Topic 3

Abstract Classes and Interfaces

Portions of these notes and code examples are based on Introduction to Java Programming by Y. Daniel Liang, Copyright Prentice-Hall, 9th Edition 2013

Learning Outcomes

- **Describe** the purpose and application of Abstract classes and **demonstrate** how the `abstract` keyword is used in Java code
- **Demonstrate** the use of abstract classes for simple code reuse (i.e. no polymorphism)
- **Describe** and **demonstrate** in code the use of abstract classes for polymorphism
- **Describe** and **demonstrate** in code the use of Java interfaces for polymorphism
- **Explain** the difference between abstract classes and interfaces and the advantages/disadvantages of each approach
- **Describe** the limitations of multiple inheritance in Java and **demonstrate** in Java code how interfaces can be used to overcome these limitations
- **Explain** static versus dynamic types and **demonstrate** how to choose appropriate types in a class hierarchy when declaring variables and parameters
- **Demonstrate** in code how to avoid the use of `instanceof` by careful use of OO/Java typing
- **Describe** and **demonstrate** in code, delegation as an alternative to inheritance

The *abstract* modifier

- An `abstract` class
  - May have zero or more abstract methods
  - Cannot be instantiated
  - Is inherited and usually enhanced by subclasses
  - Cannot be instantiated using the `new` operator, but can define 1) constructors (to be invoked from subclass) and 2) methods to be invoked, implemented or overridden
- An `abstract` method
  - Method signature without implementation
  - Cannot be contained in a non-abstract class
  - If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be declared abstract

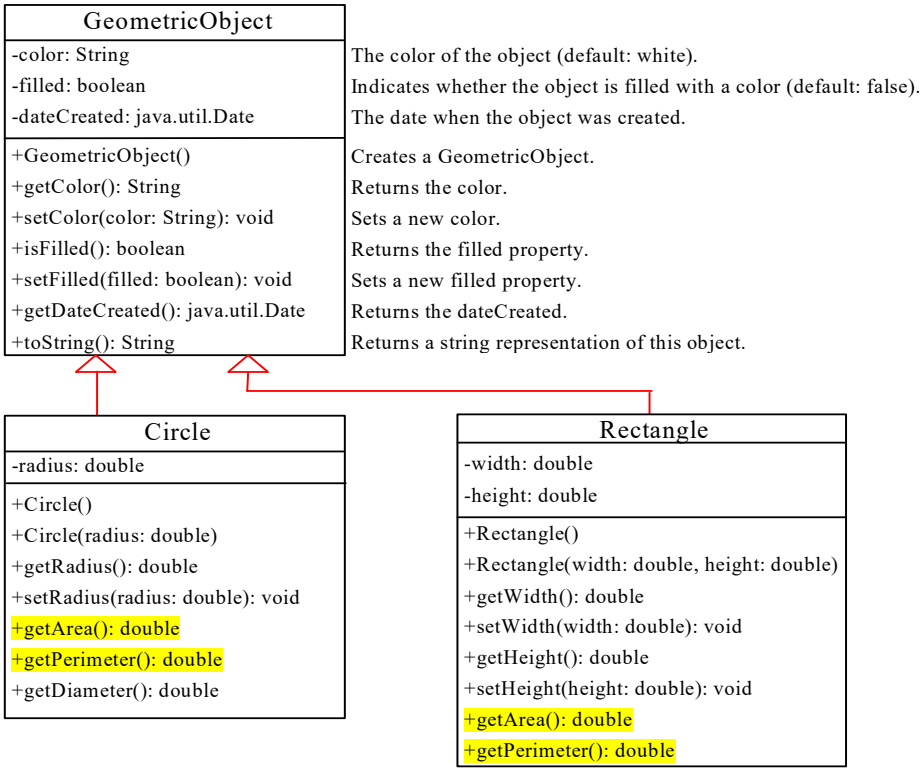
NOTE

A subclass can be abstract even if its superclass is concrete. For example, the `java.lang.Object` class is concrete, but its subclasses, such as `GeometricObject`, may be abstract.

You cannot create an instance from an abstract class using the `new` operator, but an abstract class can be used as a data type (declared/static type). Therefore, the following statement, which creates an array whose elements are of `GeometricObject` type, is correct:

```
GeometricObject[] geo = new GeometricObject[10];
geo[0]=new Circle(1.0);
```

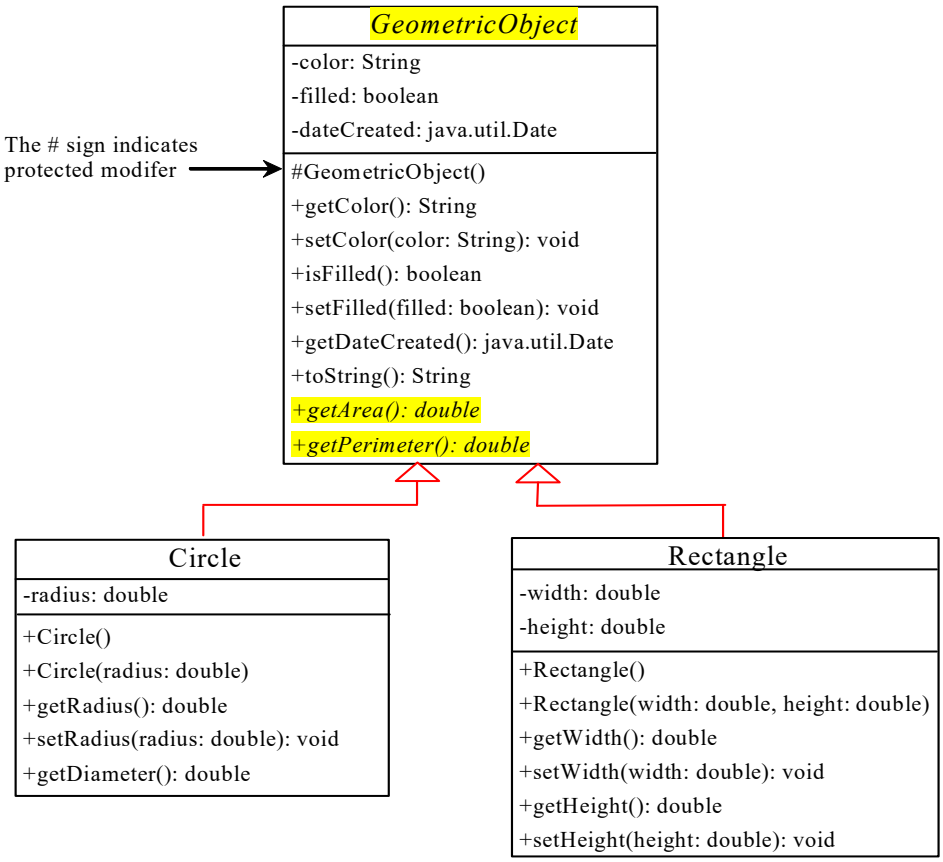
Example with no abstract classes (code reuse only from previous lecture)



LIMITATIONS

- In this example we cannot call the common methods such as `getArea()` polymorphically
  - Must treat each subtype differently and have separate collections/calls for each type
  - Or need `if` statements and `instanceof` checks every time we want to make calls (which is rarely ever necessary!) if you group them
  - leads to a lot of duplicated code/redundancy
- We could provide `getArea()` in `GeometricObject` but what would this mean and what would it return?

Example with abstract class



COMMON QUESTION ABOUT INSTANCEOF

- ***But what do I do when one of my subclasses has a specific method that doesn't exist in the rest of the hierarchy and I want to call it?***
- OPTION 1: Maintain a separate collection of that specialized type to call the specialized method (see Liang Chapter 10 example).
  - avoids having to use `instanceof`
  - but you do have to make sure objects get added to multiple collections when you create them (and remove later).
  - NOTE: is only a duplication of the reference so is not expensive to do this
- OPTION 2: instead of making the method(s) specific to the subclass create an interface and have the subclass implement that interface
  - then use OPTION 1 above but maintain a collection of the interface type instead
  - Interfaces are covered in more detail from next slide

Interfaces

An *interface* is a class-like construct that contains only constants and abstract methods. In many ways, an interface is similar to an abstract class, but an abstract class can contain variables and concrete methods as well as constants and abstract methods.

To distinguish an interface from a class, Java uses the following syntax to declare an interface:

```
public interface InterfaceName {
  constant declarations;
  method signatures;
}
```


Interface is a Special Class

An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class. Like an abstract class, you cannot create an instance from an interface using the new operator, but in most cases you can use an interface more or less the same way you use an abstract class. For example, you can use an interface as a data type for a variable or parameter type, as the result of casting, and so on.

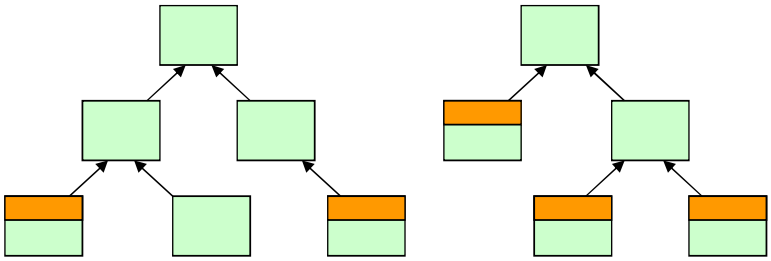
Interfaces: Non Hierarchical Commonality

Consider a set of class hierarchies, some of whose objects can be printed out to the screen.

We could **create an interface** defining how the "printable" functionality is to be implemented, and have these classes implement that interface.

The benefit is that we can then use that interface **as a data type**, much like we can with superclasses and polymorphism.

All `Printable` classes in the diagram can be referred to by their capability, even though they are completely unrelated in the hierarchy.



Example Using Interfaces

```
public interface Edible {
  /** Describe how to eat */
  public String howToEat();
}


class Animal {
}

class Chicken extends Animal
    implements Edible {
  public String howToEat() {
    return "Fry it";
  }
}

class Tiger extends Animal {
}
```

```
abstract class Fruit
    implements Edible {
}

class Apple extends Fruit {
  public String howToEat() {
    return "Make apple cider";
  }
}

class Orange extends Fruit {
  public String howToEat() {
    return "Make orange juice";
  }
}
```

Implements Multiple Interfaces

```
class Chicken extends Animal implements Edible, Comparable {
  int weight;
  public Chicken(int weight) {
    this.weight = weight;
  }
  public String howToEat() {
    return "Fry it";
  }
  public int compareTo(Object o) {
    return weight – ((Chicken)o).weight;
  }
}
```


Creating Custom Interfaces, cont.

```
public interface Edible {
  /** Describe how to eat */
  public String howToEat();
}

public class TestEdible {
  public static void main(String[] args) {
    Object[] objects = {new Tiger(), new Chicken(), new Apple()};
    for (int i = 0; i < objects.length; i++)
      showObject(objects[i]);
  }

  public static void showObject(Object object) {
    if (object instanceof Edible)
      System.out.println(((Edible)object).howToEat());
  }
}
```

This example from a textbook has some limitations. Try to identify and fix them?

Interfaces vs. Abstract Classes

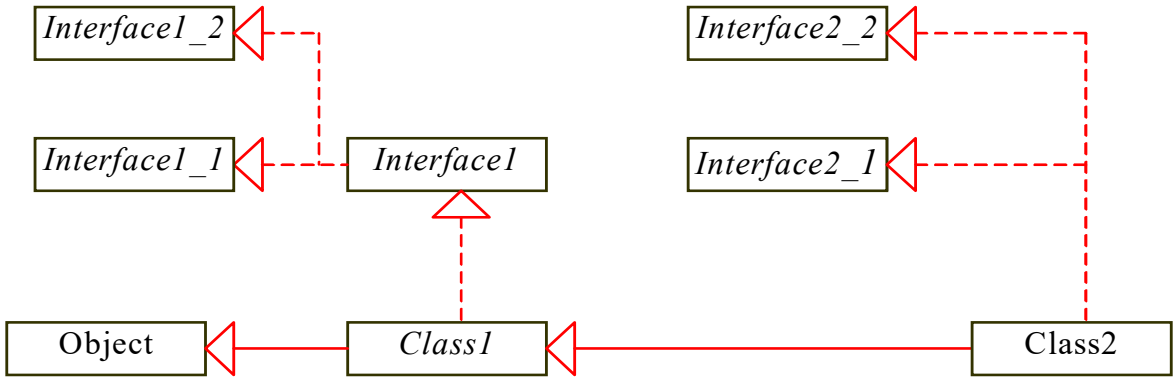
In an interface, the attributes must be constants (public final static) whereas an abstract class can have all types of data.

Each method in an interface has only a signature without implementation; an abstract class can have zero or more concrete methods with an implemented method body.

	Variables	Constructors	Methods
Abstract class	No restrictions	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be public static final	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

Interfaces vs. Abstract Classes, cont.

All classes share a single root, the Object class, but there is no single root for interfaces. Like a class, an interface also defines a type. A variable of an interface type can reference any instance of the class that implements the interface. If an interface extends an interface, this interface plays the same role as a superclass. You can use an interface as a data type for variables, parameters etc.



Suppose that c is an instance of Class2. c is also an instance of Object, Class1, Interface1, Interface1_1, Interface1_2, Interface2_1, and Interface2_2.

Design with Abstract Classes and Interfaces

Why abstract classes?

We want a generalised base class that is incomplete (and thus cannot be accidentally instantiated) but want to provide default behaviour (that may be extended or specialised later).

We want the flexibility to add methods later without changing existing clients.

Why interfaces?

- We want to treat different classes together in some specific way by means of a strict contractual interface (set of method signatures) but have no default behaviour and possibly no hierarchical relationship.
- We have many such behaviours and therefore the need for multiple inheritance which is only supported at the interface level in Java.

•We are prepared to 'lock down' the interface as a specification since changing it has significant implications on clients.

•If we do add methods we usually create a new interface that extends the original but also keep the original for backwards compatibility.

Design with Abstract Classes and Interfaces

Note that like regular inheritance abstract classes and interfaces ensure a uniform and structured design throughout the class hierarchy. Abstract classes, inheritance and interfaces, together with polymorphism and dynamic binding, promote code reuse by *allowing methods to be called in a generalised way*. It also facilitates **maintainability by placing specialised code in a single place**.

Whether to use an interface or a class?

In general for maximum extensibility it is preferable to use **both** interfaces and classes together rather than choose one or the other.

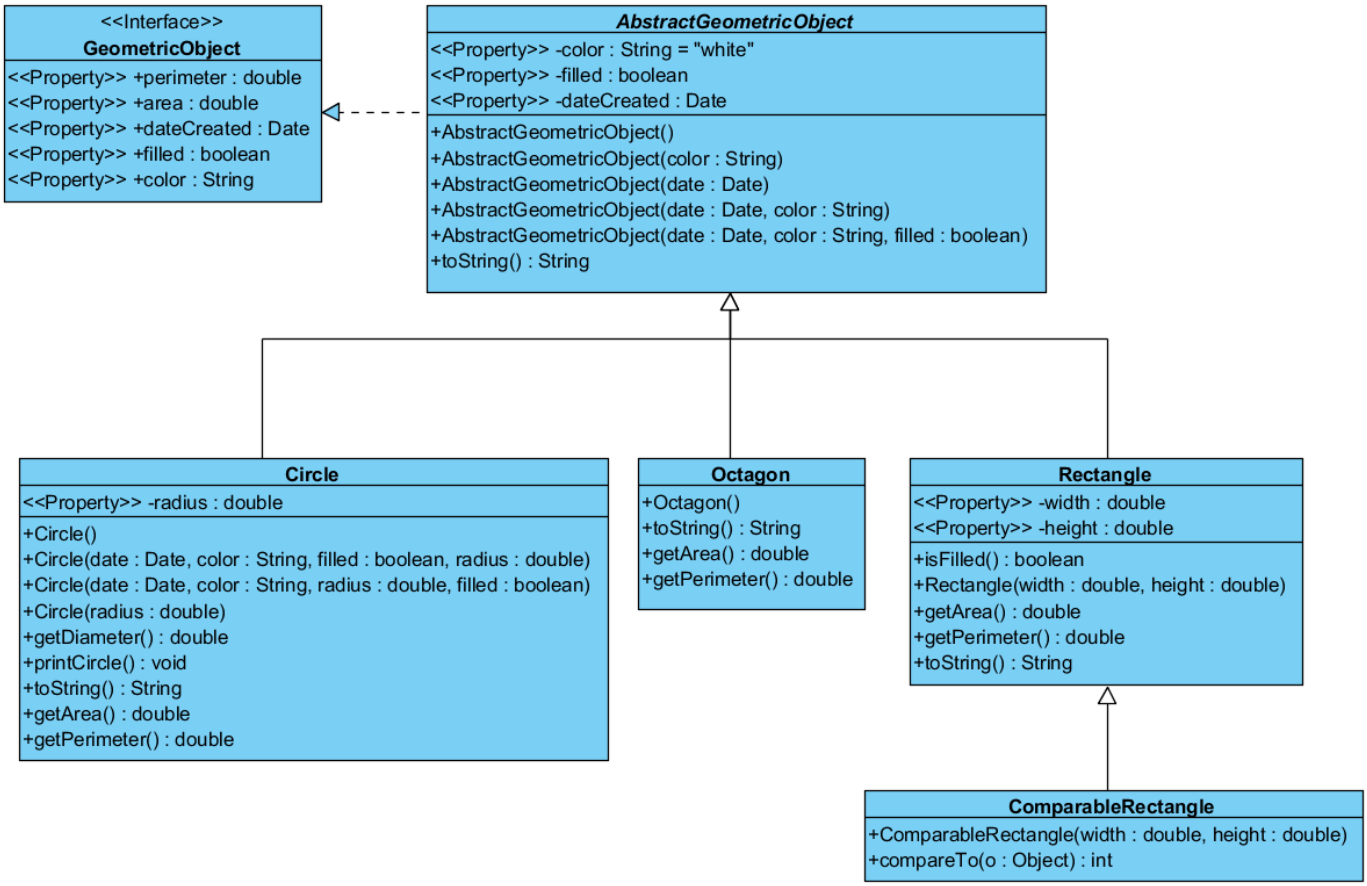
However:

- Can provide only an interface if there is some highly specialised functionality that must exist but is unlikely to have any commonality between classes (and thus must be implemented differently by all implementing classes)
- Can provide only classes where there is some very concrete functionality that is not expected to be changed or you need to ensure that a specific implementation is always used
- Can provide hierarchies of non-abstract classes where the class user can reasonably choose to instantiate classes at different levels in the hierarchy
- Can provide only classes or abstract classes if you need to add behviours later without breaking existing clients

Designing for Inheritance

- When designing for inheritance it is important to clearly **document** any *self-use* in overridable methods (i.e. non-final, public and protected)
  - This is important since otherwise a subclass may override a method and change expected behavior by not adhering to the required self-use, i.e. not calling other methods that the superclass expected to be called as part of its implementation
- *Judiciously* expose 'hooks' for extension by providing protected methods which are called at expected points of future difference
  - Too many and you unnecessarily commit to implantation decisions
  - Too few and the class lacks utility and cannot be meaningfully extended
  - In rare cases you may also provide protected attributes although this is riskier and may lead to the subclass accidentally breaking superclass functionality
- Superclass constructors should not invoke overridable methods since the method will run before the subclass constructor and may not have been properly initialized

Final GeometricObject example



Static versus dynamic types and abstraction

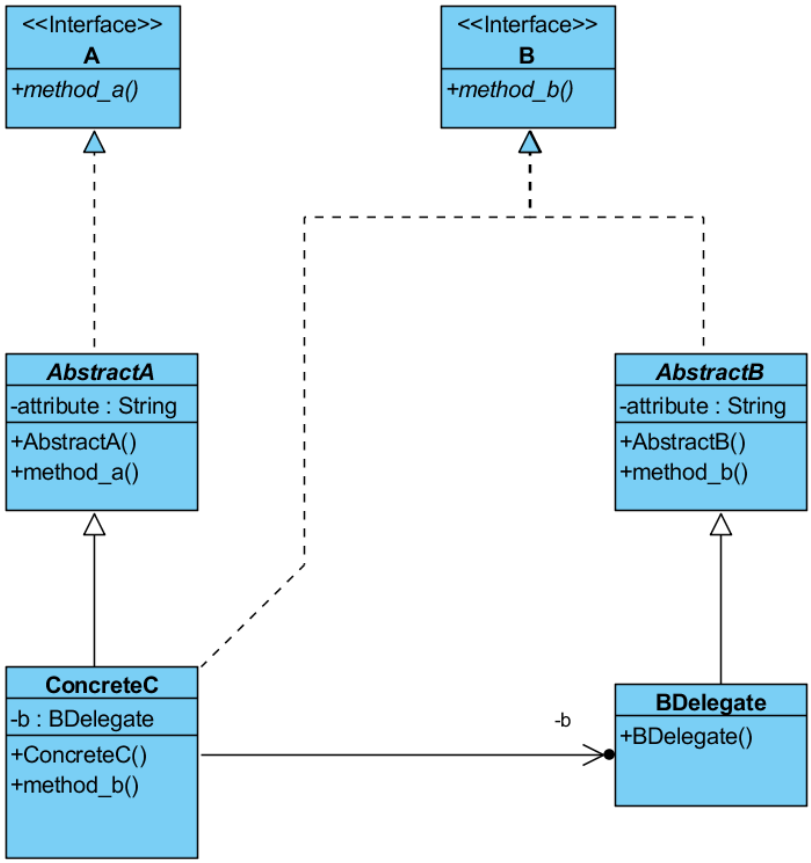
- When we instantiate an object in Java we actually specify two types:

```
List<String> myList=new ArrayList<String>();
```

- The type on the left of the '=' (instantiation operator) is called the *static (*)* or *declared* type (in the above example `List` i.e. the `java.util.List` interface)
  - This type is used by the compiler to type check any references where myList is used (e.g. you cannot assign a List to a `String`, you also cannot assign a List to an `ArrayList` since not all Lists are ArrayLists (although All `ArrayList` instances are Lists)
  - Only methods or attributes that exist in the *static/declared* type can be accessed (regardless of the type on the right of '=')
  - Try to use the most Abstract type possible (highest in class hierarchy) to favour reuse and choose interfaces over classes where possible. **This applies to parameter types as well**!
- The type on the right hand side is the *dynamic* or *runtime* type. It is the actual type of the object when instantiated during runtime and determines which actual implementation of methods are used (e.g. `ArrayList.add()` versus `LinkedList.add()`).
- Having collections of interfaces or abstract types facilitates polymorphism by placing different dynamic/runtime types in the collection

(*) Not to be confused with the static java keyword!

Delegation as an alternative to inheritance



Using delegation* to simulate multiple inheritance

- Java has a limitation that only a single class can be extended (single implementation inheritance)
- While you can implement multiple interfaces you may still have multiple default implementations (e.g. abstract classes) that you want to extend
- The previous slide shows how this can be achieved using the following steps
  - Assume two classes `AbstractA` and `AbstractB` that implement interface `A` and `B` respectively
  - Class `ConcreteC` wants to extend both of these classes i.e. `ConcreteC` is both an `A` and a `B` (this is common e.g. a tutor could be both a student and a staff member)
  - Class `ConcreteC` extends `AbstractA` (and therefore also inherits the A interface)
  - Class `ConcreteC` implements `B` since it cannot extend both `AbstractA` and `AbstractB`
  - Class `ConcreteC` instantiates `BDelegate` which extends `AbstractB` (therefore providing access to all of `AbstractB's` default behaviour via interface `A`)
  - Class `ConcreteC` implements interface `B` to pass all calls to `BDelegate`
  - i.e. `ConcreteC` is as follows (see next slide):

* NOTE: this usage could be termed *forwarding* rather than delegation since in general *delegation* implies passing a self reference which is not the case in this example. However the functionality for `B` is delegated to a separate class and so delegation is an appropriate description in this case. This is also a good example of true *Composition* since the `BDelegate` is PART OF the `ConcreteC`

```java
package sadi.topic2.delegation;

// class C is an A (inherited through AbstractA)and a B
// alternatively we could extend AbstractB, and implement/delegate A
//
// Caspar's imaginary syntax
// public class ConcreteC extends AbstractA delegates interface B to BDelegate
public class ConcreteC extends AbstractA implements B
{
   // this attribute will implement the "B" functionality via the BDelegate class
   // and avoid us cutting and pasting (since we cannot extend two abstract classes)
   private B b=new BDelegate();

   public ConcreteC()
   {
      // this constructs the "A" portion via superclass AbstractA
      super();
   }

   // method_a() is inherited from superclass AbstractA

   // method_b() from interface B is delegated to attribute "b"
   // which avoids having to duplicate method body here
   public String method_b()
   {
      return b.method_b();
   }

   // forward other methods of B interface if it had more than one method
}
```


Java 8 Interface extensions (default methods)

- Interfaces in Java 8 add more features bringing them closer to true multiple implementation inheritance
- The only limitation now compared with abstract classes is that interfaces cannot have fields/attributes
- `default` methods in an interface are instance method implementations with a body that are automatically inherited when an interface is *implemented*
- Can be called directly from a subclass to resolve conflict: `A.super.methodName();`
- Allows modifying an interface without breaking implementing classes
- Still a bit limited due to lack of instance vars but can pass params from implementing class to overcome this
- Allows pseudo multiple implementation inheritance i.e. class inherits method implementation (not just signatures) from multiple classes
- See example `sadi.topic3.multiple.java8`

Java 8 Interface extensions (static methods)

- `static` methods in an interface are similar to `default` methods however:
  - they cannot be (accidentally?) overridden which may a desirable feature
  - they can be called from static contexts
  - which may be useful in some cases and does not pose any other limitations since default methods have no field access anyway