0:15 >> In this lesson you'll learn about a form of generalization. But first, let's discuss some important programming language and design notions. A class denotes a type for its objects. The type signifies what these objects can do through public methods. For example, instances of a dog class are dog typed objects, and these objects do dog things. In modeling a problem, we may want to express subtyping relationships between two types. For example, we can have dog type as a subtype of animal type. This means a dog object is not only dog typed, it is also animal typed. So a dog object behaves not only like a dog, it should also behave like an animal. In effect, a dog is an animal. In JAVA, class inheritance with the extends keyword is often used for subtyping. If a dog subclass extends an animal superclass, a dog object behaves not only like a dog, it will also behave by default like an animal through the inherited methods and attributes of an animal. In effect, a dog is an animal. Here, the dog class inherits the implementation details of animal class.

1:20 A JAVA interface also denotes a type. Unlike a class, however, an interface only declares method signatures, and no constructors, attributes, or method bodies. It specifies the expected behaviors in the method signatures, but does not provide any implementation details. In JAVA, an interface is also used for subtyping. If a dog class implements an I animal interface, then a dog object behaves not only like a dog, but it is also expected to behave like an animal by providing all the method bodies for the method signatures listed in the interface. Just like with inheritance, the dog is an animal. However, the difference is that the dog class needs to provide the implementation details for what it means to be an animal. So, an interface is like a contract to be fulfilled by implementing classes. In both inheritance and interfaces, you achieve consistency between the dog type and the animal type so that a dog object is usable anywhere in your program when you are dealing with an animal type. Unlike inheritance, interfaces are not a generalization of a set of classes. It is important to understand that interfaces are not classes. They are used to describe behaviors. All that an interface contains are method signatures. In JAVA, we use the key word interface to indicate that we are creating one. Standard JAVA naming convention places the letter I before an actual name to indicate an interface. This interface describes three different behaviors of an animal, which are moving, speaking, and eating. Notice how we never implement or describe how these behaviors are performed. We only show that an animal has these behaviors. Another thing you might have noticed is that the interface does not encapsulate any of the attributes of an animal. This is because attributes are not behaviors. Now that we have an interface, how do we use it? We need to declare that we are going to fulfill the contract as described in the interface. The keyword in JAVA for this action is implements. Our dog class has declared that it will implement or describe the behaviors that are in the interface. When you do this, you must have all the method signatures explicitly declared and implemented in the class. This means that we must the move, speak, and eat methods in this class.

3:34 Interfaces are drawn in a similar way that classes are drawn in UMLs. Interfaces are explicitly noted in UML class diagrams using guillemets, or French quotes, to surround the words interface. The interaction between an interface and a class that is implementing the interface is indicated using a dotted arrow. The class touches the tail end of the arrow and the interface touches the head of the arrow.

3:59 We combine these notations together with a class to show that a class implements an interface.

4:05 This indicates that the class implements the interface. The standard way to draw interfaces on your UML class diagrams is to have the arrow pointing upward. This means that the interface is always toward the top, and the classes that implement them are always toward the bottom.

4:20 If we translate the JAVA code to UML, the diagram for our animal interface example looks like this.

4:27 This UML class diagram tells us that the Dog class will determine how the behavior that is described in the interface IAnimal will be implemented by repeating the method signature.

4:37 There are several advantages for interfaces. Knowing and understanding what these advantages are will help you to determine if you should use interfaces or use inheritance when you are designing your systems.

4:49 Like abstract classes, which are classes that cannot be instantiated, interfaces are a means in which you can implement polymorphism. In object oriented languages, polymorphism is when two classes have the same description of a behavior, but the implementations of the behavior may be different. This can be seen when we compare a cat and a dog. How would you describe how each of these animals speak? Well, to simply put it, a cat meows and a dog barks. The description of the behavior is the same, both animals can

speak. But the actual behavior implementation itself is different. This is known as polymorphism. It is simple to achieve in JAVA using an interface.

5:29 We create our interface the same way as we did before.

5:33 The Cat and Dog class both implement the IAnimal interface, but they each have their own versions of the speak behavior. When we ask Doug the Dog to speak, he knows how to bark, but will not know how to meow like Mittens the Cat.

5:47 Just like with class inheritance, interfaces can inherit from other interfaces. And just like with class inheritance, interface inheritance should not be abused. This means that you should not be extending interfaces if you are simply trying to create a larger interface. Interface A should only inherit from interface B if the behaviors in interface A can fully be used as a substitution for interface B. A little confused? This example should clear things up.

6:15 Lets simplify the movement of a vehicle by restricting it s movement so that it can only travel along either the x-axis or y-axis. This interface can be used to describe the behaviors of vehicles on land or on water. But what if we need to implement the movement of a plane or a submarine that can also move in the zed-axis? We do not want to add an extra behavior to the interface, because on-land and on-water vehicles do not move along the zed-axis. So what do we do? We can create a second interface that will inherit from our first one.

6:44 Now, we can use the IVehicleMovement3D interface for all vehicles that have three-dimensional movement without having to add the Zed-axis movement to the interface used by the on-land and on-water vehicles.

6:58 To understand the next advantage of interfaces, we need to step back to inheritance. There is one other form of inheritance that we haven't looked at called multiple inheritance. This is when a subclass has two or more super classes. While this is possible to do with other object oriented languages, like C++, JAVA doesn't support Multiple Inheritance. This is because inheriting from two or more superclasses can cause Data Ambiguity. When your subclass inherits from two or more superclasses that have attributes with the same name or behaviors with the same method signature, how do you distinguish between them? Since JAVA cannot tell which one you would be referencing, it does not allow for multiple inheritance so that data ambiguity is not an issue. Interfaces do not run into this issue. In JAVA, a class can implement as many interfaces as we want. This is because of the nature of interfaces. Since they are only contracts and do not enforce a specific way to complete these contracts, overlapping method signatures are not a problem. A single implementation for multiple interfaces with overlapping contracts is acceptable. There is no ambiguity here because the Person class only has one definition of a speak method, and it is the same implementation for both interfaces. This is JAVA's approach to avoid the issue that is introduced with multiple inheritance.

8:14 Interfaces are powerful tool to allow you describe a set of behaviors. Classes can implement one or more interface at a time which allows them to have multiple types. Interfaces enable you to describe behaviors without the need to implement them, which allows you to reuse these abstractions. Just like with other constructs in object oriented modeling and programming, interfaces will help you to create programs with reusable and flexible code. Although they are a useful technique, remember that you should not be generalizing all behavior contracts into interfaces. They are meant to fulfill a specific need, which is to provide a way for related classes to work consistently.