Abstract Classes and Interfaces

Introduction to `abstract` classes and `abstract` methods

- In the inheritance hierarchy (tree), classes become more specific and concrete with each new subclass. If we move from a subclass back up to a superclass, the classes become more general and less specific - i.e., more `abstract`.
- Important class design principle:
  - the superclass contains common features of all subclasses
  - Sometimes, a common action (method) is not well-defined:
    - We know what the action is, but we cannot be specific about it.
  - Sometimes, a superclass is so `abstract` that it cannot be used to create any specific instances.

> **Abstract Class**: An `abstract` class is a class that cannot be instantiated. That is, we cannot create instances of an `abstract` class.

```
public abstract class className {
    // same as a normal class
}
```

- Any class can be defined as `abstract`.
- We can define (reference) variables of an `abstract` class type
- We can also extend (derive a subclass from) an `abstract` class
- Therefore, an `abstract` class can serve as the superclass for polymorphic methods.

> **Abstract Methods**: An `abstract` method consists of only the method declaration without the method body. The method declaration consists of only the method header (data types) information.

- Declarations are used to convey data type information to the Java compiler.

```
public abstract returnType methodname(params);
```

-
  - See `AbstractSort.java`

- Relationship between `abstract` classes and `abstract` methods:
  - `abstract` method:
    - An `abstract` method is an incomplete method. It cannot be executed (because it has no method body)
  - Rule: A class that contains an `abstract` method must be defined as an `abstract` class.
    - This is to prevent users from instantiating objects that contains incomplete methods
  - Subclasses of `abstract` classes:
    - A subclass that do not define all the `abstract` methods must be defined as an `abstract` class
    - Otherwise, the class can be defined as a normal class.
- An `abstract` class cannot be defined as a `final` class
  - An abstract class cannot be instantiated, so it needs to be extended (i.e., not `final`) into a concrete class to become instantiable.
  - Similarly, we can also not have `abstract final` methods. This is because an `abstract` method is incomplete (has no method body) and must be overridden (not `final`) so the method can become concrete.
  - A class that contains an `abstract` method must be defined as an `abstract` class
- Some interesting features:
  - A class can be `abstract` even if its superclass is non-abstract.
  - An `abstract` subclass can override a non-abstract method from its superclass with an abstract method.
    - When the implementation of the method in the superclass becomes invalid in the subclass.
- Practical use of `abstract` classes:
  - When we are unsure of how a method should be defined/implemented for that class.
  - When we do not want objects of that type being instantiated (and used) - even when the class has no abstract methods.

Introduction to Interfaces

> **Interface**: An interface can be used to define common behavior for any classes (including unrelated classes). It is a similar mechanism as inheritance for unrelated class that share some behaviors (methods)

- An interface is an abstract class-like construction that contains only
  - method declarations and
  - constants (`static` and `final`)

- We can define interface variables but cannot instantiate objects with an interface (just like an abstract class)

```
public interface myInterface {
    public abstract void myMethod(); // abstract is optional
    // other methods
}
```

- An interface cannot have a constructor method. We cannot create objects of an interface type (similar to an abstract class)
- We can define variables of an interface type (to achieve polymorphism): `myInterface a;`
- We implement an interface with an implementation class:

```
public class myClass implements myInterface {
    // must override all methods decleared in the interface
    public void myMethod(){
        System.out.println("Running myMethod() in myClass");
    }
}
```

- The implementation class must override all methods declared in an interface.
- Example to use an interface
  - we cannot instantiate an object with an interface type
  - Hence, we always upcast an object of its implementation class and assign it to an interface variable.
    - We upcast an object if the class implements the interface
    - an interface is a superclass (of unrelated objects)
-
  - See `InterfaceSort.java`
- The `Comparable` interface of the Java library
  - It is the superclass of all objects that can be compared.

```
public interface Comparable<E> {
  public int compareTo(E o);
}
```

  - The syntax `<T>` is called a generic type in Java.
  - Example to use the Comparable Interface

```
public class Circle extends GeometricObject implements Comparable<Circle> {
  private double radius;
  // other method omitted
  /**
   * Arrays.sort() will only work with Comparable objects
   * The Circle class must implement the Comparable interface
   * in order to use Array.sort()
   */
  public int compareTo(Circle other) {
      double diff = this.getArea() - other.getArea();
      return (int) Math.signum(diff);
  }
}
```

- A class can inherit from only one class, but a class can implement multiple interfaces.
  - Each interface defines a set of capabilities or "roles".
  - Implementing an interface allows a class to fulfill the role defined by an interface.
  - Therefore: a class in java can fulfill multiple roles.
- `instanceof`
  - The `instanceof` operator tells us whether a variable is a member of a class or an interface (i.e., an interface is similar to a class in Java)
- Methods with a `default` implementation in an interface

```
public interface InterfaceName {
  public default returnType methodName(params) {
      // method body
  }
}
```

  - Usage: when the implementing class does not override a method with a default implementation, the Java compiler will use the default implementation as the overriding method.
- Unlike classes that can extend only 1 class, an interface can extend one or more interfaces

```
public interface Insurable extends Sellable, Transportable {
    public int insuredValue();
}
```

- The interface `Insurable` will combine:
  - All methods in `Sellable`
  - All methods in `Transportable`

- The `insuredValue()` method

Abstract Class	Interface
Can have constructors, instance variables, constants, abstract methods, and non-abstract methods	Can only have abstract methods and constants
Is extended by a subclass that may implement the abstract methods, but does not have to. (If the subclass does not implement all abstract methods, it must be defined as an abstract class too)	Is implemented by a subclass that must implement all the abstract methods

Review of OOP Concepts

- A class, is like a definition of a data type in Java.
- An instance of an object of a given class is created (instantiated) using a constructor and the `new` operator.
- Object instances in Java are accessed through reference variables.
- A subclass can extend a superclass and use its methods and instance variables through inheritance and polymorphism.
- Abstract classes can contain 0 or more abstract methods. Classes containing at least one abstract method must be abstract. Abstract classes can not be used to instantiate objects. Abstract classes are extended by a subclass that defines the abstract methods.
- Interfaces contain only abstract methods and constants and provides a template for another class to implement all of the methods declared in the interface.