CSE 1325 - Object-Oriented Programming

Interfaces

Alex Dillhoff

University of Texas at Arlington

Java Interfaces

Abstract classes and methods allow the developer to define a particular behavior for classes that inherit it.

This is useful in describing what subclasses should do.

This is limited by the fact that Java only supports single inheritance.

Java Interfaces

Luckily, Java provides **interfaces**, entities that define *what* a class should do.

Any class that implements an interface must provide definitions for those behaviors.

Java Interfaces

Consider the method `Arrays.sort(Object[] a)`.

The documentation states that all elements in the array a must implement the `Comparable` interface.

Java Interfaces

The `Comparable` interface declares a method `int compareTo(T other)`.

Any class that `implements Comparable` **must** provide a `compareTo` method.

Java Interfaces

Additionally, `compareTo` definitions should follow the following rule:

”Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.”

This rule ensures two objects can be ordered.

Java Interfaces

Interface methods are `public` by default, so there is no need to specify the keyword.

However, more recent versions of Java have expanded on what interfaces can do.

Implementing Interfaces

How can we implement interfaces in our own classes?

With the `implements` keyword.

```
public class Player implements Comparable
```

Implementing Interfaces

Then, the required methods must also be defined.

```
public int compareTo(Player other) {
    return name.compareTo(other.name);
}
```

Now, Player objects can be sorted.

Implementing Interfaces

Example: `SortPlayerExample`

Implementing Interfaces

By implementing the interface, we have full control over how our custom objects are sorted.

Since our class implements the interface, any method requiring it can safely call the `compareTo` method on our objects.

If our class did not implement it, an exception would be thrown.

Implementing Interfaces

One potential issue to look out for is when working with subclasses.

If a subclass does not implement an interface and an interface method is called on it, it will fall back to the definition provided by the superclass (if applicable).

Implementing Interfaces

If a superclass and subclass implement `Comparable` in their own way, a collection of mixed objects could result in unexpected output when sorted.

Interface Properties

Interface variables can be created, but you cannot instantiate a new object of an interface.

If we assume `Player` implements `Comparable`, then the following is valid.

```
Player p = new Player();
if (p instanceof Comparable) {
    // Do something
}
```

Creating Interfaces

We can create our own interfaces using the `interface` keyword.

Let's look at a basic example: `InterfaceExample`

Creating Interfaces

Starting with Java 8, interfaces can define a `default` implementation.

With Java 9, interfaces may include `private` methods.

It is recommended to only use these in special circumstances.

Creating Interfaces

Interfaces should be used with the original intent in mind...

only to define *what* a class should be doing.

Extending Interfaces

Just like classes, interfaces can be extended to define more specific behaviors.

This is done by simply using the `extends` keyword.

```
public interface Report extends Callback
```

Extending Interfaces

When implementing a sub-interface, the class which implements it must implement all parent methods as well.

Example: `InterfaceExtendExample`

Comparison with `abstract`

This pattern of defining behaviors is similar to that of `abstract` methods.

The main difference is that Java only allows single-inheritance, so only one source of behaviors can be included via inheritance.

Comparison with `abstract`

In any case, inheritance isn't a good match for what interfaces provide.

It is often the case that you want to ensure classes adhere to specific methods without adding the extra baggage of subclassing.

Default Methods

As of Java 8, an interface can define a `default` behavior.

This is implemented in practice by using the `default` keyword.

Default Methods

The benefit to defining a `default` behavior is that it allows interfaces to be updated without affecting previous code.

Default Methods

Consider the following scenario:
An application developer uses a custom interface that you developed as part of an API. Later, you wish to update the interface to add new methods.

If you declare the new method as usual, the application developer's program will break unless they implement the new method in all classes that implement your interface.

Default Methods

Instead, you should set a `default` behavior for your new method.

In this way, existing applications do not need to explicitly define this new method.

If they omit it, the default method will be called.

Default Methods

Example: `InterfaceDefaultExample`

Multiple Inheritance

Java does not support multiple class inheritance, but the addition of default methods allows some of that functionality.

A key difference still remains: a class has state information. An interface does not.

Multiple Inheritance

Having default methods means that classes can implement multiple interfaces with some defined behavior.

This is about as close to multiple inheritance as we can get in Java.

Multiple Inheritance

If you implement multiple interfaces, it is important to understand naming conflicts.

Consider a class implementing two interfaces named `Interface1` and `Interface2`.

Multiple Inheritance

Example: `MultipleInheritance.java`

Custom Comparators

The `Comparator` interface is another interface that allows for comparisons when sorting.

It is used by methods such as `Arrays.sort(T[] a, Comparator<? super T> c)` to define custom sort behaviors.

Custom Comparators

What if we want to override the default behavior when sorting custom objects?

We can define our own custom comparisons by subclassing `Comparator`.

Custom Comparators

By creating such classes, we can define custom sorting logic without interfering with the original interface implementation.

Example: `RollInitiative`

Cloning Objects

An important concept when dealing with objects is that of **shallow** copying versus **deep** copying.

By default, objects in Java are not copied in memory. Instead, their reference or address is copied.

This is called a **shallow** copy.

Cloning Objects

Example: `ShallowCopyExample`

Cloning Objects

The `Object` superclass has a `clone()` method for **deep** copies, but this will not work for any subclass.

A subclass must implement the `Cloneable` interface as well as override the `clone()` method.

Cloning Objects

Example: `DeepCopyExample`

Cloning Objects

There are some subtleties to consider when customizing `clone()` for your custom classes.

If your class contains subobjects that are *mutable*, these subobjects may need to be deep copied as well.

That is, they must also implement `Cloneable` and override `clone()`.

Cloning Objects

If your class contains no subobjects that are *mutable*, you can simply call `super.clone()` in your override of `clone()`.

Cloning Objects

Consider a parent class that overrides `clone()`.

Any subclasses that do not customize `clone()` will fall back to the parent implementation.

This may be problematic depending on your usage.

Cloning Objects

Deep copying is not something you may need very often.

It is important to ask yourself if your application even needs to provide deep copies.

If you can avoid it, stick to shallow copies.

Interfaces

Bonus Example: `Clock`