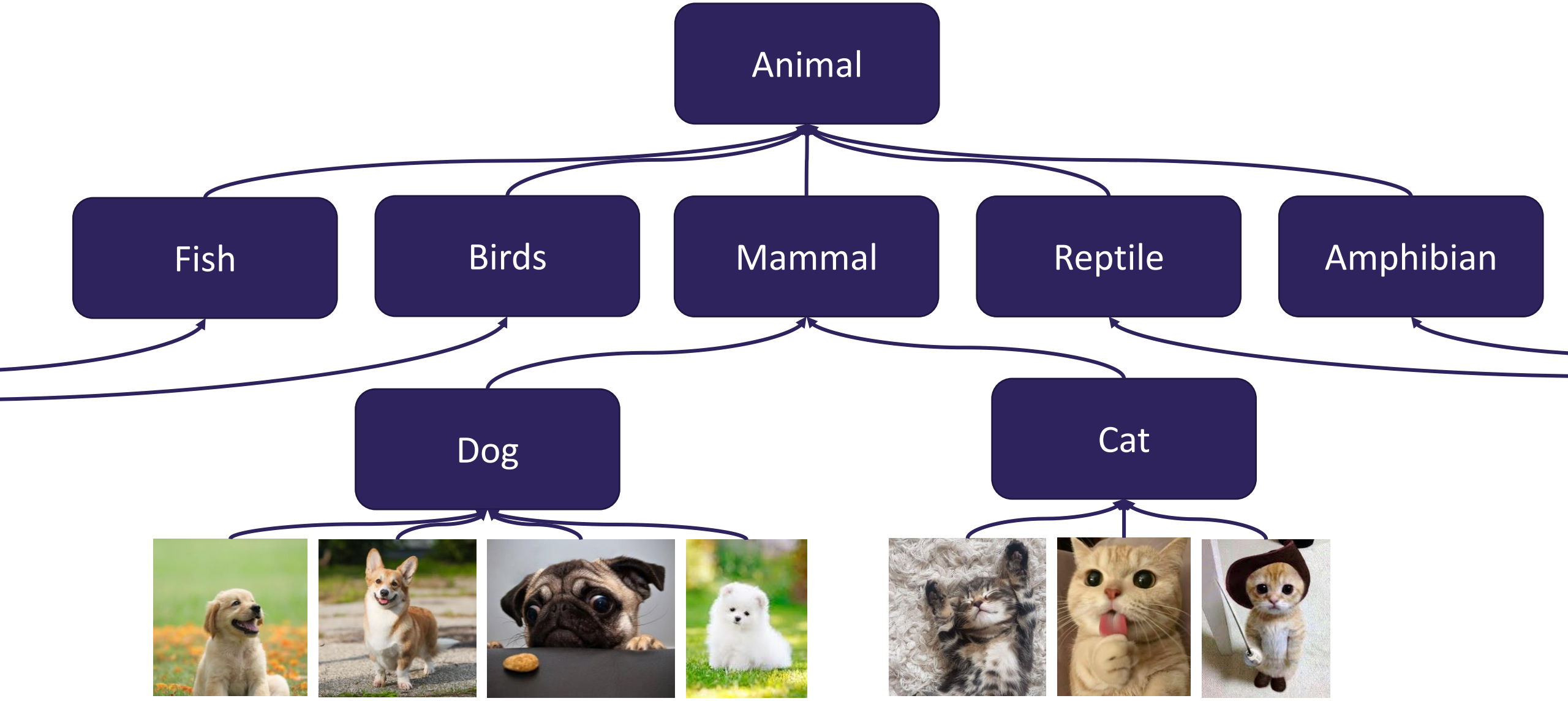
Lecture Outline

- **Inheritance**
- `Comparable`
- Polymorphism
  - Declared vs. Actual Type
  - Compiler vs. Runtime Errors

Inheritance

- Connect together a “subclass” and “superclass”
  - Borrow / “inherit” code to reduce redundancy
  - `super()` keyword can be used just like `this()`
- Syntax: `public class Subclass extends Superclass`
- Should Represent “is-a” relationships
  - `public class Chef extends Employee`
  - `public class Server extends Employee`
- In Java, all objects implicitly inherit from the Object class
  - toString(), equals(Object), etc.

Is-a Relationships



PCM Review

sli.do #cse123A

Lecture Outline

- Inheritance
- `Comparable`
- Polymorphism
  - Declared vs. Actual Type
  - Compiler vs. Runtime Errors

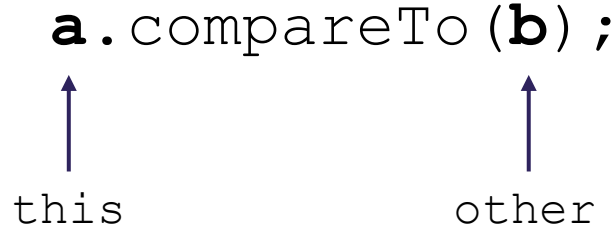
Comparable

- `Comparable<E>` is an `interface` that allows implementers to define an ordering between two objects
  - Used by `TreeSet`, `TreeMap`, `Collections.sort`, etc.

- One required method:

```
public int compareTo (E other);
```

- Returned integer falls into 1 of 3 categories
  - `< 0`: `this` is "less than" `other`
  - `= 0`: `this` is "equal to" `other`
  - `> 0`: `this` is "greater than" `other`



Subtraction Trick

- `compareTo` implementation when comparing two integers (`a`) ascending:

```
if (this.a < other.a)      -> negative number
else if (this.a > other.a) -> positive number
else                       -> 0
```

- This is just subtraction!

```
this.a - other.a
```

- What if we wanted to sort descending?

```
other.a - this.a
```

- **Warning**: this only works for integers! Doubles have issues with truncation.

Lecture Outline

- Inheritance
- `Comparable`
- **Polymorphism**
  - Declared vs. Actual Type
  - Compiler vs. Runtime Errors

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: 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: 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

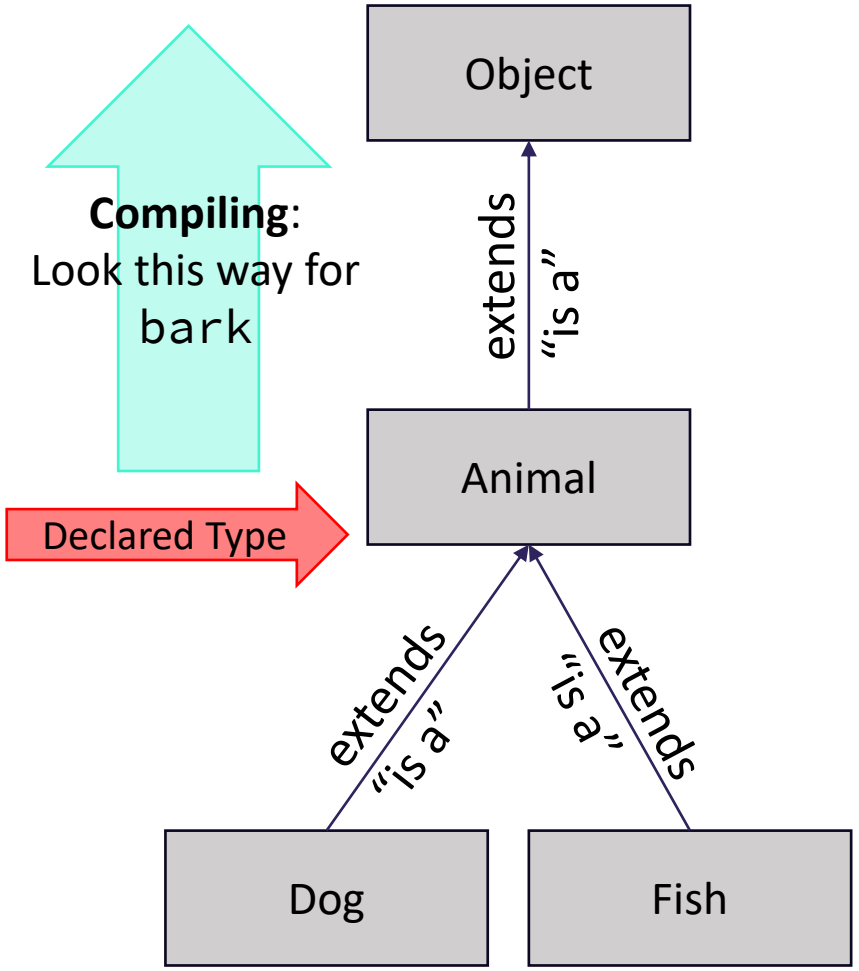
```
Dog bucky = new Dog("Bucky");
```

Declared Type: Dog
Actual Type: Dog

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

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?

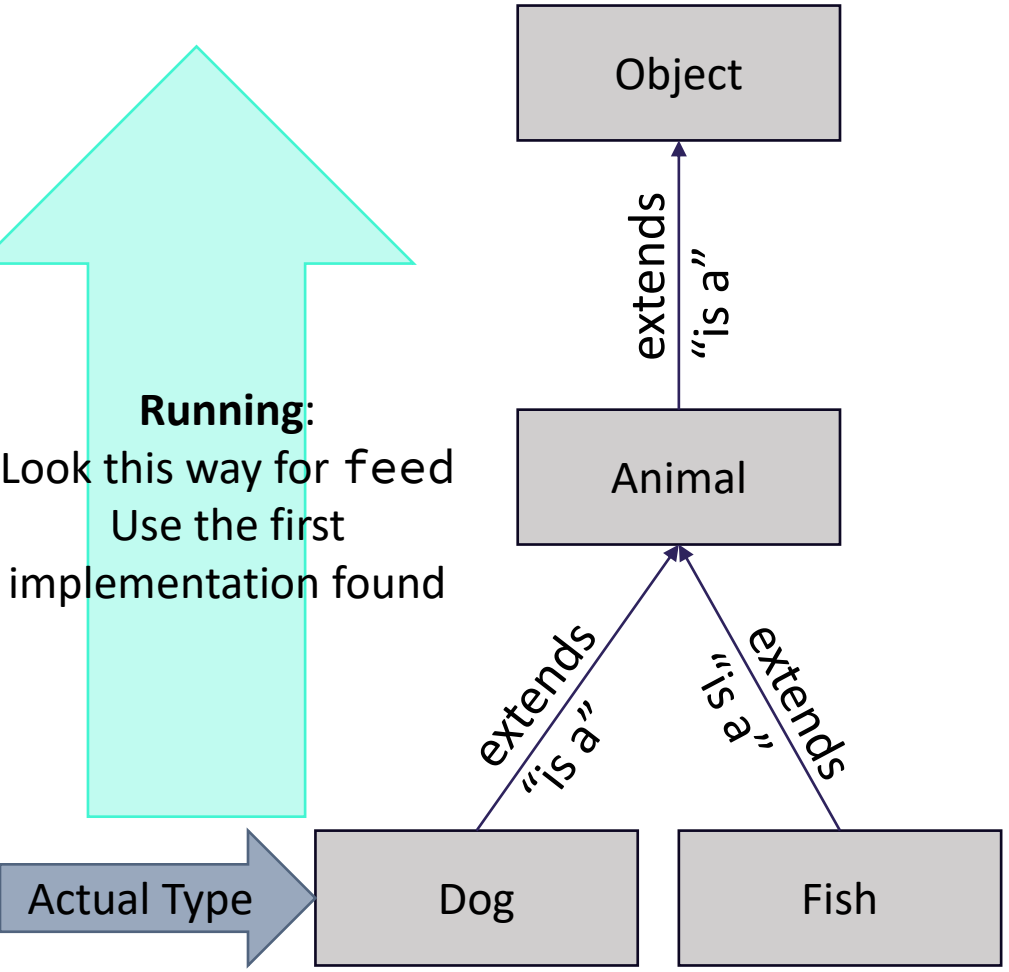
If not… Compile Error!

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.

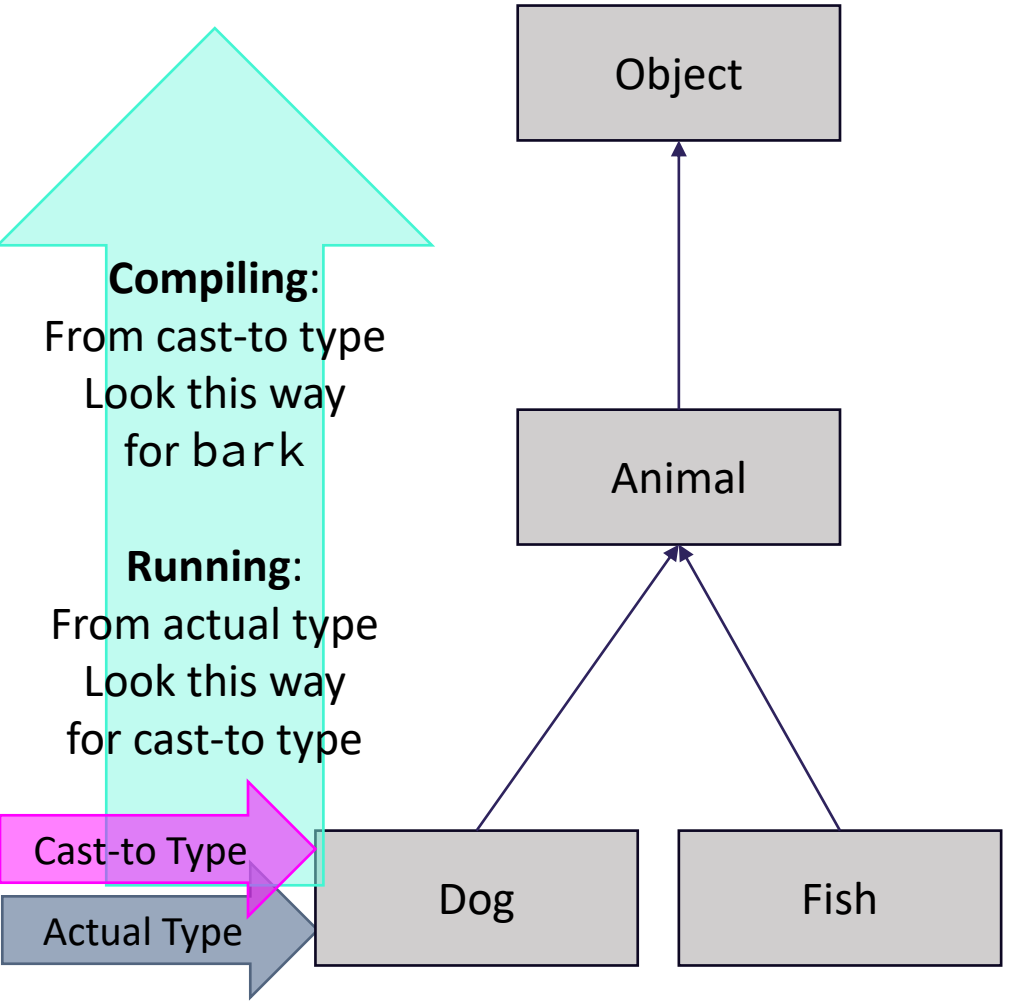
Start from the actual type, then go "up" to super classes until you find the method. Run that first-discovered version.

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!

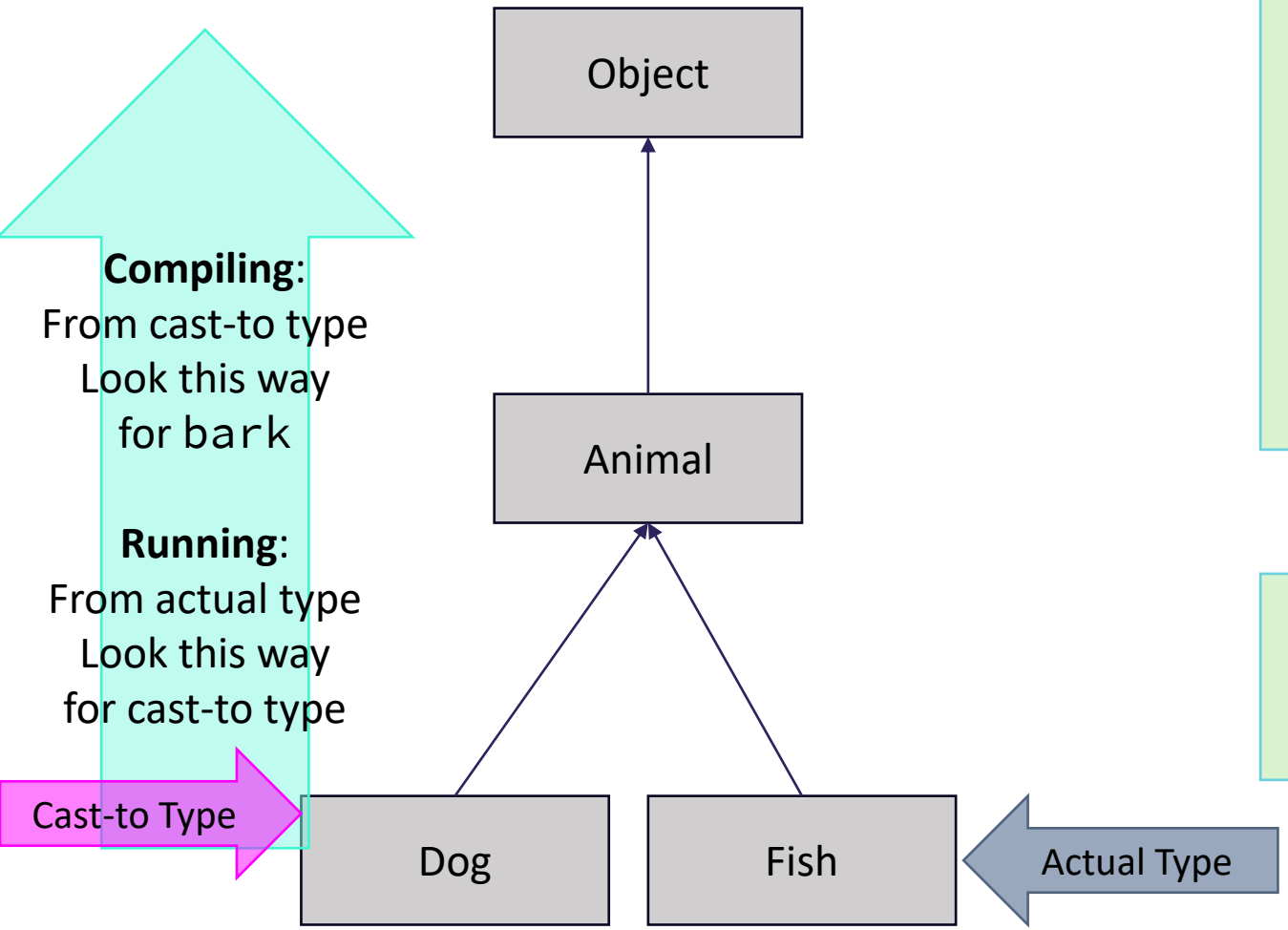
When Running:

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

Compiler vs. Runtime Errors

With the following declaration and initialization:

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