

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: James Wilcox

Announcements

- Creative Project 0 due tonight, Wed October 1 at 11:59pm!
 - See generic [Creative Project rubric](#) posted on website
- Programming Assignment 0 will be released tomorrow, Thurs Oct 2
 - Focused on inheritance and abstract classes

Lecture Outline

- **Polymorphism Review** 
 - Declared vs. Actual Type
 - Compiler vs. Runtime Errors
- Abstract Classes

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 Animal(), new Dog(), new Husky()};  
for (Animal a : arr) {  
    a.speak();  
}
```

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 d = (Dog) a;  
d.bark(); // No more CE
```
 - Runtime error: attempting cast to type that is not a superclass of actual type

```
Animal a = new Fish();  
Dog d = (Dog) a; // Can't cast -> RE  
d.bark();
```
 - Order matters! Compilation before runtime

Declared Type and Actual Type

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

ActualType must be a subclass of (or same as) DeclaredType

```
Animal rufus = new Dog("Rufus");
```

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 rufus = new Dog("Rufus");
```

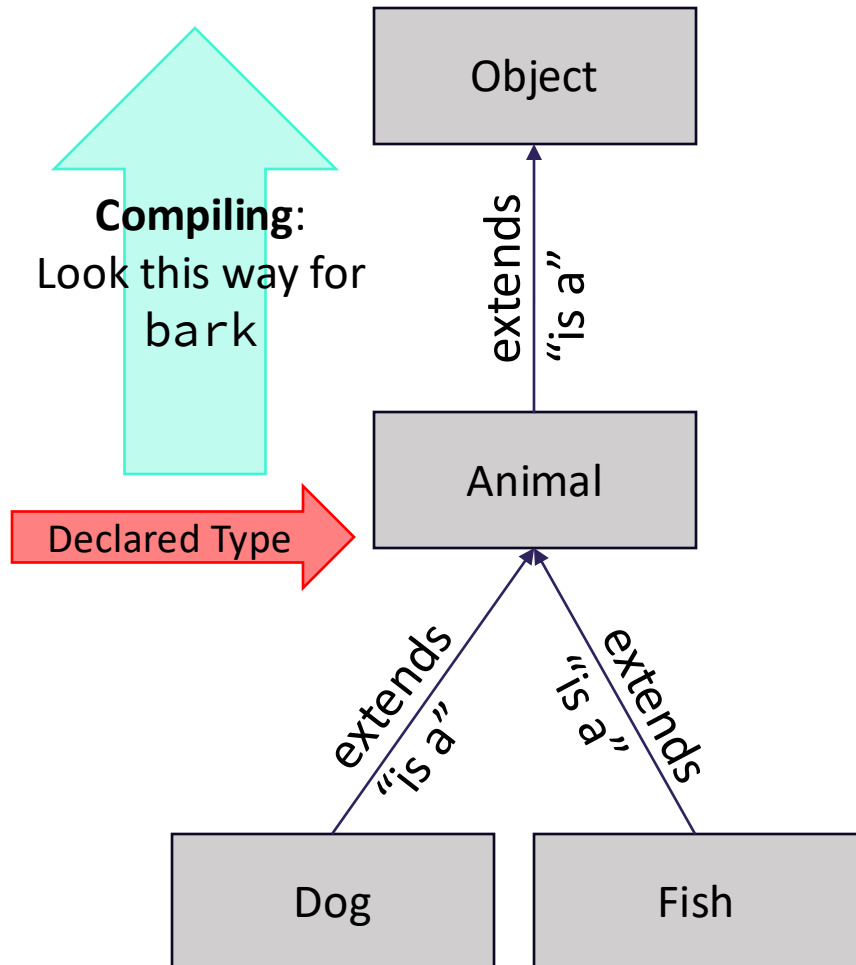
Declared Type: Dog

Actual Type: Dog

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

Compiling Method Calls

```
Animal rufus = new Dog();  
rufus.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?

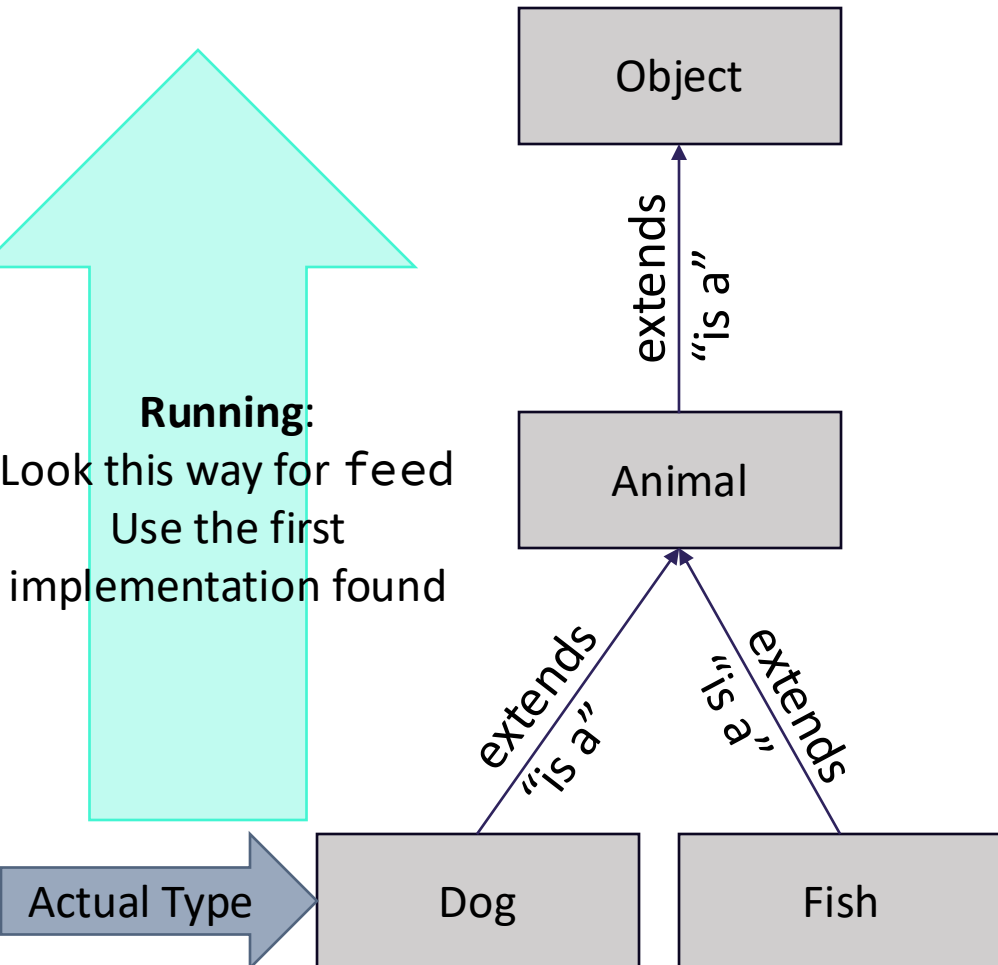
If not... Compile Error!

In this example:

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

Running Method Calls

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



When running:

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

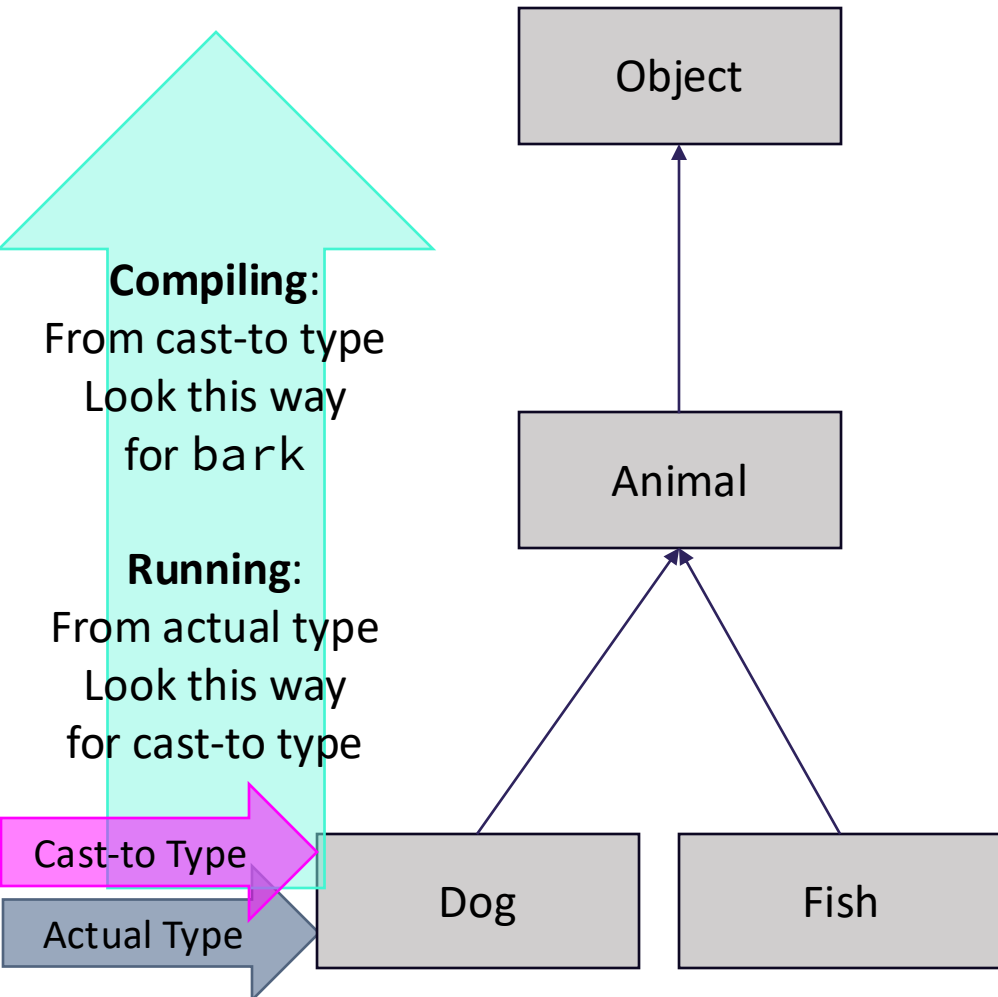
Start from the **actual type**, then go “up” to super classes until you find the method. Run that first-discovered version.

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

Casts and Method Calls

```
Animal rufus = new Dog();  
Dog d = (Dog) rufus;  
d.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!

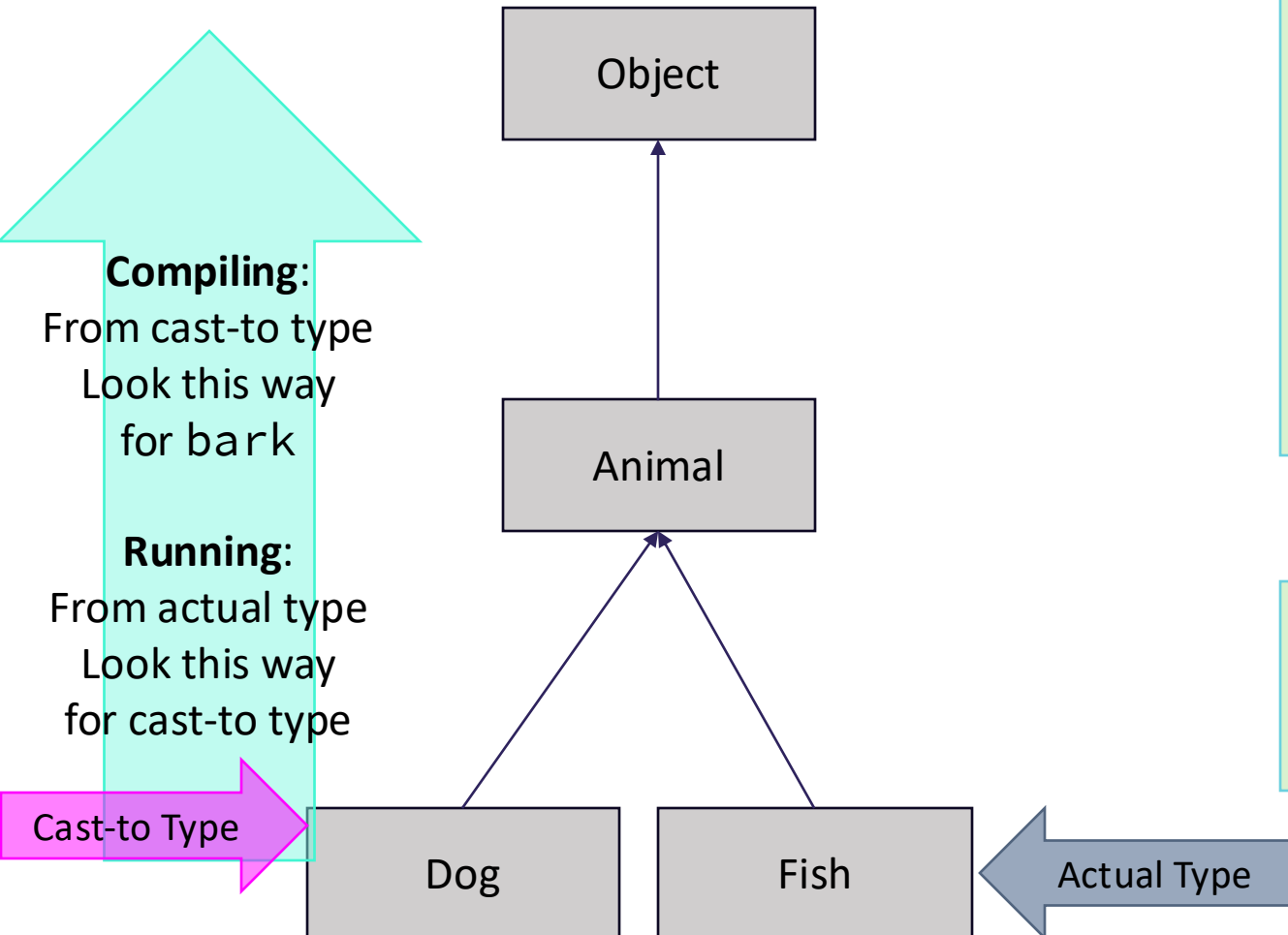
When Running:

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

This example has no error

Casts and Method Calls

```
Animal rufus = new Fish();  
Dog d = (Dog) rufus;  
d.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?

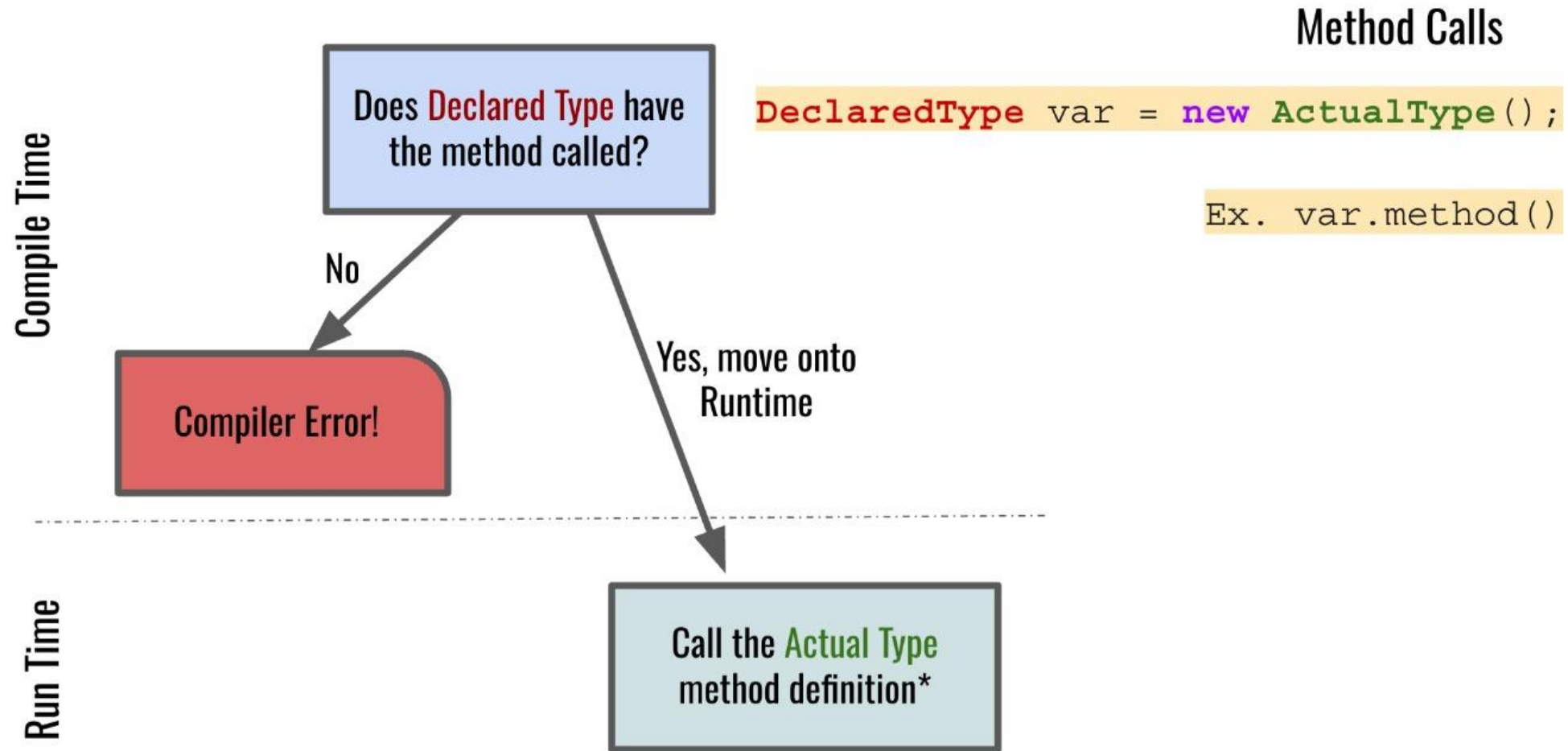
If not... Compile Error!

When Running:

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

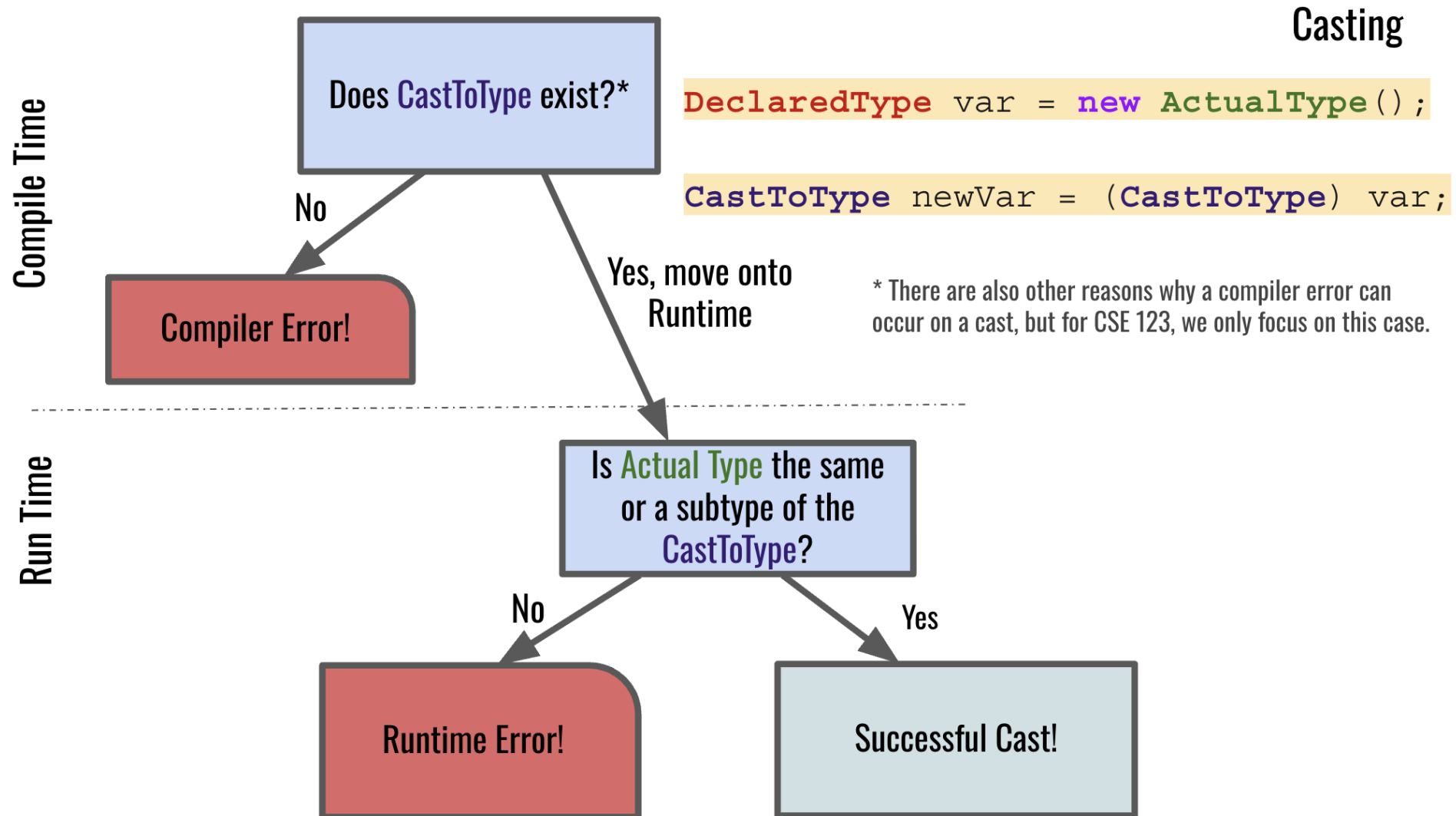
This example has a runtime error

Compiler vs. Runtime Errors: Method Calls




* Start at the Actual Type at run time. If it doesn't exist in Actual Type, trace up the inheritance tree to find the nearest inherited method definition

Compiler vs. Runtime Errors: Casting



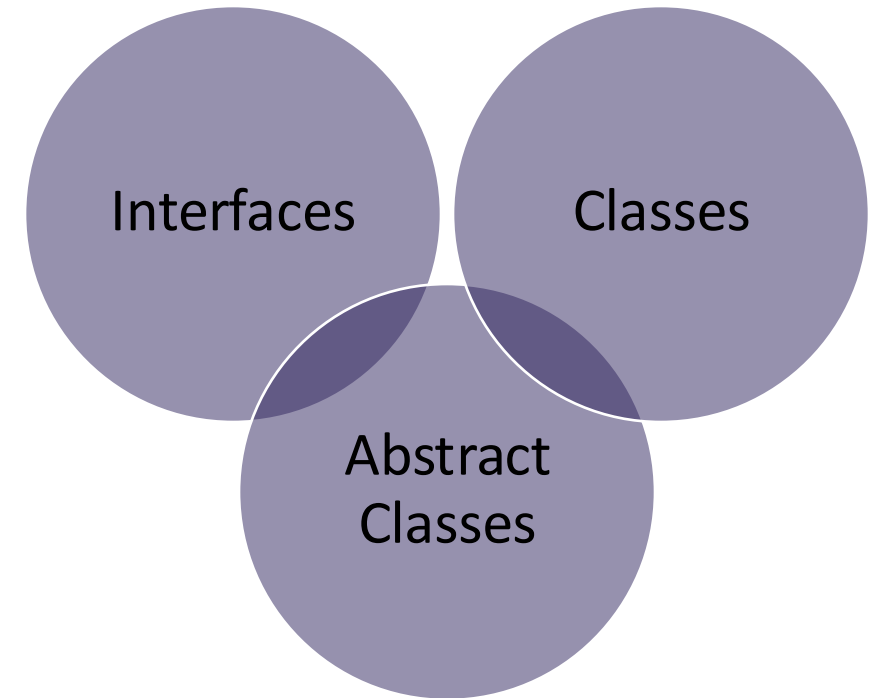
Lecture Outline

- Polymorphism Review
 - Declared vs. Actual Type
 - Compiler vs. Runtime Errors
- **Abstract Classes** 

Abstract Classes

- Mixture of Interfaces and Classes

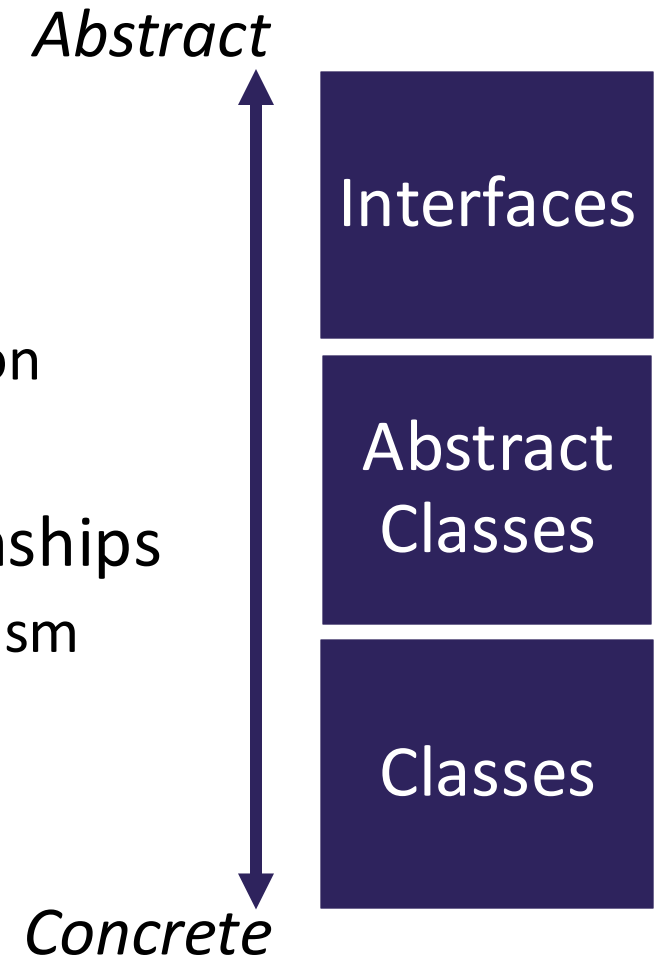
- Interface similarities:
 - Can contain (abstract) method declarations
 - Can't be instantiated
- Class similarities:
 - Can contain method implementations
 - Can have fields
 - Can have constructors



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

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