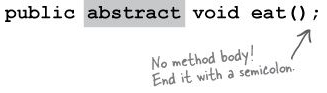
**B”H**

**Chapter 8**

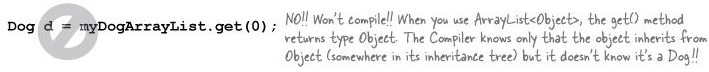
* To exploit polymorphism, we need interfaces
* An interface is a 100% abstract class. An abstract class is a class that can’t be instantiated. But can still be used for a reference type, for the purpose of polymorphism,
* Some classes just should not be instantiated! For example Animal in contrast to the subclass Wolf.
* By marking the class as abstract, the compiler will stop any code, anywhere, from ever creating an instance of that type – i.e. stop anyone from saying “new” on that type.
* A concrete class just means that it’s OK to make objects of that type.
* An abstract class has virtually no use unless it is extended.
  + There is an exception to this — an abstract class can have static members (see Chapter 10).
* An abstract method means the method must be overridden. An abstract method has no body.



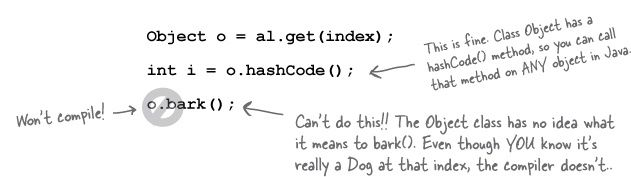
* If you declare an abstract method, you MUST mark the class abstract as well.
* But you can mix both abstract and non-abstract methods in the abstract class.
* Implementing an abstract method (in first concrete subclass) is just like overriding a method. Implementing here means to provide a body.
* The first concrete class in the inheritance tree must implement all abstract methods. You can, however, pass the buck by being abstract yourself.
* But remember that an abstract class can have both abstract and non-abstract methods, so Canine (which is abstract), could implement an abstract method from Animal, so that Dog didn’t have to.
* Think of the public methods in your class as your contract, your promise to the outside world about the things you can do
* Every class in Java extends class Object. Class Object is the mother of all classes; it’s the superclass of everything.
* Any class that doesn’t explicitly extend another class, implicitly extends Object.
* Some methods from the Object class:
  + equals( Object o)
    - Tells you if two objects are considered ‘equal’.
  + getClass()
    - Gives you back the class that object was instantiated from.
  + hashCode()
    - Prints out a hashcode for the object (for now, think of it as a unique ID).
  + toString()
    - Prints out a String message with the name of the class and the hashcode
* Some of Object’s methods are marked final, which means you can’t override them like getClass(),
* You’re encouraged (strongly) to override hashCode(), equals(), and toString() in your own classes, and you’ll learn how to do that a little later in the book.
* The Object class serves two main purposes:
  1. To act as a polymorphic type for methods that need to work on any class that you or anyone else makes
  2. To provide real method code that all objects need. Some of the most important methods in Object are related to threads, and we’ll see those later in the book.
* If you want to make an ArrayList that will literally take any kind of Object, you declare it like this:



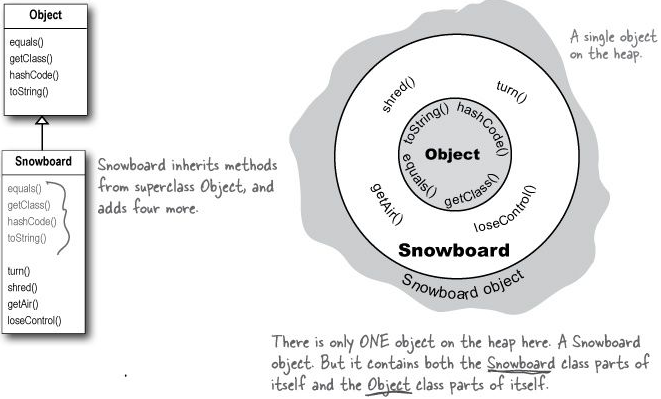
* But what happens when you try to get the Dog object and assign it to a Dog reference?



* The compiler decides whether you can call a method based on the reference type, not the actual object type.



* An object contains everything it inherits from each of its superclasses.

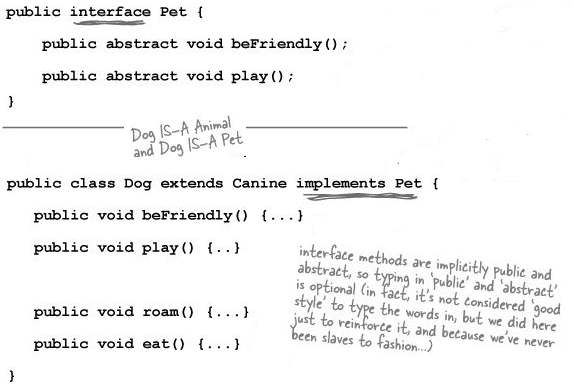


* If a reference is like a remote control, the remote control takes on more and more buttons as you move down the inheritance tree.
* Casting an object reference back to its real type. (You can use the instanceof operator to check):



**Interfaces**

* See pages 219 thru 223 on amazing example (using the Pet interface) why interfaces are needed.
* A Java **interface** solves the multiple inheritance problem by giving you much of the polymorphic benefits of multiple inheritance without the pain and suffering from the Deadly Diamond of Death (DDD).
* The way in which interfaces side-step the DDD is surprisingly simple: make all the methods abstract! That way, the subclass must implement the methods (remember, abstract methods must be implemented by the first concrete subclass), so at runtime the JVM isn’t confused about which of the two inherited versions it’s supposed to call.
* It defines only abstract methods.
* All interface methods are implicitly public and abstract



* Benefit of using Java interfaces:
  + Polymorphism, polymorphism, polymorphism. Interfaces are the ultimate in flexibility, because if you use interfaces instead of concrete subclasses (or even abstract superclass types) as arguments and return types, you can pass anything that implements that interface.
  + Classes from different inheritance trees can implement the same interface
  + The fact that you can’t put in implementation code turns out not to be a problem for most good designs, because most interface methods wouldn’t make sense if implemented in a generic way. In other words, most interface methods would need to be overridden even if the methods weren’t forced to be abstract.
  + For example; do you want an object to be able to save its state to a file? Implement the Serializable interface. Do you need objects to run their methods in a separate thread of execution? Implement Runnable. Etc.
  + A class can implement multiple interfaces



* Use an interface when you want to define a role that other classes can play, regardless of where those classes are in the inheritance tree.
* The keyword **super** lets you invoke a superclass version of an overridden method, from within the subclass.

