Introduction

In this section, I'll be talking about abstraction and generalization.

These concepts, in action, reduce the amount of code you have to write and encourage extensible and flexible code.

When I say code is extensible, I mean it can support future enhancements and changes, with little or no effort.

An extensible application is designed with change in mind.

In this section of the course, we'll be looking at Java's support for these two concepts.

Generalization and Abstraction

We use the terms **Abstraction** and **Generalization**, when we start trying to model real world things in software.

Before I launch into interface types and abstract classes, I want to talk about what these concepts mean.

Generalization

When you start modeling objects for your application, you start by identifying what features and behavior your objects have in common.

We generalize when we create a class hierarchy.

A base class is the most general class, the most basic building block, which everything can be said to have in common.

Abstraction

Part of generalizing is using abstraction.

You can generalize a set of characteristics and behavior into an abstract type.

If you consider an octopus, a dog, and a penguin, you will probably say they're all animals.

An animal is really an abstract concept.

An animal doesn't really exist, except as a way to describe a set of more specific things.

If you can't draw it on a piece of paper, it's probably abstract.

Abstraction simplifies the view of a set of items' traits and behavior, so we can talk about them as a group, as well as generalize their functionality.

Java's support for Abstraction

Java supports abstraction in several different ways.

- Java allows us to create a class hierarchy, where the top of the hierarchy, the base class, is usually an abstract concept, whether it's an abstract class or not.
- Java lets us create abstract classes.
- Java gives us a way to create interfaces.

Abstract method

In the videos that follow, I'll be talking a lot about abstract and concrete methods.

An abstract method has a method signature, and a return type, but doesn't have a method body.

Because of this, we say an abstract method is **unimplemented**.

Its purpose is to describe behavior, which any object of that type will always have.

Conceptually, we can understand behaviors like move or eat on an Animal, so we might include those as abstract methods, on an abstract type.

You can think of an abstract method as a contract.

This contract promises that all subtypes will provide the promised functionality, with the agreed upon name and arguments.

Concrete method

A concrete method has a method body, usually with at least one statement.

This means it has operational code, that gets executed, under the right conditions.

A concrete method is said to **implement** an abstract method, if it overrides one.

Abstract classes and interfaces can have a mix of abstract and concrete methods.

Method Modifiers

I've already covered access modifiers and what they mean for types, as well as members of types.

And you know there are public, protected, package, and private access modifiers, as options for the members.

Method Modifiers

In addition to access modifiers, methods have other modifiers, which I'll list here, as a high-level introduction.

Modifier	Purpose
abstract	When you declare a method abstract, a method body is always omitted. An abstract method can only be declared on an abstract class or an interface.
static	Sometimes called a class method, rather than an instance method, because it's called directly on the Class instance.
final	A method that is final cannot be overridden by subclasses.
default	This modifier is only applicable to an interface, and I'll talk about it in our interface videos.
native	This is another method with no body, but it's very different from the abstract modifier. The method body will be implemented in platform-dependent code, typically written in another programming language such as C. This is an advanced topic and not generally commonly used, and we won't be covering it in this course.
synchronized	This modifier manages how multiple threads will access the code in this method. I'll cover this in a later section on multi-threaded code.

Abstraction - What is it?

Now consider these sentences.

- We adopted a new pet this weekend.
- I ordered something I really wanted from the store.
- I bought a ticket and won a prize.

If I said any of these things to a friend or coworker, it might be frustrating for them.

I haven't given them enough information in each of these cases.

They can't visualise what I am talking about, because they lack details. Which I didn't provide.

New pet, something ordered, and a ticket, are too general when talking about one item.

Abstraction - What is it?

On the other hand, when we talk about **groups of things**, we don't usually need too many specifics.

Consider these sentences.

I need to get home to feed the animals.

I'm waiting for my box of stuff from an online store to be delivered.

So here, animals, and stuff are probably enough information, to fully describe the situation.

The abstract class

The abstract class is declared with the **abstract** modifier.

Here I declare an abstract class called Animal.

```
abstract class Animal {}     // An abstract class is declared with the abstract
                             // modifier.
```

An **abstract** class is a class that's **incomplete**.

You can't create an instance of an abstract class..

```
Animal a = new Animal();  // INVALID, an abstract class never gets instantiated
```

An abstract class can still have a constructor, which will be called by its subclasses during their construction.

The abstract class

An abstract class's purpose is to define the behavior its subclasses are required to have, so it always participates in **inheritance**.

For the examples on this slide, assume that Animal is an abstract class.

Classes extend abstract classes and can be concrete.

Here, Dog extends Animal. Animal is abstract, but Dog is concrete.

```
class Dog extends Animal {}  // Animal is abstract, Dog is not
```

The abstract class

A class that extends an abstract class can also be abstract itself, as I show with this next example.

Mammal is declared abstract, and it extends Animal, which is also abstract.

```
abstract class Mammal extends Animal {}  // Animal is abstract, Mammal is also
                                         // abstract
```

And finally an abstract class can extend a concrete class.

Here we have BestOfBreed, an abstract class, extending Dog, which is concrete.

```
abstract class BestOfBreed extends Dog {}  // Dog is not abstract, but
                                           // BestOfBreed is
```

What's an abstract method?

An abstract method is declared with the modifier **abstract**.

You can see on this slide, that we're declaring an abstract method called move with a void return type.

It simply ends with a semi-colon.

It doesn't have a body, not even curly braces.

```
abstract class Animal {

    public abstract void move();
}
```

Abstract methods can only be declared on an abstract class or interface.

What good is an abstract method, if it doesn't have any code i

An abstract method tells the outside world that all Animals will move, in the example I show here.

```
abstract class Animal {

    public abstract void move();
}
```

Any code that uses a subtype of Animal, knows it can call the move method, and the subtype will implement this method with this signature.

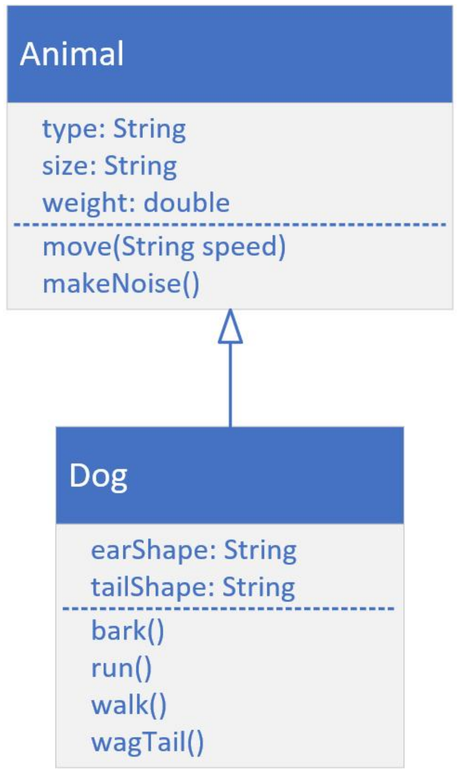
This is also true for a concrete class, and a concrete method that's overridden.

You might be asking, what's the difference, and when would you use an abstract class.

Animal and Dog Class Diagram from our Inheritance exa

In the videos on **inheritance**, we created a very basic Animal class, and then we extended it to create a Dog.

This is the class diagram from that video again.



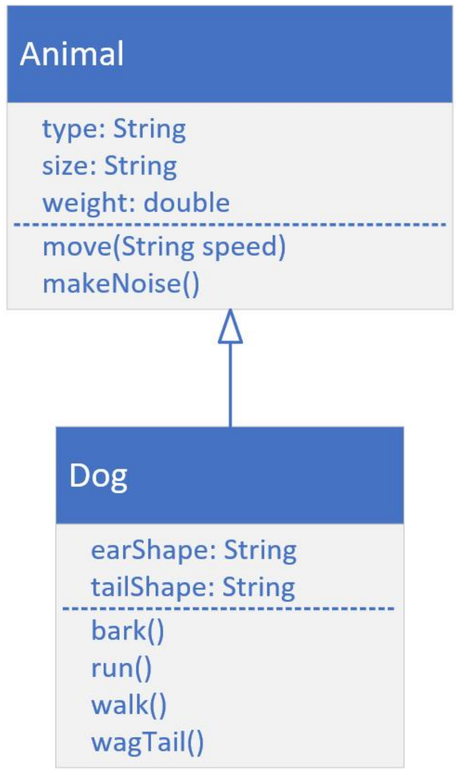
In that example, Animal was a concrete class, so the move and makeNoise methods had code in their method bodies.

Subclasses have choices when they extend a concrete class with concrete methods.

- They can inherit the same behavior from their parent. This means they don't have to even declare the methods in their class bodies.
- They can override the behavior from their parent. This means they have a method with the same signature, but with their own code in there, ignoring the parent's code altogether.
- They can also override the behavior but leverage the parent's method code, by calling the parent's method, using super in their overridden code.

Animal and Dog Class Diagram, What if Animal were abs

What happens if Animal is declared as abstract, and the move and makeNoise methods are also abstract?



If Animal is abstract and it's methods are abstract, subclasses no longer have the options I just talked about.

There is no concrete method for a subclass to inherit code from.

Instead, the subclass must provide a concrete method for any abstract method declared on its parent.

The subclass won't compile if it doesn't implement the abstract methods.

An Abstract class doesn't have to implement abstract met

An abstract class that extends another abstract class has some flexibility.

- It can implement all of the parent's abstract methods.
- It can implement some of them.
- Or it can implement none of them.
- It can also include additional abstract methods, which will force subclasses to implement both Animal's abstract methods, as well as Mammal's.

Why use an abstract class?

In truth, you may never need to use an abstract class in your design, but there are some good arguments for using them.

An abstract class in your hierarchy forces the designers of subclasses to think about, and create unique and targeted implementations, for the abstracted methods.

It may not always make sense to provide a default, or inherited implementation of a particular method.

An abstract class can't be instantiated, so if you're using abstract classes to design a framework for implementation, this is definitely an advantage.

Why use an abstract class?

In our example, we don't really want people creating instances of Animals or Mammals.

We used those classes to abstract behavior at different classification levels.

All Animals have to implement the move and makeNoise methods, but only Mammals needed to implement shedHair, as I demonstrated.

Interface vs. Abstract Class

You saw that an abstract class requires its subclasses to implement its abstract methods.

An **interface** is similar to an abstract class, although it **isn't a class** at all.

It's a **special** type, that's more like a **contract** between the class and client code, that the compiler enforces.

By declaring it's using an interface, your class must implement all the abstract methods on the interface.

A class agrees to this because it wants to be **known by that type**, by the outside world or client code.

An **interface** lets **classes that might have little else in common** be recognized as a special reference type.

Declaring an interface

Declaring an interface is similar to declaring a class, using the keyword **interface**, where you would use the keyword **class**.

On this slide, I'm declaring a public interface named FlightEnabled.

```
public interface FlightEnabled {}
```

An interface is usually named according to the set of behaviors it describes.

Many interfaces will end in 'able', like Comparable, and Iterable. Meaning something is capable or can perform a given set of behaviors.

Using an interface

A class is associated to an interface by using the **implements** clause in the class declaration.

In this example, the class Bird implements the FlightEnabled interface.

```
public class Bird implements FlightEnabled {

}
```

Because of this declaration, we can use FlightEnabled as the reference type and assign it an instance of bird.

In this code sample, I create a new Bird object but assign it to the FlightEnabled variable named flier.

```
FlightEnabled flier = new Bird();
```

A class can use extends and implements in same declarat

A class can only **extend** a **single class**, which is why Java supports only single inheritance.

However, a class can **implement many interfaces**, providing flexibility and modularity. This allows for the combination of different sets of behaviors, making interfaces a powerful feature.

A class can **both extend** another class and **implement** one or more interfaces.

```
package dev.lpa;

public class Bird extends Animal implements FlightEnabled, Trackable {
}
```

In this example, the Bird class extends or inherits from Animal, but it's implementing both a FlightEnabled, and Trackable interface.

We can describe Bird by what it is and what it does.

The abstract modifier is implied on an interface

I don't have to declare the interface type abstract, because this modifier is implicitly declared for all interfaces.

```
abstract interface FlightEnabled {  // abstract modifier here is unnecessary
                                    // and redundant
```

Likewise, I don't have to declare any method abstract.

In fact, any method declared without a body, is really **implicitly declared both public and abstract**.

The three declarations shown on this slide, result in the same thing, under the covers:

```
public abstract void fly();     // public and abstract modifiers are redundant,
                                // meaning unnecessary to declare
abstract void fly();            // abstract modifier is redundant, meaning
                                // unnecessary to declare
void fly();                     // This is PREFERRED declaration, public and
                                // abstract are implied.
```

All members on an interface are implicitly public

If you omit an access modifier on a **class member**, it's **implicitly package private.**

If you omit an access modifier on an **interface member**, it's **implicitly public.**

This is an important difference, and one you need to remember.

Changing the access modifier of a method to **protected** on an interface, is **a compiler error**, whether the method is concrete or abstract.

Only a concrete method can have private access.

The Bird Class

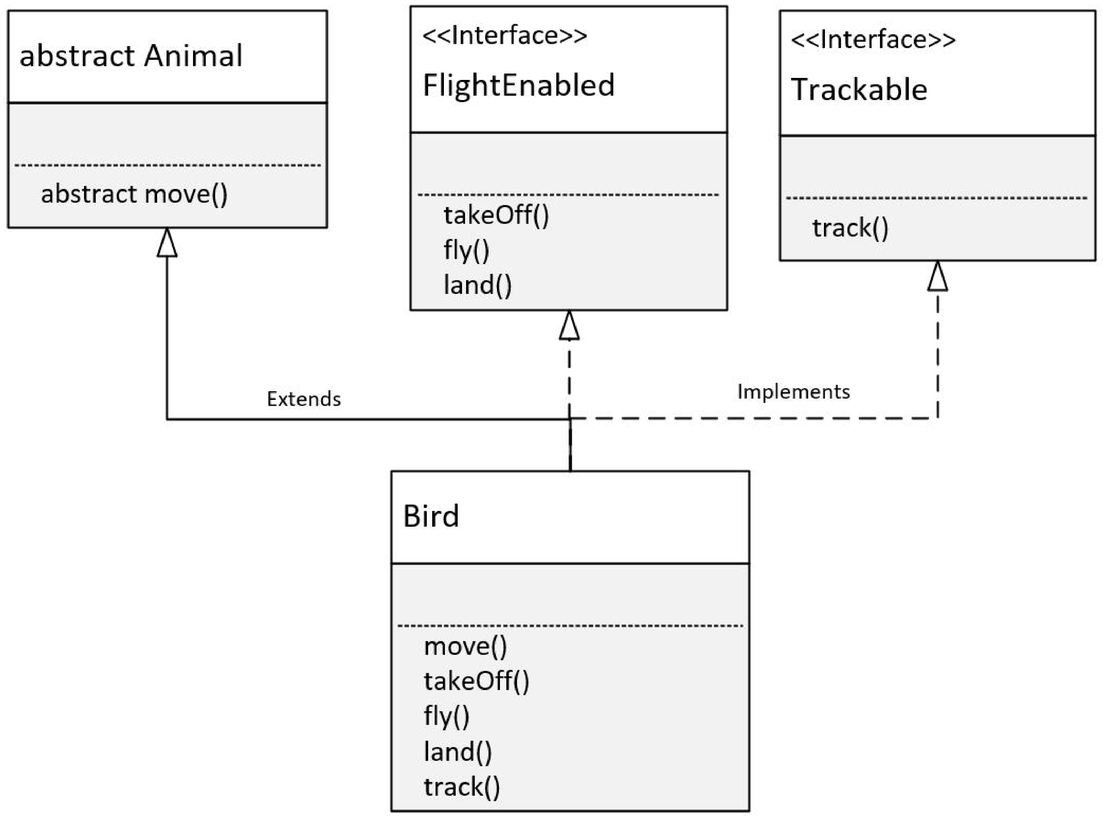
An interface lets us treat an instance of a single class as many different types.

The Bird Class inherits behavior and attributes from Animal, because I used the extends keyword in the declaration of Bird.

Because the move method was abstract on Animal, Bird was required to implement it.

The Bird Class implements the FlightEnabled interface.

This required the Bird class to implement the takeOff, fly, and land methods which were the abstract methods on FlightEnabled.



The Bird Class

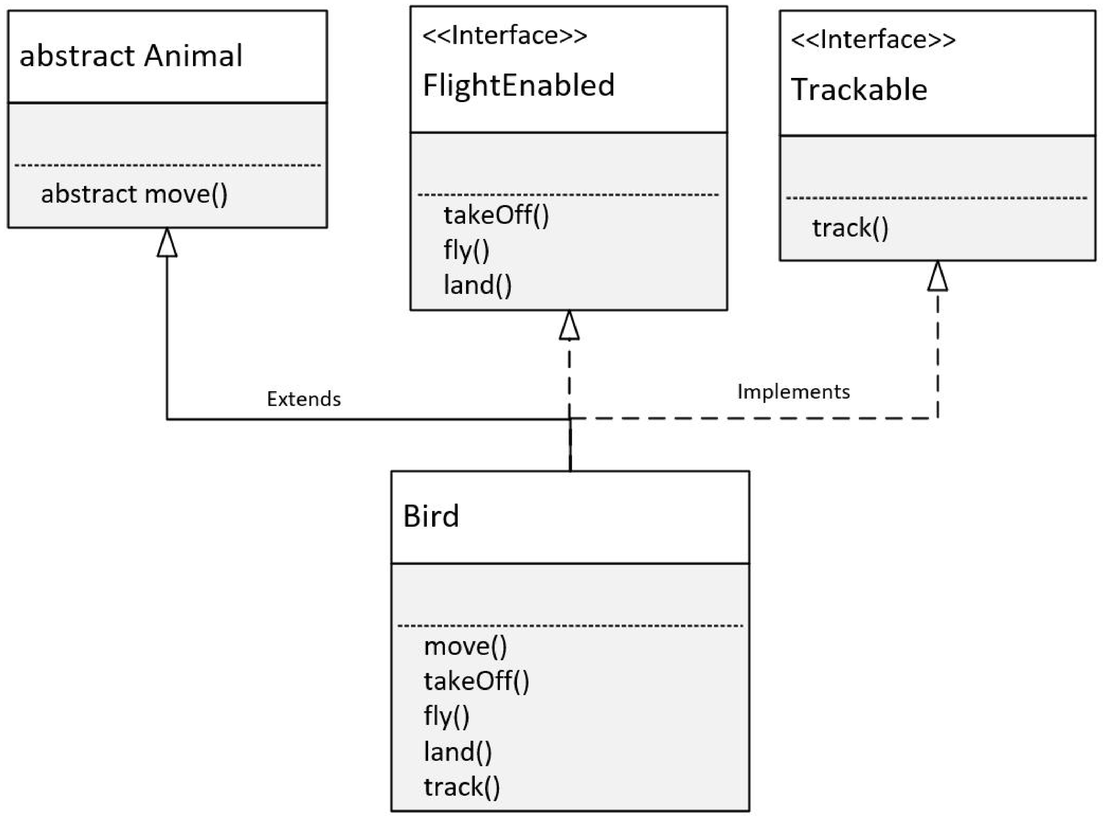
The Bird Class also implements the Trackable interface.

This required the Bird class to implement the track method, which was the abstract method declared on Trackable.

Because of these declarations, any instance of the Bird class can be treated as a Bird.

This means it has access to all of Bird's methods, including all those from Animal, FlightEnabled, and Trackable.

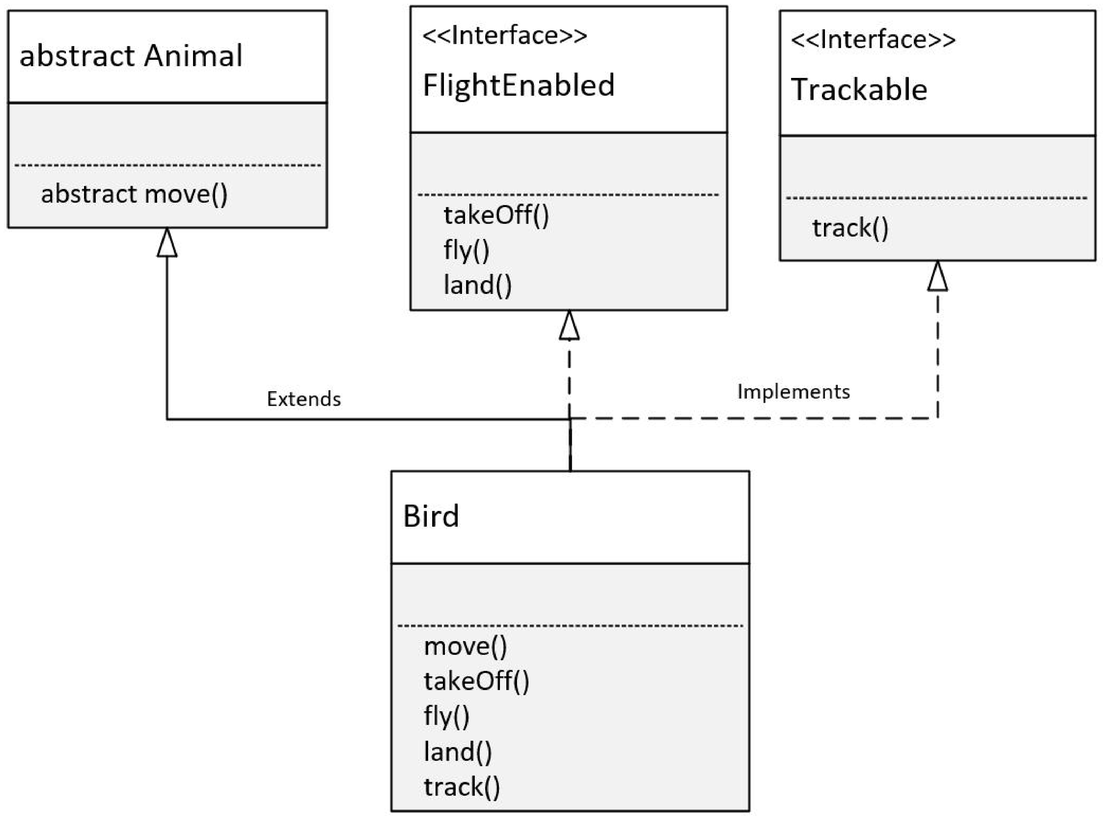
An instance of Bird can be treated like or declared as an Animal, with access to the Animal functionality described in that class, but customized to Bird.



The Bird Class

It can be used as a FlightEnabled type, with just the methods a FlightEnabled type needs but again customized for the Bird.

Or it can take the form of a Trackable object, and be tracked with specifics for the Bird class.



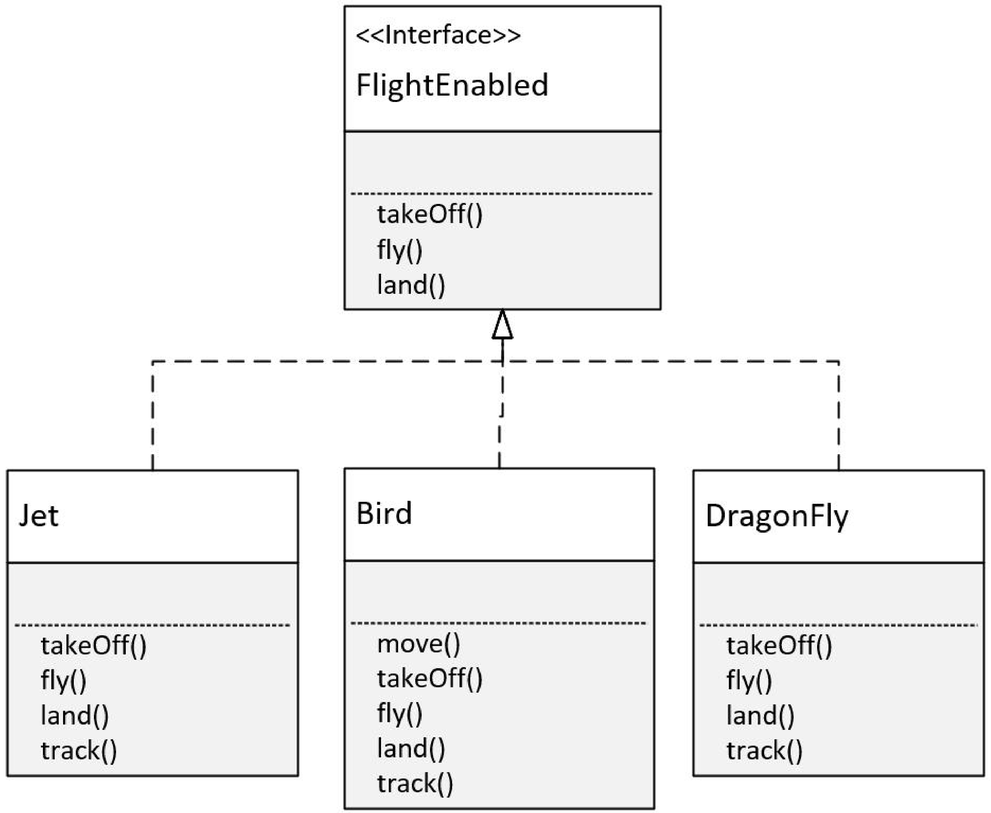
The FlightEnabled Interface

Interfaces let us take objects that may have almost nothing in common, and write reusable code so we can process them all in a similar manner.

On this slide, you can see that a Jet, a Bird, and a DragonFly are very different entities.

But because they implement FlightEnabled, we can treat them all as the same type, as something that flies, and ignore the differences in the classes.

Interfaces allow us to type our objects differently, **by behavior only**.



The final modifier in Java

When we use the final modifier, we prevent any further modifications to that component.

- a final method means it can't be overridden by a subclass.
- a final field means an object's field can't be reassigned or given a different value after its initialization.
- a final static field is a class field that can't be reassigned or given a different value after the class's initialization process.
- a final class can't be overridden, meaning no class can use it in the extends clause.

The final modifier in Java

- a final variable in a block of code, means that once it's assigned a value any remaining code in the block can't change it.
- a final method parameter means we can't assign a different value to that parameter in the method code block.

The final static field, is what you're really creating, when you declare a field on an interface.

Constants in Java

A constant in Java is a variable that can't be changed.

A *constant variable* is a final variable of primitive type, or type String, that is initialized with a constant expression.

Constants in Java are usually named with all uppercase letters and with underscores between words.

A static constant means we access it via the type name.

We saw this with the INTEGER.MAX_VALUE, and the INTEGER.MIN_VALUE fields.

A field declared on an Interface is always public, static and

Java lets us specify these like an ordinary field on an interface, which might be kind of confusing and misleading to a new Java programmer.

But we can declare them with any combination of those modifiers, or none at all with the same result.

These all mean the same thing on an interface.

```
double MILES_TO_KM = 1.60934;
final double MILES_TO_KM = 1.60934;
public final double MILES_TO_KM= 1.60934;
public static final double MILES_TO_KM = 1.60934;
```

Extending Interfaces

Interfaces can be extended, similar to classes using the extends keyword.

On this slide, I show an interface, OrbitEarth that extends the FlightEnabled interface.

This interface requires all classes to implement both the OrbitEarth and the FlightEnabled abstract methods.

```
interface OrbitEarth extends FlightEnabled {}
```

Unlike a class, an interface can use the extends expression with multiple interfaces:

```
interface OrbitEarth extends FlightEnabled, Trackable {}
```

Implements is invalid on an interface

An interface doesn't implement another interface, so the code on this slide won't compile.

In other words, implements is an invalid clause in an interface declaration.

```
interface OrbitEarth implements FlightEnabled {}  // INVALID, implements is
                                                   // invalid clause for
                                                   // interfaces
```

Abstracted Types - Coding to an Interface

Both interfaces and abstract classes are **abstracted reference types**.

Reference types can be used in code, as variable types, method parameters, return types, list types, and so on.

When you use an abstracted reference type, this is referred to as **coding to an interface**.

This means your code doesn't use specific types, but rather, more generalized ones, usually an interface type.

This technique is preferred, because it allows many runtime instances of various classes to be processed uniformly by the same code.

It also allows for substitutions of some other class or object that still implements the same interface, without forcing a major refactor of your code.

Using interface types as the reference type is considered a best practice.

Coding to an Interface

Coding to an interface scales well, to support new subtypes, and it helps when refactoring code, as I've just showed.

The downside though is that alterations to the interface may wreak havoc, on the client code.

Imagine that you have fifty classes using your interface and you want to add an extra abstract method to support new functionality.

As soon as you add a new abstract method, all fifty classes won't compile.

Your code isn't backwards compatible with this kind of change to an interface.

Interfaces haven't been easily extensible in the past.

But Java has made several changes to the Interface type over time, to try to address this last problem.

What's happened to the Interface since JDK 8

Before JDK 8, the interface type could only have public abstract methods.

JDK 8 introduced the **default** and **public** static methods and JDK 9 introduced **private** methods, both static and non-static.

All of these new method types (on an interface) are concrete methods.

The Interface Extension Method - the default method (as of JD

An extension method is identified by the modifier **default**, so it's more commonly known as the default method.

This method is a **concrete** method, meaning it has a code block and we can add statements to it.

In fact, it has to have a method body, even if the body is just an empty set of curly braces.

It's a lot like a method on a superclass because we can override it.

Adding a default method doesn't break any classes currently implementing the interface.

Overriding a default method

Just like overriding a method on a class, you have three choices when you override a default method on an interface.

- You can choose not to override it at all.
- You can override the method and write code for it, so that the interface method isn't executed.
- Or you can write your own code, and invoke the method on the interface, as part of your implementation.

public static methods on an interface (as of JDK8)

Another enhancement that Java included in JDK 8, was support for public static methods on an interface.

Static methods don't need to specify a public access modifier, because it's implied.

When you call a public static method on an interface, you must use the interface name as a qualifier.

In the ArrayList lectures, you may remember I used two static helper methods, on the Comparator interface which were added in JDK 8.

These were Comparator.naturalOrder() and Comparator.reverseOrder().

Private methods (JDK 9)

JDK 9 gave us private methods, both static and not static.

This enhancement primarily addresses the problem of re-use of code within concrete methods on an interface.

A private static method can be accessed by either a public static method, a default method, or a private non-static method.

A private non-static method is used to support default methods and other private methods.

Abstract Class

- Abstract classes are very similar to interfaces. You can't instantiate either of them. Both types may contain a mix of methods declared with or without a method block.
- With abstract classes, you can declare fields that aren't static and final. Instance fields in other words.
- Also with abstract classes, you can use any of the four access modifiers for its concrete methods.
- You can also use all but the private access modifier for its abstract methods.
- An abstract class can extend only one parent class, but it can implement multiple interfaces.
- When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class.
- However, if it doesn't, then the subclass must also be declared abstract.

Use an Abstract class when...

You want to share code among several closely related classes (Animal for example, with fields, name, age...).

You expect classes that extend your abstract class to have many common methods or fields or require access modifiers other than public.

You want to declare non-static or non-final fields (for example, name, age). This enables you to define methods that can access and modify the state of an object (getName, setName).

You have a requirement for your base class to provide a default implementation of certain methods, but other methods should be open to being overridden by child classes.

Summary: An abstract class provides a common definition, as a base class, that multiple derived classes can share.

Interface

- An interface is just the declaration of methods, which you want some classes to have, it's not the implementation.
- In an interface, we define what kind of operation an object can perform. These operations are defined by the classes that implement the interface.
- Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the Java compiler.
- You can't instantiate interfaces, but they may contain a mix of methods declared with or without an implementation.
- All methods on interfaces declared without a method body, are automatically public and abstract.
- An interface can extend another interface.

Interface

- Interfaces are more flexible and can deal with a lot more stress on the design of your program, because they aren't part of the class hierarchy.
- A best practice way of coding, is commonly called Coding to an Interface.
- By introducing interfaces into your program, you're really introducing points of variation, at which you can plug in different implementations for that interface.
- **Summary: The interface decouples the "what" from the "how", and is used to make different types behave in similar ways.**
- Since Java 8, interfaces can now contain default methods. i.e. methods with implementation. The keyword **default** is used mostly for backwards compatibility. Public static methods were also introduced in Java 8.
- Since Java 9, an interface can also contain private methods, commonly used when default methods share common code.

Use an Interface when...

- You expect that unrelated classes will implement your interface. For example, two of Java's own interfaces, Comparable and Cloneable, can be implemented by many unrelated classes.
- You want to specify the behavior of a particular data type, but you're not concerned about who implements its behavior.
- You want to separate different behavior.

Interfaces are the used in many of Java's own featu

- I've briefly discussed some interfaces, like List and Queue, and their implementations, **ArrayList** and **LinkedList**. These are part of what Java calls it's **Collections Framework**.
- Interfaces are also the basis for many of the features that are coming up, for example **lambda expressions**, which were introduced in JDK8.
- Another example is **Java's database connectivity support, or JDBC**, built almost entirely with interfaces. The concrete implementation of methods is different for each database vendor, and comes in the form of JDBC drivers. This enables you to write all database code, without being concerned about the details of the database you're connected to.

Interface vs. Abstract Class

I've said that interfaces and abstract classes are both abstracted types, and abstracted types are used as reference types in code.

The table on this slide is a summary of the similarities and differences.

	Abstract Class	Interface
An instance can be created from it	No	No
Has a constructor	Yes	No
Implemented as part of the Class Hierarchy. Uses Inheritance	Yes (in extends clause)	No (in implements clause)
records and enums can extend or implement?	No	Yes
Inherits from java.lang.Object	Yes	No
Can have both abstract methods and concrete methods	Yes	Yes (as of JDK 8)
Abstract methods must include abstract modifier	Yes	No (Implicit)
Supports default modifier for it's methods	No	Yes (as of JDK 8)
Can have instance fields (non-static instance fields)	Yes	No
Can have static fields (class fields)	Yes	Yes - (implicitly **public static final**)

The interface Challenge

In this challenge, you'll be working on creating some mappable output.

In the past decade or so, maps have become part of so many applications.

Most things, when drawn on a map, fall into three categories, a point, a line, or a polygon or geometric shape.

The result of your code will be text that could be printed out to a file for exchanging data with a mapping application.

One such file is a specially formatted file, called geojson, which is a JSON file extended for geographical elements.

You don't have to know JSON or geojson to be successful at this challenge.

The interface Challenge

For this challenge, you'll simply create a String for every feature that will be mapped.

An example of such a String is shown on this slide.

```
"properties": { "name": "Sydney Opera House", "usage": "Entertainment"}
```

The interface Challenge

First, create a **Mappable** Interface.

The interface should **force classes to implement three methods**.

- One method should return a **label** (how the item will be described on the map).
- One should return a **geometry type** (POINT or LINE) which is what the type will look like on the map.
- The last should return an **icon type** (sometimes called a map marker).

The interface Challenge

In addition to the three methods described, the interface should also include:

- A constant String value called JSON_PROPERTY, which is equal to: "properties":{%s}. A hint here, using a text block will help maintain quotation marks in your output.
- Include a default method called toJSON() that prints out the type, label, and marker. I'll show examples shortly.
- A **static method**, that takes a Mappable instance as an argument. This method should print out the properties for each mappable type, including those mentioned above, but also any other fields on the business classes.

The interface Challenge

You'll also want to create two classes that implement this interface, a **Building** and **UtilityLine**.

- One class, in my case the Building, should have a geometry type of POINT, and one class should have a geometry type of Line. The UtilityLine class will be my example for a class that will be a LINE on a map.
- The Building will be shown on a city map, as a point with the icon and label specified and the Utility Line will be a line on the map.

The interface Challenge

Your final output should look something like I show on this slide.

You should output the geometry type, the icon information, and the label.

Here is an example for a building, including type, label, and marker, but also the building name and usage, which are fields on Building.


```
"properties": {"type": "POINT", "label": "Sydney Town Hall (GOVERNMENT)", "marker": "RED STAR", "name": "Sydney Town Hall", "usage": "GOVERNMENT"}
```


And here is an example for a fiber optic Utility line, so a LINE, a green dotted line, would get drawn for a fiber optic cable on College Street.


```
"properties": {"type": "LINE", "label": "College St (FIBER_OPTIC)", "marker": "GREEN DOTTED", "name": "College St", "utility": "FIBER_OPTIC"}
```


You can see that the properties are a comma delimited list in curly braces with the property or field name in quotes, then a colon, followed by the property value or field value, and that's also in double quotes.

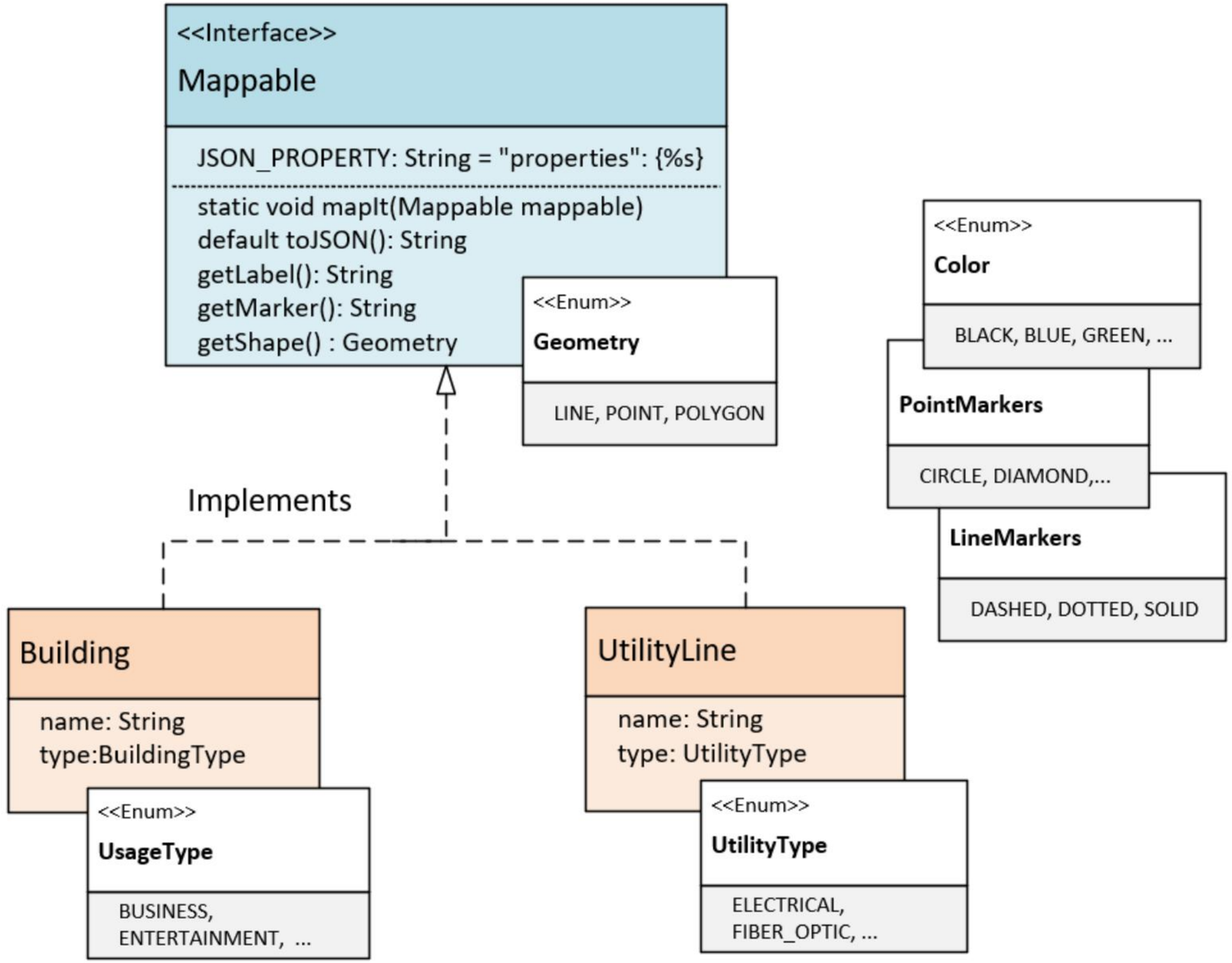
The Class Diagram

On the Mappable interface, I have one constant, JSON_PROPERTY, and 3 abstract methods, getLabel, getMarker, and getShape.

I'm also including a default method, toJSON, which is going to return a String.

The getShape method returns an enum type, Geometry, and the valid types on this enum are LINE, POINT, and POLYGON.

Use enums for Color, and the PointMarker and LineMarker types.



The Class Diagram

Two business classes to be mapped, Building and UtilityLine.

For Building, Use an enum to describe the building type.

For the UtilityLine, use enum to describe the type of utility it is.