Generalization

When we start modeling objects for our application, we start by what features and behavior they 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 **Abstraction.** You can generalize a set of characteristics and behavior into an abstract type. For example, an octopus, a dog, and a penguin, you would probably say they are all animals. An animal is an **Abstract** concept. Java supports abstraction in many 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 also lets us create abstract classes
* Java gives us a way to create interfaces

Abstract class

An abstract class can extend an abstract class or a concrete class. An abstract class is declared with the abstract modifier

Public abstract class Animal{} //valid

Abstract class is incomplete and you cannot create an instance of an abstract class

Animal a = new Animal(); // invalid

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

An abstract class can extend a concrete class.

Class Dog extends Animal{} // Animal is abstract by Dog is not

An abstract class can extend another abstract class

Abstract class Mammal extends Animal {} Animal is abstract, Mammal is also abstract.

An abstract class cannot be directly instantiated using the new keyword. A concrete class can be directly instantiated using the new keyword

Concrete class can extends an abstract class

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

Abstract class can also have concrete methods.

An abstract class doesn’t have to implement flexibility

An abstract class extending another abstract class can do the following

* It can implement all of the parent’s abstract methods
* It can implement some of them
* It can implement none of them
* It can also include additional abstract methods, which will force to implement both abstract classes methods.

Abstract Method

An abstract method has a method signature, and a return type, but does not have a method body. Because of this we say 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. Think about an Abstract method as a contract. This contract promises that all subtypes will provide the promised functionality, with agreed upon name and arguments.

In a child class the override method can add statements.

Abstract class Animal {

Public abstract void move();

}

Abstract method tells the world every animal can 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.

**You cannot have a private access modifier on abstract method.**

Concrete Method

A concrete method has a method body, usually with at least one statement. It has operational code, that gets executed, under the right conditions. A concrete method is said to be **implemented** an abstract method, if it overrides one. Abstract classes and interfaces, can have a mix of abstract and concrete methods. An abstract class can have concrete methods

Method Modifiers

Access modifiers are public, protected, package, private access modifiers as an option for the members. Non-access modifiers are, abstract, static, final, default, native, synchronized.

**Public:** The class is accessible by any other class

**Default:** The class is only accessible by classes in the same package. This is used when you don't specify a modifier.

**Protected**: The code is accessible in the same package and **subclasses**.

**Private**: The code is only accessible within the declared class

Non- access modifiers

Abstract: method without a body, can only be declared on an abstract class, or an interface.

Static: called class method, it is directly called on the Class instance.

Final: cannot be overridden by subclasses.

Default: applicable to an interface

Native: implemented on platform dependent code.

Synchronized: used to manage how multiple threads will access the code in this method.

**Interface**

Abstract class required a sub class to implement its Abstract methods.

An interface is similar to an Abstract class but not a class at all.

It is a special type. More like a contract between the class and the client code. By declaring it using an interface, you must implement all the abstract methods, on the interface. A class agrees to this, because it wants to be known by that type, by outside world or the client code. An Interface lets classes that might have little less common, be organized as a special reference type.

**Declaration: public interface FlightEnabled{]**

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

**Using an Interface**

A class is associated to an Interface by using the implements clause in the declaration

Public class Bird implements FlightEnabled {

}

In the above example the class bird implements the Flight bird interface

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

FlightEnabled flier = new Bird();

In the above code sample, we create new Bird object, but we assign it to the FlightEnabled variable named flier.

We can use both extends and implements keywords in one class declaration. A class can extends only one class. That is why Java is called single inheritance. But a class can implements many interfaces. This gives the developer plug and play functionality. A class can both extends another class and implements one or more interfaces.

We don’t have to manually declare methods as abstract in an interface. Any method in an interface is implicitly declared both public and abstract.

**interface** FlightEnabled{  
  
 **public abstract void** takeoff(); public and abstract modifiers are redundant  
 **abstract void** land(); abstract modifier is redundant  
 **void** fly(); preferred declaration public and abstract are implied  
}

if we omit an access modifier on a class member, its implicitly package private.

If we omit an access modifier on an interface member, its implicitly public. Changing an access modifier to protected, on an interface, is a complier error, whether the method is abstract or concrete.

**Any fields declared in an Instance are implicitly public static and final which means they are really constants. Constants in JAVA are all uppercase letters with underscores between them for example MILES\_TO\_KM**

**Interfaces** can be extended like classes using the extends keyword

**interface OrbitEarth extends FlightEnabled {}**

OrbitEarth have abstract methods from both OrbitEart and FlightEnabled

Unlike class an interface can extends multiple interfaces

Interfaces does not implements other interfaces unlike classes.

Both enum and record can implement interfaces but they cant extend classes, abstract or not