COMP6700/2140 Interfaces. Types in Java

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

March 2017

Topics

1. Multiple Inheritance by Contract
2. Interface Declaration
3. Most Important OOP Principle: Program to Interface
4. Designing Classes for Inheritance
5. Default Methods: now almost Traits

Interfaces: Only declared behaviour

Multiple *inheritance by implementation* (where it is allowed) is not too much of a complexity feature, rather a "joy killer". And it is rarely path to a good design solution. Often, that path is *inheritance by contract*.

In Java, inheritance by contract comes via types defined by contract (promise): `interface` keyword is used in declaration instead of (`abstract`) `class`. An interface is an abstract class stripped of any method implementations and state fields. **No implementation, no state! Pure contract.** The only members allowed are non-private method declarations (*abstract* modifier is redundant), and *constants* — *static* (no instantiation!) and non blank *final*s (again, `public static final` modifiers are redundant in declarations):

```
interface Verbose {
   int SILENT  = 0; // interface constants have fully CAPITALISED identifiers
   int TERSE   = 1;
   int NORMAL  = 2;
   int VERBOSE = 3;

   void setVerbosity(int level);
   int getVerbosity();
}
```

In the manner of *extending* abstract class to get complete implementation and object creation, an interface must be *implemented* into the concrete class via implementation of every method declared in the interface.

Interfaces: Implementation

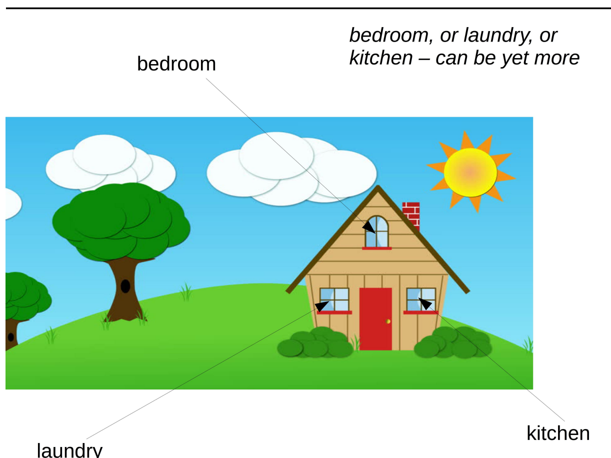
The *diamond of death* phenomenon cannot occur if the implementing class inherits *only one* (abstract or concrete, does not matter) class, *and* implements a number (32767 to be exact; imaging a class which implements that many!) of interfaces, because there is no competing implementations which vie to become the one in the derived class (there is at most one).

If a class implements multiple interfaces, they are declared in a comma separated list following the `implements` keyword:

```
class ClassD extends ClassA implements InterfaceB, interfaceC {
   ... adding own fields and methods ...
   ... possibly overriding methods form ClassA ...

   void methodOfB(){
   ... ...
   }

   int methodOfC(){
   ... ...
   }

}
```

Object with Multiple Types

- An object may have multiple types — nothing wrong with this
- The same object means different things to different ~~people~~ clients
- More types can be added later without affecting other clients (would need a new class which implements a new interface); sometimes the original object is recreated using the *Decorator Pattern* which creates an illusion that the old object assumes new features.



Interfaces as types, polymorphism again

Interfaces allow breaking of a program into parts: classes which *implement* a particular interface are *decoupled* from classes which *use* the behaviour declared in that interface.

- Use *interface* type as a *formal parameter* for a method — any method promised by the interface parameter can be invoked:

  ```
  doSmth(type param) { // at compile time type is an interface;
      param.foo();     // at run time param is an object
      param.bar();     // of implementing class
  }
  ```

  When the method is called, an *object of a real class*, which implemented the *type* interface, is passed as the *actual parameter*.
- Same principle applies to method return types — they are declared as an interface type, but when executed, an object of a concrete class which implements that interface is returned.
- Interface alone suffices to compile the code which uses it. At run-time, objects of a real class are required.
- Interfaces define the method *signatures*, and implementing classes must preserve them.
- *An interface name can be used anywhere a type can be used*; `interface` provides the most flexible form of type definition.

Interfaces are extendable (`NewInterface extends OldInterface`) for adding new method declarations without breaking existing classes which implement the *OldInterface* type. Unlike class extension, a multiple inheritance of interfaces is allowed (no case for competing implementations, and parent interface's constants get hidden).

Examples of Interfaces

- Objects often needs to be compared (smaller, equals, greater?) — always possible if their class has implemented the `java.lang.Comparable` interface:

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

  Here `T` is the type parameter which represents the objects' class: `String implements Comparable<String>`. Sounds weird? Not really.
- `java.util.Comparator` interface is similar to `Comparable` but its method takes two parameter (objects to be compared):

  ```
  public interface Comparator<T> {
     public int compare(T t1, T t2);
  }
  ```

  Will be used when we need to implement comparison of objects with a new aspect, but the objects' class is already *Comparable* (and in a wrong way).
- `java.lang.Runnable` for defining a task which will be executed in a new thread:

  ```
  public interface Runnable {
    public void run();
  }
  ```

  A *Runnable* object is given to a new thread as a parameter; the program will decide itself when to start the thread (only then, the defined inside `run()` process will be started).

Objects of these interfaces are now often replaced by λ-expressions.

Default Methods

Java 8 has extended existing interfaces (like `java.util.Collection`) to provide new methods needed in programming with streams (`java.util.Collection.stream()` method). By the old rule, this should have broken the backward compatibility, because every class which implemented such extended interface now had to provide an implementation for the new methods (even if the old code did not use it). Is the "sacred cow" of Java is slain? NO!

The defender methods (those newly added) are now **implemented in interfaces!** They are **concrete methods** marked by the qualifier `default` (it's not seen in the class interface, **demo**).

```
interface Person {
    long getId(); // usual abstract method
    default String getName() { // public defender (aka virtual extension) method
        return "John Doe, NSA analyst";
    }
}
class USAPatriot implements Person {
    public long getId() { return 42; }
}
```

This compiles OK, and the call `new USAPatrion().getName()` returns … what you might have guessed. Classes still can implement multiple interfaces. If two (or more) have default methods with the same name, the name resolution rules apply (not so simple Java now, eh?). More on this later, in **F2**.

Abstract Classes vs Interfaces

Which one to choose

Repeat the difference:

- Interfaces provide a form of multiple inheritance without complex semantics. For abstract classes, the multiple inheritance is forbidden, even if all methods are abstract.
- An abstract class can contain partial implementation, protected fields and methods, static methods and other standard elements, while interfaces can only have public static constants and public methods without implementation.

Therefore:

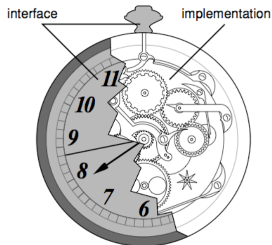
- If multiple inheritance is envisaged, interfaces are used.
- But abstract classes give easier re-use through available (partial) implementation.
- Abstract classes allow to constrain modification (via use of `final` modifier) of some aspects of behaviour.
- If a major class is meant to be extended, make it to implement an interface, and make clients of this class to refer to the interface instead.

Deeper view of Object

The idea that *object encapsulates its implementation and reveals its interface* represents only the technical aspect. A deeper and more powerful idea is that **Object *is* its Interface**.

If an object is an idea of computation, then it is more stable and has more longevity if it has less accidental (implementation specific) characteristics. In its ideal, purified form an object is defined by operations which one can perform with it, and by nothing else.

The watch measures time, it does not count it!



(Courtesy of Apple Inc.)

Three kinds of type in Java

Three reference types — classes, abstract classes, and interfaces — seems excessive, isn't?

1. Concrete classes — No questions! Templates for real objects
2. Abstract classes — Understandable! Help to reuse code
3. Interfaces — ???

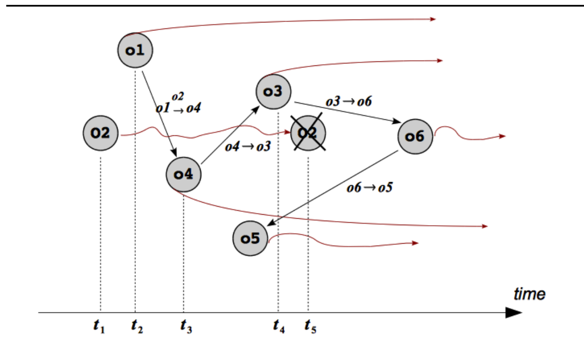
The most important OO idea: **"Program to interface!"**

When writing a program, try — as much as possible — to program to interface. The client code of an object (a class which has it as a field, or a method which uses it as an actual parameter, or a code block which is given access to an object) should be only revealed that part of its interface which they actually use, not more! If the object reference has many types (*ie*, it is instantiated to a class which implements multiple interfaces), and the client needs only one of them, it is only with that type the object should be declared to its clients.

Ideas, concepts and knowledge in general (including software) — they are most useful, stable and valuable if they are expressed in abstract terms.(How many ideas and how many *working* artefacts from Antiquity are around today?) In software, this abstraction of object properties is its interface.

Interface View of the World

Physics analogy: motion of a bunch of electrically charged particles (electrons) in an external electro-magnetic field. What do we need to know to *fully* describe the motion? Electron's internal structure needs **not** to be known to calculate the trajectories; only their "interface": charge, mass, position and velocity. Future advances in Physics may reveal electron internal structure, but the laws of motion will remain intact.



Dynamics in terms of “Interfaces”

Signals and methods in OO programs

1. (time `t1`, `t2`...): Objects `o1`... `o6` are created at various stages of execution
2. (`t2`): An object `o1` sends a signal to `o4` with data containing a reference to an object `o2`
3. (`t3 >= t2`): The object `o4` receives this signal and executes a method (or, several methods); the received signal contains a reference to `o2` (a parameter to one of `o4` methods), and `o4` may invoke methods of `o2` as a part of its response; the state of `o4` *may* change, and it *may* emit a signal to another object `o3` passing some data; the data which are passed are obtained as *return values* of methods which the object `o4` executed between receiving the signal from `o1` and sending the signal to `o3`
4. (`t5`): All references to the object `o2` are lost, this object “dies” (is garbage-collected)
5. States of all objects undergo evolution during the program execution (red lines), which depends both on the global program logic (“external force”), and on signals they receive from other objects (including their own).

Design for inheritance

When a class is extended and an implementation of its method is changed, it's said then the method is *overridden* (not confuse with method *overloading*). When we are dealing with instances of a class, it is the *actual class of the object* which determines which implementation is used. Apart from changing implementation, the overriding can *widen* the access modifier, *ie*, make protected method `public`, but it cannot make it `private`. If the parental implementation needs to be used, it can be accessed via the reference to the superclass: `super.do()`.

When designing a class for extension, one must consider which access modifier to give to fields (`private` or `protected`, the last choice can be good for performance but must be exercises carefully) and to methods (`protected` or `public`). Like many design issues, this is a complex one and requires experience and care. Pragmatic advice: if you do not trust the class's possible children to preserve the integrity of the class contract, then limit the access of its members. If the class method should not be overridden no matter what, it must be declared `final` (this will disallow any possible future overriding of the method by its descendants). If the whole class is not meant to be extended, it itself must be declared `final` (all methods in such a class are implicitly `final`).

```
final class NonExtendableClass { .... }
```

Attempts to declare a class which extends *NonExtendableClass* will be met with obstinate compiler error.

Where to look for this topic in the textbook?

- Hortsmann's *Core Java for the Impatient*, Ch. 3.1, 3.2, 3.3
- Oracle's Tutorial Interfaces
- For a critical view about the design decision to have both — `abstract class` and `interface` — type declarations, as well as two *different* statements to define a type — with `extends` and `implements` keywords, read a short blog by Robert C. Martin (aka Uncle Bob) "Interface Considered Harmful" (I recommend reading this blog regularly)