NPTEL

noc24_cs43

Programming in Java

Live Interaction Session 8: 19th Mar 2024

Why does Java have primitive data types?

Some object-oriented languages, don't support any primitive data types at all, meaning everything is an object.

But most of the more popular object oriented languages of the day, support both primitive types and objects, as does Java.

Primitive types generally represent the way data is stored on an operating system.

Primitives have some advantages over objects, especially as the magnitude, or number of elements increase.

Objects take up additional memory, and may require a bit more processing power.

We know we can create objects, with primitive data types as field types, for example, and we can also return primitive types from methods.

Why don't all of Java's collection types support primitives?

But when we look at classes like the ArrayList, or the LinkedList, these classes don't support primitive data types, as the collection type.

In other words we can't do the following, creating a LinkedList, using the int primitive type.

This code won't compile.

```
LinkedList<int> myIntegers = new LinkedList<>();
```

This means, we can't use all the great functions Lists provide, with primitive values.

Why don't all of Java's collection types support primitives?

```
LinkedList<int> myIntegers = new LinkedList<>();
```

More importantly, we can't easily use primitives, in some of the features we'll be learning about in the future, like generics.

But Java, as we know, gives us wrapper classes for each primitive type.

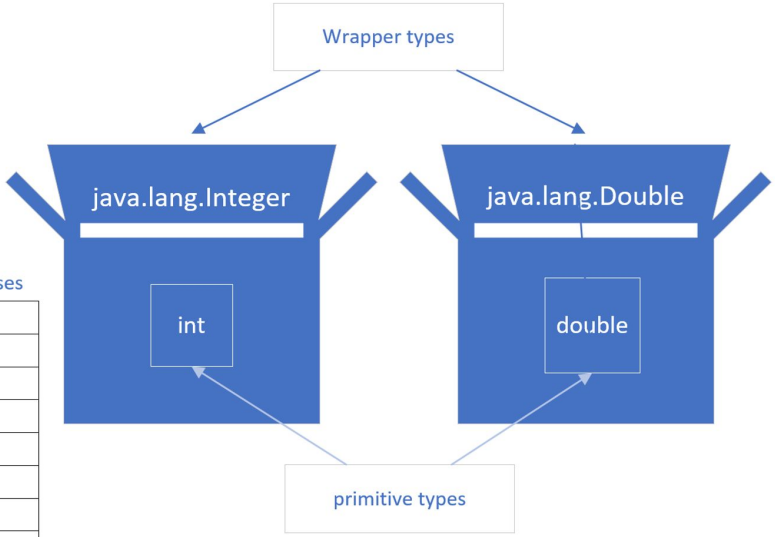
And we can go from a primitive to a wrapper, which is called boxing, or a wrapper to a primitive, which is called unboxing, with relative ease in Java.

What is Boxing?

A primitive is boxed, or wrapped, in a containing class, whose main data is the primitive value.

Each primitive data type has a wrapper class, as shown on the list, which we've seen before.

Each wrapper type boxes a specific primitive value.



Wrapper Classes

Boolean
Byte
Character
Double
Float
Integer
Long
Short

How do we box?

Each wrapper has a static overloaded factory method, valueOf(), which takes a primitive, and returns an instance of the wrapper class.

The code shown on this slide, returns an instance of the java.lang.Integer class, to the boxedInt variable, with the value 15 in it.

We can say this code **manually boxes** a primitive integer.

```
Integer boxedInt = Integer.valueOf(15);
```

Deprecated Boxing using the wrapper constructor

Another manual way of boxing, which you'll see in older code, is by creating a new instance of the wrapper class, using the new keyword, and passing the primitive value to the constructor.

We show an example of this here.

```
Integer boxedInt =  new Integer(15);
```

If you try this in IntelliJ, with any Java version greater than JDK-9, IntelliJ will tell you, this is deprecated code.

Deprecated Code

Deprecated code means it's older, and it may not be supported in a future version.

In other words, you should start looking for an alternate way of doing something, if it's been deprecated.

Using new (with a constructor) is deprecated for wrappers

```
Integer boxedInt =  new Integer(15);
```

Java's own documentation states the following:

It is rarely appropriate to use this constructor.

The static factory valueOf(int) is generally a better choice, as it is **likely to yield significantly better space and time performance**.

This deprecation applies to all the constructors of the wrapper classes, not just the Integer class.

In truth, we rarely have to manually box primitives, because Java supports something called **autoboxing**.

What is autoboxing?

We can simply assign a primitive to a wrapper variable, as we show on this slide.

```
Integer boxedInt = 15;
```

Java allows this code, and it's actually preferred, to manually boxing.

Underneath the covers, Java is doing the boxing. In other words, an instance of Integer is created, and it's value is set to 15.

Allowing Java to autobox, is preferred to any other method, because Java will provide the best mechanism to do it.

What is autoboxing?

```
Integer boxedInteger = 15;

int unboxedInt = boxedInteger.intValue();
```

Every wrapper class supports a method to return the primitive value it contains.

This is called unboxing.

In the example on this slide, we've autoboxed the integer value 15, to a variable called boxedInteger.

This gives us an object which is an Integer wrapper class, and has the valueof 15.

To unbox this, on an Integer class, we can use the intValue method, which returns the boxed value, the primitive int.

What is autoboxing?

```
Integer boxedInteger = 15;

int unboxedInt = boxedInteger.intValue();
```

This method is called **manually unboxing**.

And like boxing, it's unnecessary to manually unbox.

Automatic unboxing

```
Integer boxedInteger = 15;

int unboxedInt = boxedInteger.intValue();
```

Automatic unboxing is really just referred to as unboxing in most cases.

We can assign an instance of a wrapper class, directly to a primitive variable.

The code on this slide shows an example.

We're assigning an object instance to a primitive variable, in the second statement.

Automatic unboxing

```
Integer boxedInteger = 15;

int unboxedInt = boxedInteger.intValue();
```

This is allowed, because the object instance is an Integer wrapper, and we're assigning it to an int primitive type variable.

Again, this is the preferred way to unbox a wrapper instance.

Autoboxing Challenge with ArrayLists

Okay, so it's time for a challenge on autoboxing and unboxing.

In this challenge, you will need to create a simple banking application, with a Customer and Bank type.

The **Customer** will have a name, and an **ArrayList** of transactions containing **Double** wrapper elements.

A customer's transaction can be a credit, which means a positive amount, or it can be a debit, a negative amount.

Autoboxing Challenge with ArrayLists

The **Bank** will have a name, and an **ArrayList** of customers.

- The bank should **add a new customer**, if they're not yet already in the list.
- The bank class should allow a customer to **add a transaction**, to an existing Customer.
- This class should also **print a statement**, that includes the customer name, and the transaction amounts. This method should use unboxing.

Enumeration

The enum type is Java's type to support something called an enumeration.

Wikipedia defines enumeration as, “*A complete ordered listing of **all the items** in a collection*."

The enum type

Java describes the enum type as: A special data type that contains predefined constants.

A constant is a variable whose value can't be changed, once it's value has been assigned.

So an enum is a little like an array, except it's elements are known, not changeable, and each element can be referred to by a constant name, instead of an index position.

The enum type

```
public enum DayOfTheWeek {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

An enum, in its simplest form, is described like a class, but the keyword enum, replaces the keyword class.

You can name the enum with any valid identifier, but like a class, Upper CamelCase is the preferred style.

Within the enum body, you declare a list of constant identifiers, separated by commas. By convention, these are all uppercase labels.

One example of an enum, is the days of the week, as shown here.

The enum type

```
public enum DayOfTheWeek {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

An enum is ordered, by the way you declare the constants.

This means that SUNDAY is considered the first day of the week, and SATURDAY is the last day of the week.

The enum type

The enum type is used to declare a limited set of constants, and sometimes, there is a natural order to the listing, as in the case of days of the week.

Some other examples of possible enum declarations might be:

- The months in the year: JANUARY, FEBRUARY, MARCH, etc.
- The directions in a compass: EAST, NORTH, WEST, SOUTH.
- A set of sizes: EXTRA_SMALL, SMALL, MEDIUM, LARGE, EXTRA_LARGE.

The enum type

Underneath the covers, the enum type is a special type of class, which contains fields to support the constants, but we'll get into that, in a later discussion.

You don't have to understand all the internals of an enum, to derive the benefits of using this type.

Once you get used to how this type works, you may find many places to use an enum.

They simplify your code, and make it more readable in many ways.

Abstraction and Generalization

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

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

An extensible application is designed with change in mind.

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 would 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 let's us create abstract classes.
- Java gives us a way to create interfaces.

Abstract method

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.

It's 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

In addition to access modifiers, methods have other modifiers, which we'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 we'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. We'll cover this in a later section on multi-threaded code.

The abstract class

The abstract class is declared with the abstract modifier.

Here we are declaring 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 it's subclasses are required to have, so it always participates in **inheritance**.

For example,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.

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.

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 in it?

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

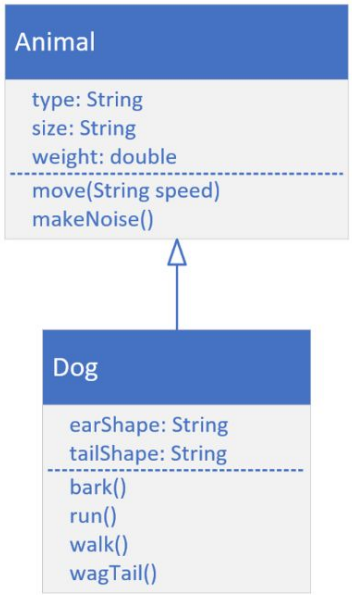
```
abstract class Animal {

    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.

So, what's the difference, and when would you use an abstract class.

Animal and Dog Class Diagram from Inheritance example



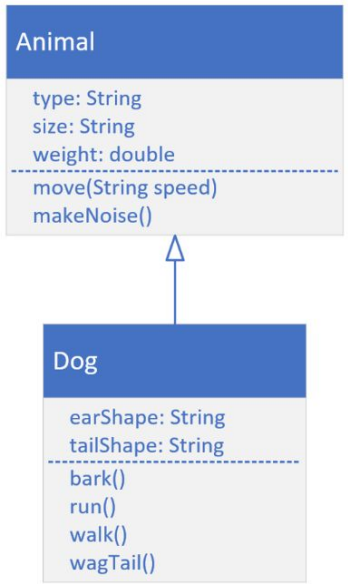
In that example, Animal was a concrete class, so the move and makeNoise methods had code in their method bodies.

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 abstract?

But 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 we 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 methods

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.

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 we demonstrated.

Abstract Class Challenge

In this challenge, you need to build an application, that can be a store front, for any imaginable item for sale.

Instead of the Main class we usually create, create a Store class, with a main method.

The **Store** class should:

- manage a **list of products for sale**, including displaying the product details.
- manage an order, which can just be a **list of OrderItem** objects.
- have methods to **add an item to the order**, and **print the ordered items**, so it looks like a sales receipt.

Abstract Class Challenge

Create a **ProductForSale** class that should have at least three fields: a **type**, **price**, and a **description**, and should have methods to:

- get a Sales Price, a **concrete method**, which takes a **quantity**, and **returns the quantity times the price**.
- print a Priced Line Item, **a concrete method**, which takes a **quantity**, and should **print an itemized line item** for an order, with **quantity and line item price**.
- show Details, an **abstract method**, which represents what might be **displayed** on a product page, **product type, description, price**, and so on.

Create an **OrderItem** type, that has at a minimum 2 fields, **quantity** and a **Product for Sale**.

You should create **two or three classes that extend the ProductForSale class**, that will be products in your store.

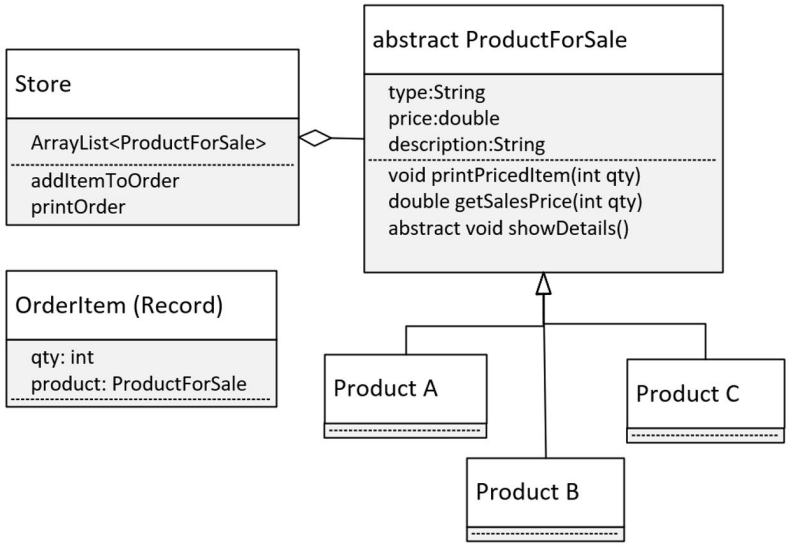
The Design

Let's look at one of the approach.

This covers all the requirements we talked about.

You'll notice I'm specifying that OrderItem will be a record, and this is just to keep the code simple.

And we are not really specifying what our store products are, we can really put anything there.



Interface vs. Abstract Class

We 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 the 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 class.

For example, 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, again meaning something is capable, or can do, 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, we create a new Bird object, but we assign it to the FlightEnabled variable named flier.

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

A class can use extends and implements in same declaration

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

But a class can **implement many interfaces**. This gives us **plug and play functionality,** which is what makes them so powerful.

```
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

We 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, we don't have to declare any method abstract.

In fact, any method declared without a body, is really i**mplicitly 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 we omit an access modifier on a **class member**, it's **implicitly package private.**

If we 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

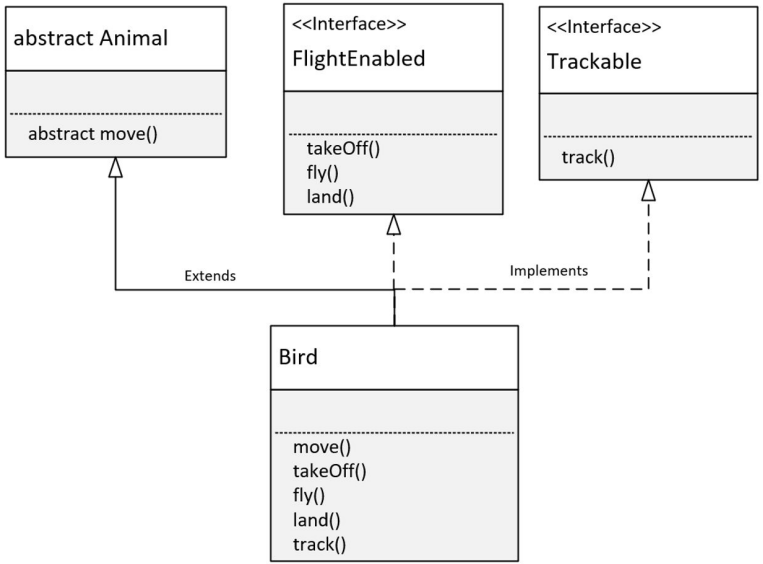
An interface lets us treat an instance of a single class as many different types.

The Bird Class inherits behavior and attributes from Animal, because we 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, the abstract methods on FlightEnabled.



The Bird Class

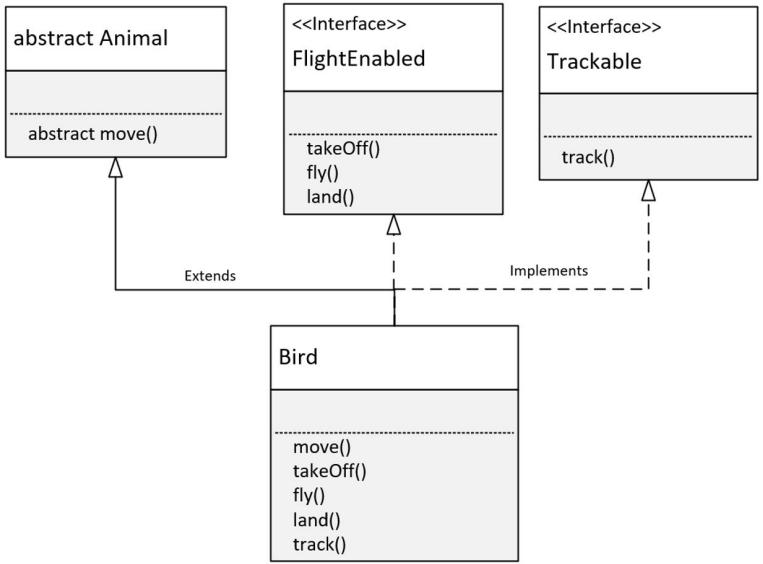
The Bird Class also implements the Trackable interface.

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.

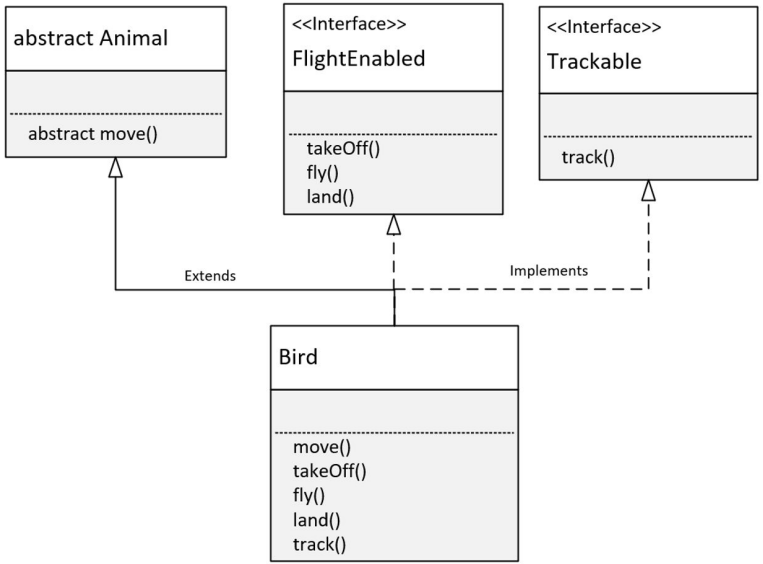
An instance of Bird can be treated like, or declared as an Animal, with access to the Animal functionality, described in that class, but customized to Bird.



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.



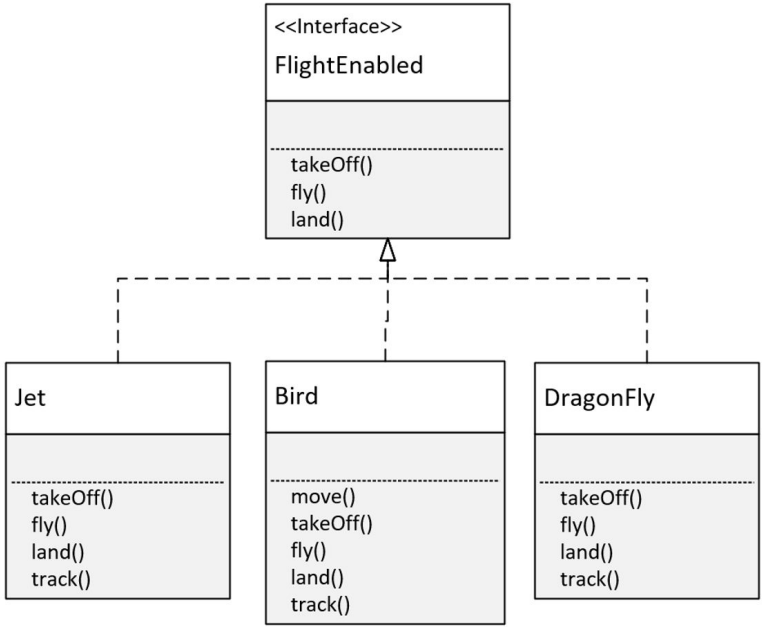
The FlightEnabled Interface

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

Java let's 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.

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, and 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 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.

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

Imagine that you have 50 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 50 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** method and public **static** methods, and JDK 9 introduced **private** methods, both static and non-static.

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

The Interface Extension Method - the default method (as of JDK8)

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 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

So 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 the 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.

Earlier you may have tried 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.

Interface vs. Abstract Class

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`**)