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

abstract class Animal{}

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.