ALGONQUIN COLLEGE

MODULE 09: INTERFACES AND COLLECTIONS

Professor : Dave Houtman

Office: T323

Office Hrs: Friday 11:30 – 12:45

Email: houtmad@algonquincollege.com

9.0 What is an interface?

Each reference type falls into one of three categories, shown below:

a) (arrays)
b) Classes
c) Interfaces

Simple *arrays* form the first type. They may be thought of as primitive versions of full-blown *classes*, which make up the second category of reference types.

In this module we explore the last of these three types, **interfaces**.

9.0 What is an interface?

An interface is similar to an abstract class in almost every way, except in the following three aspects:

1. An interface is treated like a special kind of class in Java, one that—up until Java 8!—consisted *entirely of constants and abstract methods*. Therefore, being abstract,

   a) it cannot be instantiated. As with an abstract class, there's no such things as a `new` interface; and

   b) *all* of its abstract methods must be overridden in the first concrete subclass

2. Unlike a class, *you can have multiple interfaces*.

3. You **implement** an interface rather than extend it.

9.0 What is an interface?

The form of an interface is similar to a class, except the word `interface` replaces the word `class`:

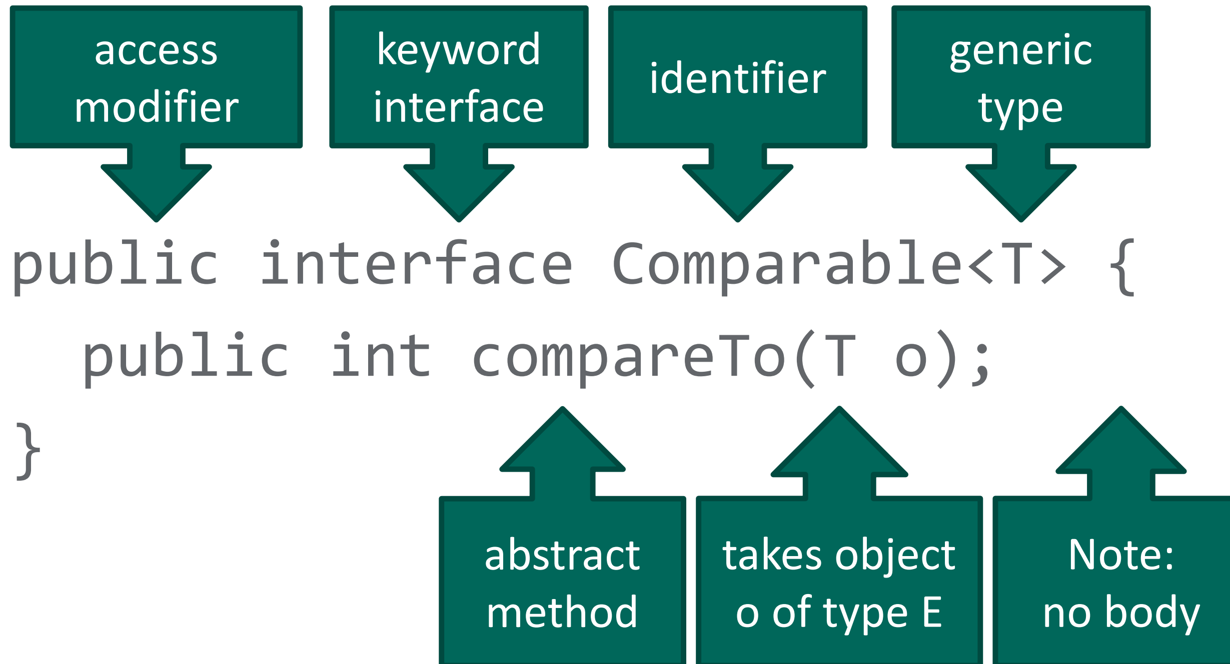
```
[access modifier] interface identifier{

   public static final fieldIdentifier1;    // constant properties
   public static final fieldIdentifier2;
   ...
   public abstract returnType methodIdentifier1();   //methods
   public abstract returnType methodIdentifier2();
   ...
}
```

ALGONQUIN COLLEGE

9.0 What is an interface?

Aside from the keyword `interface`, interface declarations look like abstract class declarations in virtually every other way. For example:



Note that interfaces are abstract by definition, hence there is no need to insert the word `abstract` in the interface header, or in the method header either.

9.1 Abstract Classes v. Interfaces

In almost every way, you can treat an interface like a class with abstract methods:

- As with abstract classes, you cannot instantiate a new object from an interface using the `new` operator;
- You must import interfaces the same way as you do for classes using, e.g. `import java.interfacename.*`
- You can use an interface as a data type in the declaration of identifiers, in arrays, and as parameters passed to functions
- Since *every* method in an interface is abstract,* *every* method must be overridden in the derived class…unless the derived class is abstract itself
- You can use an interface for casting
- Just as classes inherit via *class inheritance*, interfaces inherit via *interface inheritance*. However, since the two are so similar, we often refer to both as just *inheritance*, thus inferring that this may involve either classes *or* interfaces. But technically, rather than sub- and superclasses, we have subinterfaces and superinterfaces

* again, a reminder that this is only true up to Java 8

9.1 Abstract Classes v. Interfaces

There are a few notable differences between classes and interfaces:

- As previously stated, interfaces use the word `implements` rather than `extends`. For example, given `InterfaceA`, you can write:

```
public class MyNewClass implements InterfaceA {
    …
}
```

- You can both *extends* a class and *implements* an interface, e.g.

```
public class MyNewClass extends BaseClass
                                           implements InterfaceA {
    …
}
```

Thus `MyNewClass` inherits methods from both `BaseClass` and `InterfaceA`. And of course, it must override any abstract methods it inherits.

ALGONQUIN COLLEGE

9.1 Abstract Classes v. Interfaces

But note that the declaration

```
public class MyNewClass implements InterfaceA {
    …
 }
```

is equivalent to

```
public class MyNewClass extends Object
                                    implements InterfaceA{
    …
}
```

ALGONQUIN COLLEGE

9.1 Abstract Classes v. Interfaces

- You can add multiple interfaces by separating each interface with a comma:

```
public class MyNewClass implements InterfaceA, InterfaceB {
    …
}
```

As noted earlier, while a derived class can only extend from *one* base class, it can have *multiple* interfaces. Thus the problem of multiple inheritance *does not apply to interfaces.*

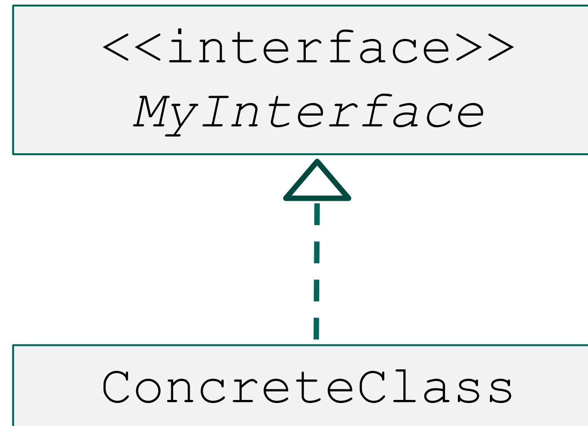
9.1 Abstract Classes v. Interfaces

- While interfaces are ‘implemented’ rather than ‘extended’, nonetheless the form of a bounded generic type for an interface uses ‘extends’, just as it would for a class. Thus

```
public class MyClass<T extends MyInterface>{
    //...
}
```

9.1 Abstract Classes v. Interfaces

- As with abstract classes, interface identifiers are italicized in UML. Unlike abstract classes, interface inheritance is symbolized using a dashed line



Note: the double angular brackets, `<<  >>` , known as *guimettes*, are sometimes used to signal additional information in UML diagrams. This use is called a **stereotype**; the guimettes signal that we wish to extend UML beyond its regular use. In this case, the notation `<<interface>>` reminds us that we are not dealing with a class.

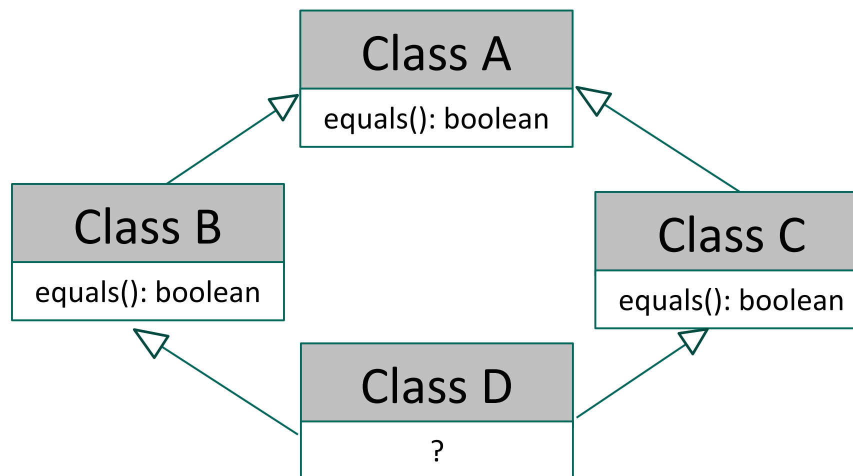
9.1 Abstract Classes v. Interfaces

In its original incarnation, prior to Java 8, interfaces were treated as the purest form of an abstract class—so pure, in fact, that they didn't actually define any code, but only declared specifications in derived classes.

Supertype	Must implement/extend to a concrete class before object instantiation?	Must override *every* base class method in the derived class?
A concrete class	NO	NO
An abstract class with no abstract methods	YES	NO
A class with at least one abstract method	YES	NO only the abstract method(s) in the base class
An interface	YES	YES

9.1 Abstract Classes v. Interfaces

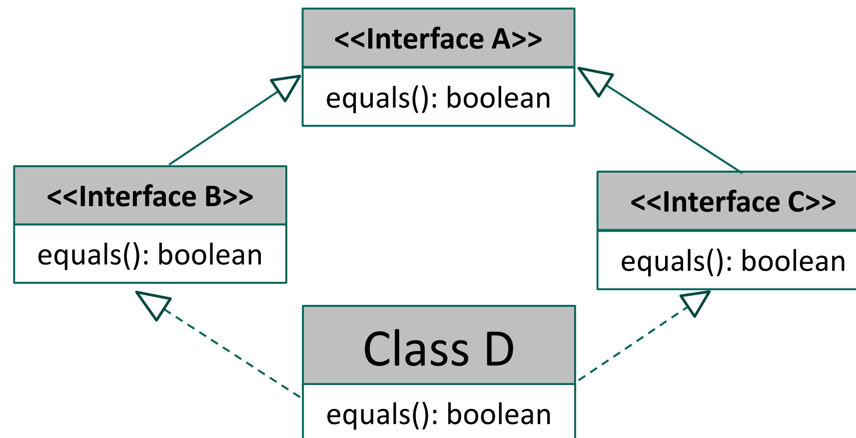
Given that interfaces and abstract classes are so much alike, a perfectly valid question to ask at this point is: what do you need interfaces for? The answer takes us back to Module 03. Recall that Java does not permit **multiple inheritance**, since it leads to the **diamond problem**:



In the diagram above, Class D inherits from both B and C, and both of the latter have overridden A's `equals()` method. But whose `equals()` method does D use?

9.1 Abstract Classes v. Interfaces

Interfaces solve the diamond problem: this is their chief benefit over abstract classes (which, after all, are restricted by **single inheritance**). In the diagram below, A, B, nor C are interfaces, and so contain no code. Class D doesn't need to know 'which' method to implement, since each interface `equals()` method is just a header, with no body to define the actual execution. Since there are no concrete methods to inherit from, D must supply its own code for `equals()`.



This was an awkward and overly-restrictive fix to the diamond problem, but it at least offered the programmer something approximating multiple inheritance.

9.1 Abstract Classes v. Interfaces

As of Java SE 8, you can now include static and non-abstract methods in interfaces. Java now includes the following 'non-conflict' rules, designed to eliminate the possibility of a 'diamond problem' occurring. These rules may be stated as follows:

1. *Classes (and superclasses) take higher precedence than interfaces, when duplicate method signatures are found in each*
2. *Derived interfaces have higher precedence that base interfaces*
3. *If rules 1 and 2 cannot resolve the conflict then the implementing class must override and provide a method with the same signature*

In other words, the new rules don't so much resolve potential problems as clarify those situations when they might be resolved automatically, thus allowing multiple inheritance in Java without the inherent risk of the diamond problem.

Whether these rules successfully solve the problem, and to what extent multiple inheritance is adopted by cautious Java programmers, remains to be seen.

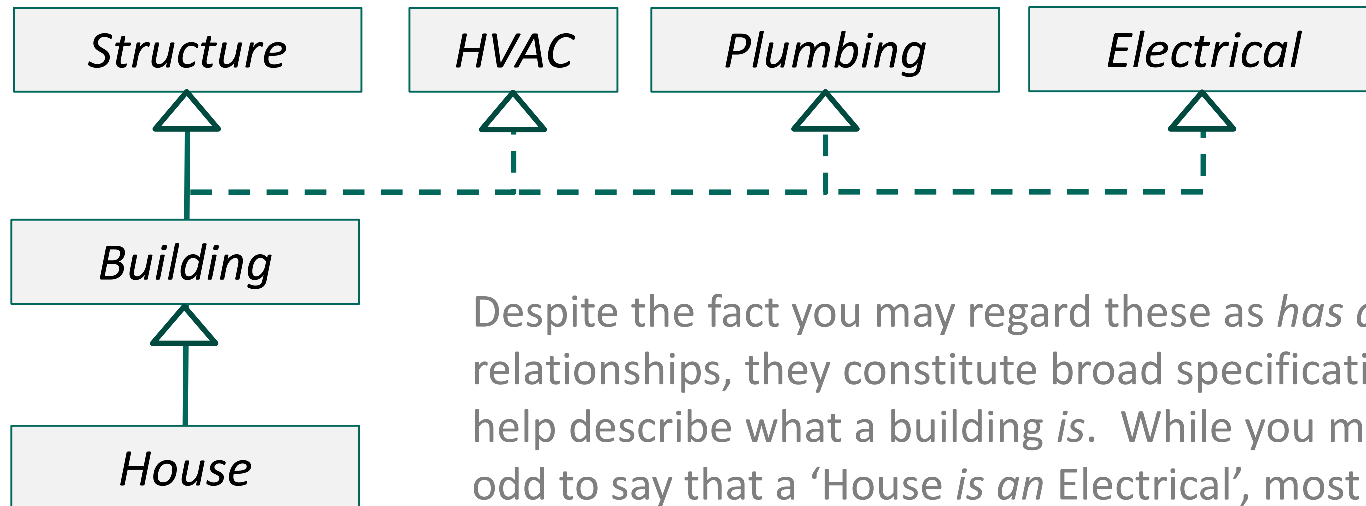
9.1 Abstract Classes v. Interfaces

Regardless, Java still contains both classes *and* interfaces. Since an interface looks just like a purely abstract class, a reasonable question to ask is: when should I use one and when should I use the other?

In general, use inheritance from a class when there is a strong *is a* relationship: a dog *is a* animal, a house *is a* building. Interfaces should be used when there is a weak *is a* relationship, or even an abstract *has a* relationship, especially when a broad specification or behaviour is involved.

9.1 Abstract Classes v. Interfaces

For example, a building *is a* structure. However, any building must have certain core features—such as heating, plumbing and electrical facilities—which, while not central to the definition of what a structure is, are essential to the definition of what a building *includes*. In the diagram below, the three interfaces straddle the boundary between an abstract *has a* feature and a weak *is a* (or *is part of a*) feature.

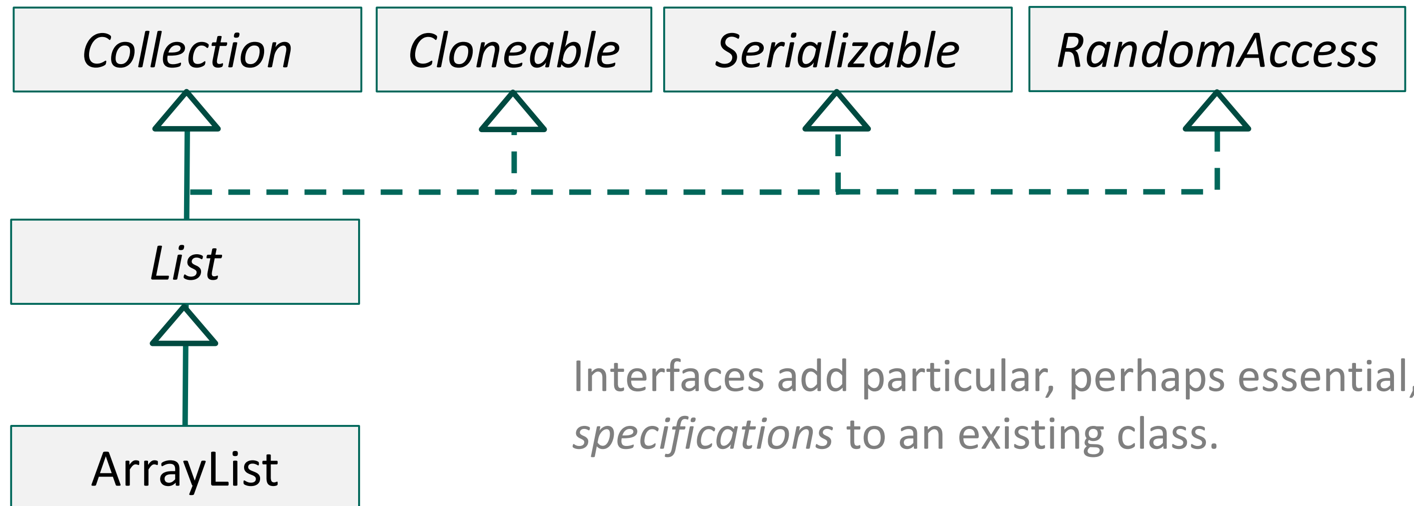


Despite the fact you may regard these as *has a* relationships, they constitute broad specifications that help describe what a building *is*. While you may find it odd to say that a 'House *is an* Electrical', most of us in the Western world would agree that a modern house lacking electrical outlets lacks an essential feature of 'houseness'.

9.1 Abstract Classes v. Interfaces

In the world of programming, interfaces ensure that certain features are *guaranteed* to appear in subclasses (since their abstract methods *must* be overridden).

So think of interfaces as containing the specifications that must be tacked on to existing classes in order to guarantee their suitability for a particular task.



Interfaces add particular, perhaps essential, *specifications* to an existing class.

9.1 Abstract Classes v. Interfaces

So the purpose of an interface is to declare a common set of behaviours for its derived classes, and therefore, any objects instantiated them. Essentially, an interface forms a contract that says: any concrete class that implements an interface will fulfill a particular set of requirements, as specified by the interface's methods.

If your code implements an interface,

then you must override all the interface's abstract methods with your own.