

▼ Polymorphism in Java Means Inheritance AND/OR Implementation

- Classes "inherit from" superclasses
- Classes "implement" interfaces, a special kind of class
- Classes can *directly* have only one superclass, but can implement as many interfaces as they want.

▼ Inheritance (superclasses)

▼ In Java, classes "inherit from" the classes they *extend*

- If you don't specify otherwise, all classes you write extend `java.lang.Object`, which provides a default set of behaviors (addressing, low-level plumbing) for all objects to inherit.

▼ A class you write can extend from an *Abstract Class* or a *Concrete Class*

▼ abstract classes are classes that can never be instantiated (no "new AbstractThingy()").

- their only purpose is to provide some plumbing and / or an interface for concrete classes to use.
- They can consist of concrete and abstract methods: concrete methods actually "do stuff," whereas abstract methods act like methods in an interface -- a subclass must implement them.
- most classes are concrete classes. this means you can create instances of them and put data in and call methods on those instances.
- You can *override* the public behavior of any up-the-chain superclass by defining a method with the same signature, return value, and number and type of arguments. You can still get to the "old" one by explicitly calling "super.myMethod()".
- You inherit (meaning you have access to) the *protected* and *public* data members of your superclass, so you can manipulate them and any behavior in the superclass will use the data as it normally does.
- You inherit all the behavior of your superclass. Public methods on it can be called on instances of your class. In fact, public methods all the way up the inheritance chain (all the way to `Object`) can be called on your object.
- When you construct your class (`new MyClass()`), by default all zero-parameter constructors of all superclasses all the way up the chain get invoked (in up-the-chain order). You may explicitly invoke constructors in your direct superclass; you would do this if you wanted to invoke a constructor that takes arguments.

▼ Implementation (interfaces)

- In Java, classes "implement" interfaces they *implement*
- An interface includes *public static final* members (constants used by all implementing classes) and, more importantly for this discussion, *abstract* methods.
- An *abstract method* is a method that has no implementation. It specifies only a return type, a name for the method, and the number and type of arguments it takes.
- All classes that claim to "implement" the interface must *implement* all of its abstract methods. This means that they are no longer abstract. Your implementing class must have a real method in it with the return type, name, and number and type of arguments for each method so specified in the interface.

▼ Why does polymorphism matter?

- Polymorphism is just a fancy word for flexibility.

- ▼ Classes can be treated as instances of themselves, or as *instances of any interface they implement*, or as *instances of any class all the way up-the-chain* in their inheritance hierarchy.
 - If I write a custom linked list class called MyLinkedList, I might have it *extend* an existing OtherLinkedList class **and** *implement* the List interface.
 - My new class can be treated by Java methods and placed into Java reference variables that use any of the following types: MyLinkedList, OtherLinkedList, List, and Object (presuming that OtherLinkedList extends Object directly).
- ▼ Polymorphism via inheritance is a powerful feature. You can write data structures, for example, will take objects of a type that is far "up the chain," so that you can store objects of many different classes. That way, the data structure can hold any kind of data.
 - ▼ Look at the Java SDK. Most people are familiar with the "Vector" class, which is basically a dynamically growable array.
 - Note that Vector's "input" methods take arguments of type Object, and the "output" methods return Objects. This means that you can store any kind of Object in the Vector (meaning any class in Java, because all classes descend from Object).
 - This does not mean you *should* store objects of many different types in a single instance of Vector. In fact, you should almost always store only Objects of the same type in one Vector. It just means that Vector is *flexible* and can accommodate any Object.
 - Vector's use of Object is an example of *polymorphism via inheritance* in action. Because all objects *inherit from* Object, all objects can be stored by Vector.
- ▼ An example in the JDK of polymorphism via implementation is the java.lang.Comparable interface.
 - A class that implements Comparable must provide a *compareTo(Object)* method. This method is *specified* as an abstract method in the interface.
 - ▼ The object whose compareTo method is being called must determine if the incoming Object is less than, equal to, or greater than itself.
 - In most cases, the Object will be of the same class as the object the compareTo method was called on. Our object can then just compare its data with the incoming Object and decide the value of the comparison.
 - ▼ In other cases, the called object will be of a compatible but not identical type with the incoming Object. For this to work, we must follow two rules.
 - that both of them have a common superclass at some point in the inheritance chain (can even be Object if criterion #2 is met)
 - AND the data / methods used for comparison must exist in that "common denominator" superclass.
 - In still other cases (99% of the time due to a programmer making a mistake), the Object passed in is of a completely incompatible (and therefore not comparable) type.
 - This is a very useful interface for implementing sort algorithms, trees, linked list insertion algorithms, etc., because (as noted above in the discussion of Vector) usually you will be storing or sorting items of the same type, or a common supertype.
 - Your sort algorithm can consist of a method: *sort(Comparable[])* which would take in an array of class instances that implement the *compareTo(Object)* method and whose class claims to implement Comparable.

- ▼ If you want your *insert(Object)* method in your linked list to insert items in sorted order, can traverse itself when inserting, comparing the to-be-inserted item with each item, and inserting it at the proper point in the list.
- The only problem is, `Object` as a base class (remember, *insert(Object)* takes in an `Object`) only provides *equals(Object)* and *hashCode()*, which don't really do what we need to do in order to get a comparison.
 - The solution is, change *insert(Object)* to *insert(Comparable)* !!!! Now our linked list won't have to "figure out" how to "figure out" where to put the item in the list -- it can simply call *compareTo(Object)* on the to-be-inserted object, comparing it with the object in each node in the list.
 - See? You don't get *compareTo* from a just-plain-Object, but your linked list (what does it care if it's storing Objects or Comparables?? oh, yeah? did you ask it?) will get a lot of use out of storing `Comparable` instances instead.