

Object-Oriented Programming Part 2

Java Accelerator 7

Lesson 1.3





In this lesson we'll become familiar with the main features of an object-oriented programming language.

Learning Objectives



Design Java classes.




Compare and contrast the use of interfaces, composition, and inheritance.



Instantiate and reference common Java types.

Interfaces

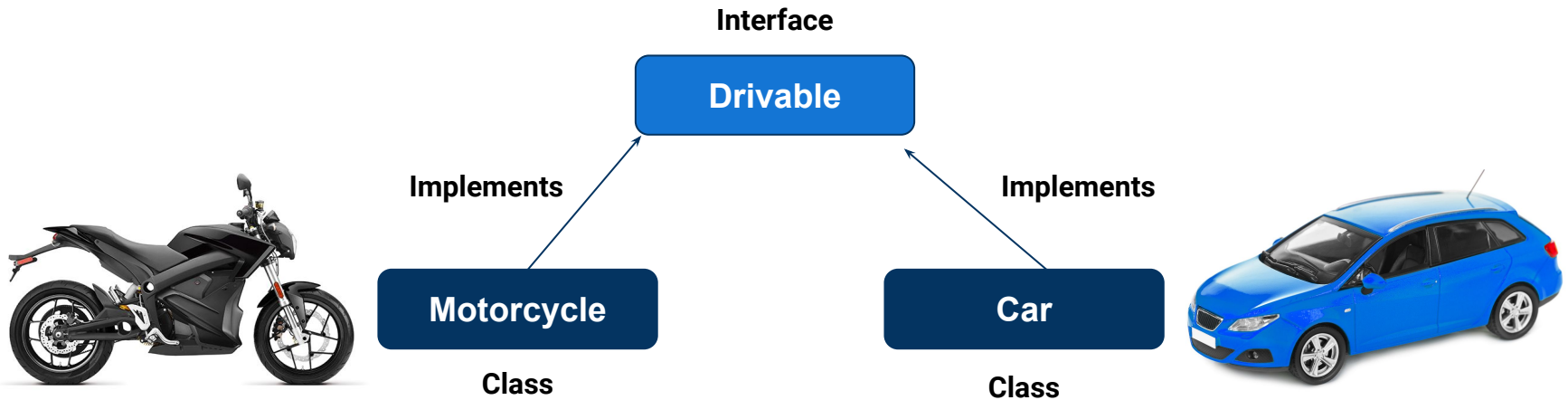


Interfaces allow us to express loosely coupled, contract-like relationships.

Interfaces

Beyond the contract metaphor, interfaces also advertise interoperability.

An interface indicates that an implementing class has a particular set of capabilities. So if two classes fulfill the contract expressed in a particular interface, you can expect the two classes to share the same capabilities defined in that interface. (They might have other, completely unrelated capabilities as well.)





**What are some real-world
examples of contract-like,
loosely coupled relationships?**

Interface Examples



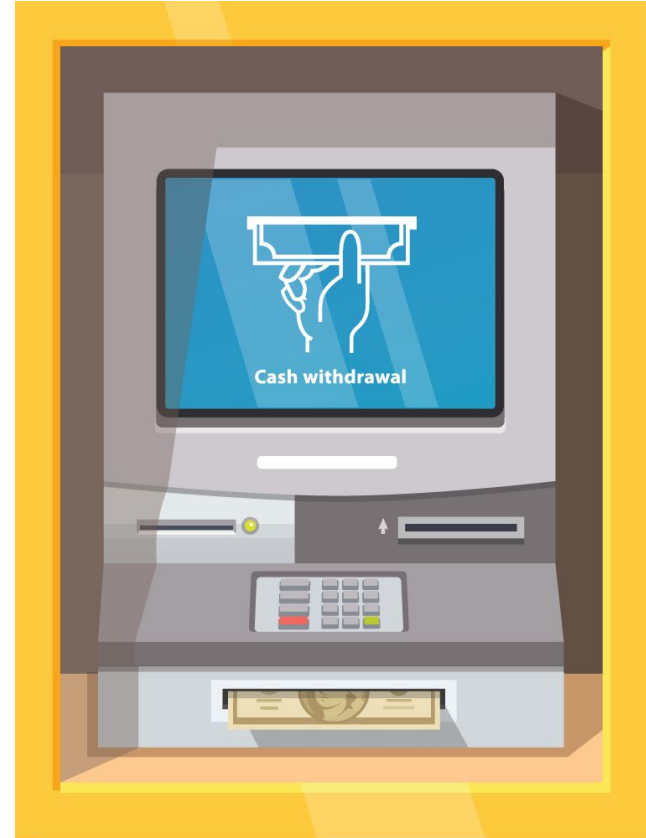
ATM machines



Amazon



Drive-through pharmacy



Interface

An interface is a contract.

01

Interfaces help us express contract-based relationships—these relationships are loosely coupled and tell us nothing about how the functionality behind the contract is fulfilled.

02

Interfaces can help us create a specification that allows different components to “plug in” to a system.



Time to Code

Interface Declaration

Suggested Time:



Activity: UserIO Interface

Suggested Time:

Composition



We're going to explore the
has-a, or composition,
relationship in this lesson.

What Is Composition?

Key points:



One of the key features of an object-oriented language is that objects can be made up of other objects. We call this idea **composition**.



There is no reason to reinvent the wheel if you can leverage existing code. It is often useful to pull code out into new classes so that the code can be used in several places.



The outside world doesn't need to know how we implement the features of the class. What we're implementing and what we're delegating are part of the **private implementation**.



Composition represents a **has-a** relationship.



**What are some examples of composition
from the real-world?**

Has-A Relationships

A house **HAS** A
refrigerator.





A car HAS AN engine.



Time to Code



Composition Account

Suggested Time:

30 minutes



What are some limitations with the final version of the Account class?



Activity: Composition Home

Suggested Time:

30 minutes

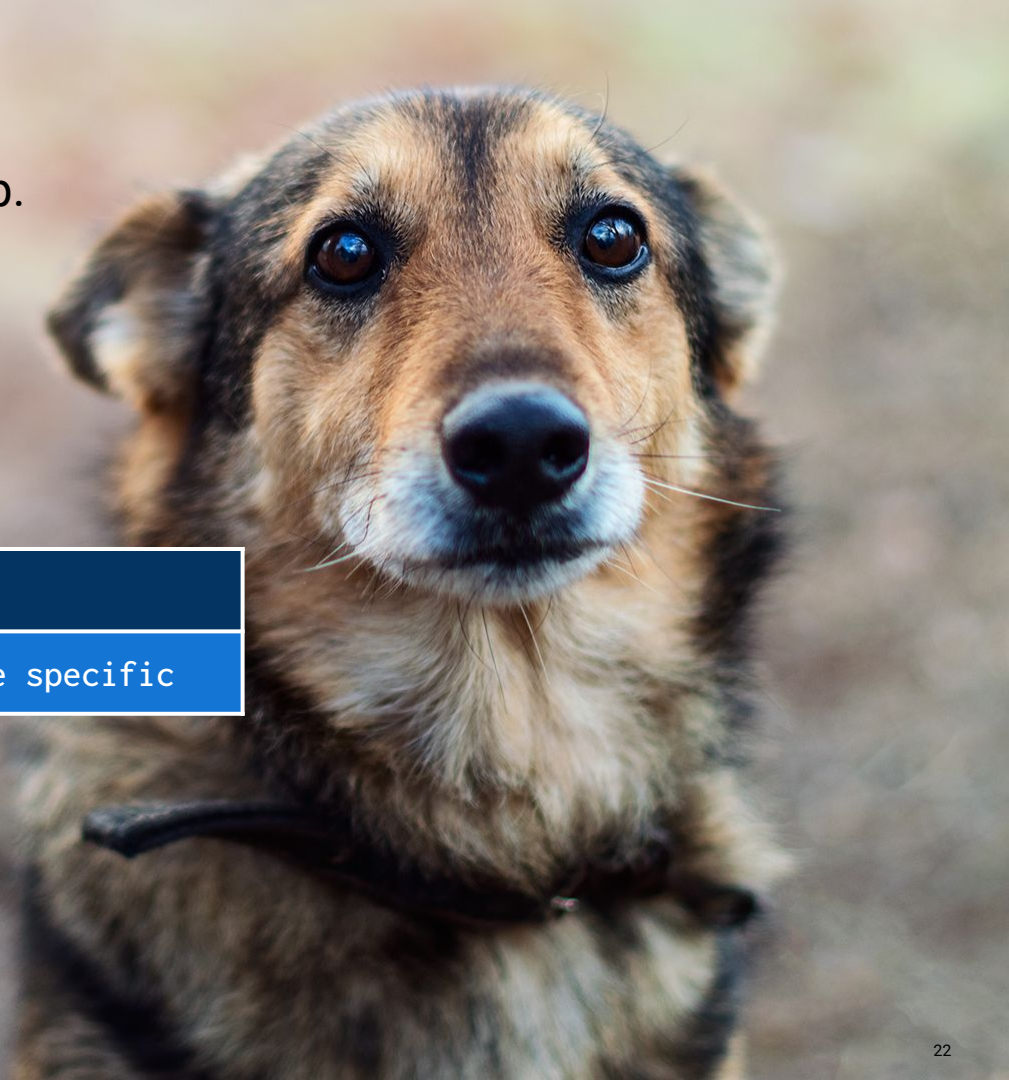
Inheritance

Inheritance

Inheritance models an **is-a** relationship.
is-a relationships are hierarchical.

A good inheritance hierarchy
is built from the most general
to the most specific.

animal ->	mammal ->	dog
general ->	more specific ->	even more specific





Terminology

Base class



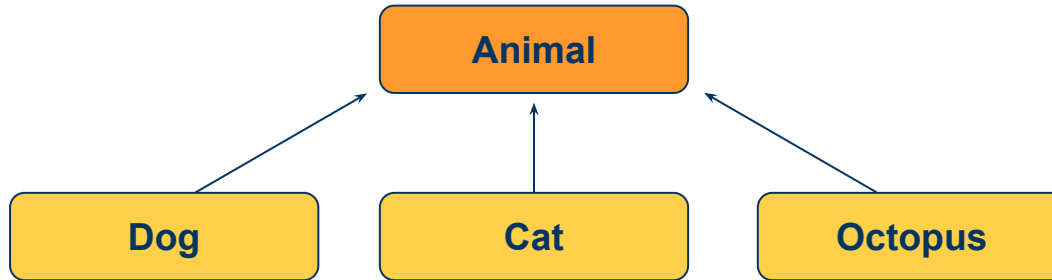
This is also referred to as the Parent class or superclass.



Classes extend a base class.



These classes inherit the methods and properties of the base class.



Derived class

This is also referred to as the subclass, child class, or extended class.



The derived class inherits all of the methods and properties from the base class.



The derived class specializes the base class.



The derived class can add more methods and properties.



The derived class can provide a different implementation of the properties and methods that exist in the base class. This is called **overriding**.

Derived class

We use the `extends` keyword to accomplish inheritance:

```
class Dog extends Mammal {  
  
}
```



Time to Code



Inheritance

Suggested Time:

Polymorphism

Polymorphism Example

animal ->

mammal ->

dog



Can we treat a dog as a mammal?



What happens if we do?



Can we "access" all the features of a dog if we're treating it as a mammal?

Polymorphism Example

Developer ->

TeamLead ->

Architect



What happens if we treat Architect as TeamLead?



Can we use all the characteristics of Architect if we treat it as TeamLead?

Polymorphism Example

Developer ->

TeamLead ->

Architect



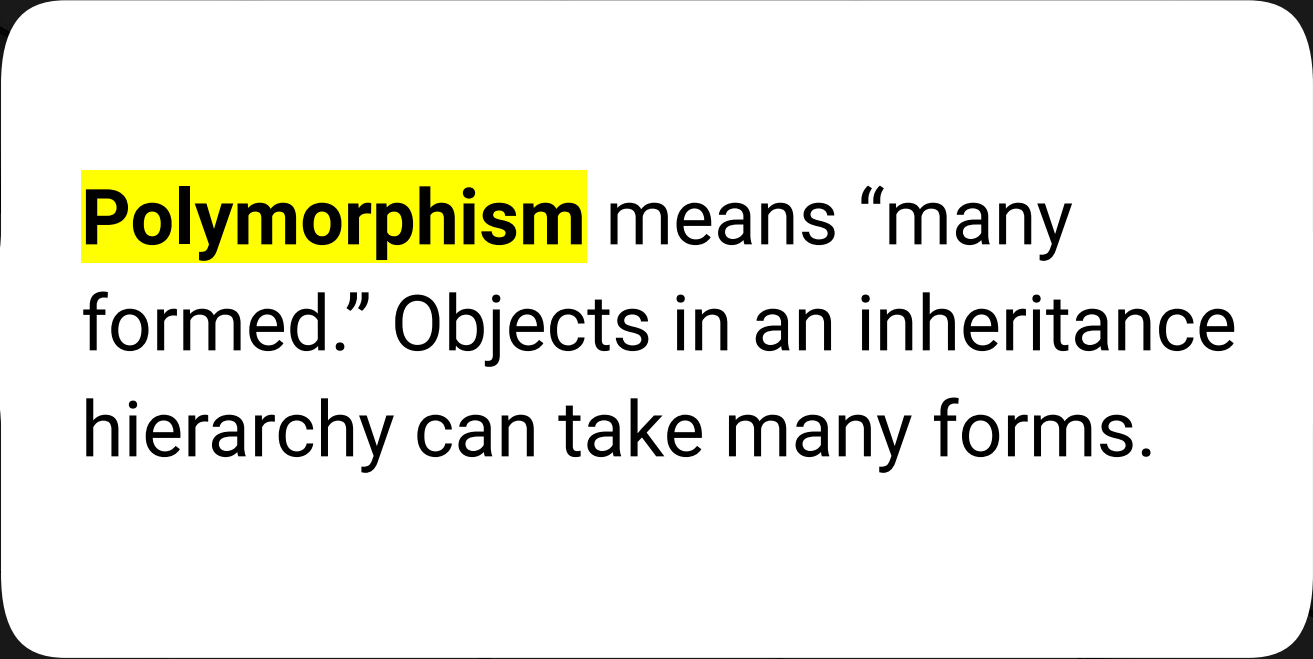
What about treating an Architect as a Developer?



Can we use all the characteristics of an Architect?



Can we use all the characteristics of a TeamLead?



Polymorphism means “many formed.” Objects in an inheritance hierarchy can take many forms.



Time to Code

Polymorphism

Suggested Time:



Activity: Shapes and Perimeters

Suggested Time:



Interfaces



Interfaces express a loosely coupled, contract-based relationship. The relationship is all about WHAT the class does but has nothing to do with HOW the class does it.



Interfaces are used extensively in enterprise development as a way to provide pluggable structures and interoperability.



Interfaces cannot have properties.



Interfaces generally do not have implementations for their methods—just definitions.



Interfaces can provide default implementations for methods, which is useful if you need to add a method to an existing interface.



You can prevent existing classes that implement the interface from breaking when you introduce a new method by using the default method implementation.

Composition



Composition expresses a **has-a** relationship.



Composition reflects the principle of OOP that objects are made up of other objects.



We can reuse existing objects in classes or create new classes that can be used in several places.



For example, a car HAS a steering wheel, but a car isn't a steering wheel.

Inheritance



Inheritance expresses an is-a relationship.
For example, a cat **is-a** mammal and an architect **is-a** developer.



Both classes and interfaces can be extended.



Child classes automatically inherit the public and protected members of parent classes.



Method overriding is the mechanism that lets us replace the implementation of a method provided by a parent class.

Polymorphism



Polymorphism means "many forms" and refers to the fact that objects can be referred to by either their base class types or interface types. For example, we can use a `Developer` reference to refer to an `Architect`.



Remember, derived types can also be base types. A `TeamLead` is derived from a `Developer`. But `TeamLead` is the base type of an `Architect`.



Remember that polymorphism only works in one direction:
All cats are mammals, but not all mammals are cats.