Programming against an interface

You've got interfaces - now use them!

Why using interfaces

Using interfaces lets you

- Hide implementations from client code
- Protect client code from changes in concrete classes
- Swap implementations for certain parts of your system without breaking client code
- Hide external systems or low-level design decisions from client code

In short: Interfaces help you focus on *what should be done* rather than *how should it be done*!

What we mean

When we say “program to an interface” we mean

- let your references be of interface type (if there is one)
- or at least be of abstract class type
- or as high up the class hierarchy as possible.

Since this lecture is from a chapter about Java interfaces, we’ll stick to the top bullet here.

How the benefits are realized - hide impl.

Let’s say that we have an application which can create output from an object formatted as different formats (XML, HTML, JSON, CSV). The client code which requests such a formatting action doesn’t need to know too much about what the format is and what the class name of the formatter is:

```
Formatter formatter = FormatterFactory.getFormatter();
String formattedDocument = formatter.format(orderList);
saveDocument(formattedDocument, formatter.fileSuffix());
```

How the benefits are realized - hide impl.

- We can’t tell what the actual class of the formatter is
- We don’t even know what the *format* is (is it XML? Something else?)
- We don’t know what the file suffix of the document is either

Do we really need to know all of the above? What happens if we know?

```
Formatter formatter = FormatterFactory.getFormatter();
String formattedDocument = formatter.format(orderList);
saveDocument("orders",
             formattedDocument,
             formatter.fileSuffix());
```

How the benefits are realized - hide impl.

- We can’t tell what the actual class of the formatter is
- We don’t even know what the *format* is (is it XML? Something else?)
- We don’t know what the file suffix of the document is either

If the client code knows the above (or worse: needs to know), then we risk ending up with code like this:

```
XmlFormatter formatter = new XmlFormatter();
String formattedDocument = formatter.format(orderList);
saveDocument("orders", formattedDocument, "xml");
// hope we don’t change format soon, so we’d have to recode
```

How the benefits... - protect from changes

If the implementation of e.g. XmlFormatter changes, then our code also needs to change:

```
XmlFormatter formatter = new XmlFormatter(2); //indents
String formattedDocument = formatter.format(orderList);
saveDocument("orders", formattedDocument, "xml");
// hope we don’t change format soon, so we’d have to recode
```

Above, the author of the XmlFormatter class decided to change the constructor, so we had to change also our client code.

How the benefits... - Swap implementations

Our team has written a better implementation of XmlFormatter, XmlSuperFormatter. If we want to use that instead, we have to change our code:

```
XmlSuperFormatter formatter = new XmlSuperFormatter();
String formattedDocument = formatter.format(orderList);
saveDocument("orders", formattedDocument, "xml");
// hope we don't change format soon, so we'd have to recode
```

If we had programmed to the interface Formatter instead, we wouldn't have to chage our code.

How the benefits... - Hide low level stuff

The order list comes from some external source. Right now it comes from a database, but later it might come from some storage system on the network.

```
Database.logIn();
List<Order> orderList = Database.getOrderList();
Database.close();
```

If we had programmed to an interface, Storage, instead, we would have hidden the fact that we deal with a database, so that we could switch to some other datasource later on.

How the benefits... - Hide low level stuff

Using an interface instead of the Database class:

```
Storage storage = StorageManager.getStorage();
List<Order> orderList = storage.getOrderList();
// format the order list...
```

If we change datasource to some network system later on, the code above will stay the same.

Of course, StorageManager must be able to notice the change somehow.

Some collections examples

We have a method totalPrice which takes the orderList and calculate the total price of all orders. This is not the best implementation:

```
Storage storage = StorageManager.getStorage();
ArrayList<Order> orderList = storage.getOrderList();
BigDecimal totalPrice = totalPrice(orderList);

// some other place:
public BigDecimal totalPrice(ArrayList<Order> orders) {
  // calc. and return the total price
}
```

If we change to LinkedList<Order>, how many places do we have to change?

Some collections examples

If we change to LinkedList<Order>, how many places do we have to change?

```
Storage storage = StorageManager.getStorage();
ArrayList<Order> orderList = storage.getOrderList();
BigDecimal totalPrice = totalPrice(orderList);

// some other place:
public BigDecimal totalPrice(ArrayList<Order> orders) {
  // calc. and return the total price
}
```

At least three places (client class, Storage interface (and implementations!), totalPrice()).

Some collections examples

Preferred way - use an interface type!

```
Storage storage = StorageManager.getStorage();
List<Order> orderList = storage.getOrderList();
BigDecimal totalPrice = totalPrice(orderList);

// some other place:
public BigDecimal totalPrice(List<Order> orders) {
  // calc. and return the total price
}
```

Works whether getOrderList really returns an ArrayList or a LinkedList or any other kind of List!

Concluding remarks

Remember: focus on what should be done, and not how it should be done.

A list should keep a sequence of orders. That's the "what". If it stores the sequence in an array or using links is not interesting at all. That's the "how".

Same with the formatter example earlier. The "what" is to format a list of orders. The "how" is the class doing the actual formatting.

Same with the storage example. The "what" is to store orders. The "how" is whether to store them using a database or some other source.

Further reading

- https://www.artima.com/lejava/articles/designprinciples.html
- https://dzone.com/articles/programming-to-an-interface