OOP F2005
Lecture 7
Interfaces & Design by Contract

Kasper Østerbye
IT University Copenhagen

Today's schedule

- Design by Contract
  - why the term contract
  - what design issue is captured, and why bother
  - what is a pre-condition
  - what is a post-condition
  - what is a class invariant
  - How to do this in Java.
    - you cannot, only in documentation
    - experimental java's exist
- Assert in Java
  - loop invariants
  - design by contract

Design by contract

Consider the service offered by the Danish postal service:

For 4.50 dkr, they will deliver a letter to the address printed on the front of the envelope, if the letter is posted in Denmark, if it is less than 23x17x0.5 cm. If it is handed in before a given time of the day, there are a next day delivery guarantee.

The advantage for me is that I know how much postage to put on a standard letter, I know when to post it, and when I should expect it to arrive.

The advantage for the postal service is to receive 4.50, not to have obligations for late delivery, and not to have obligations for odd size letters.

```
class DanishMailBox {
    …
    /*pre:
        l is within size restrictions & current
        time is before postingDeadline
      post:
        letter l is delivered at address next day
    */
    public void sendStandardLetter(Letter l){…}
    public Time postingDeadline(){…}
    …
}
```

3

The pre condition is what must be true before we start the method, the post condition is what must be true after the method has been executed.

It is the responsibility of the caller of the method to make sure the pre-condition is met, it is the responsibility of the called object to make sure that the post condition is met.

Iterator contract I

An iterator is an object which will provide sequential access to a collection.

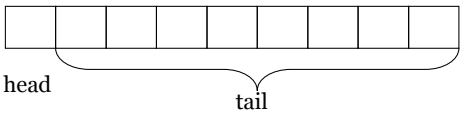
The java.util.Iterator<E> interface has the following methods :

- boolean hasNext()
- E next()
- void remove()

hasNext returns true if the iterator is not empty.

next returns the head, and moves to tail.

remove removes the head from the underlying collection – or throws an UnsupportedOperationException



An iterator is *empty* if it will not be able to produce any more elements. The next element which it will produce is called its *head*, and all the remaining is its *tail*.

Note, given an iterator itr, itr.tail is an iterator which will return the same elements as itr, except for itr.head.

4

The Iterator in java.util is one of the many examples of bad design in Java (there are more examples of good design though). It is not considered good design to throw the UnsupportedOperationException. Instead, the designers should have defined two interfaces. One named SimpleIterator, which has only the methods hasNext and next. And an other RemovableIterator, which extends SimpleIterator with the remove method:

```
interface SimpleIterator{
        boolean hasNext()
        E next()
}
interface RemoveableIterator extends SimpleIterator{
        void remove()
}
```

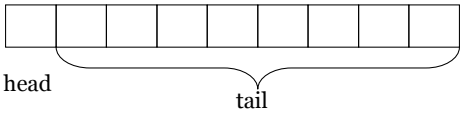
This way a (collection) class can decide which kind of Iterator it wants to implement.

Iterator contract II

An empty pre-condition means that the method can always be called.

Notice, it is the responsibility of the caller to make sure that next() and peek() is not called when the list is empty.

Notice, we will assume that the iterator does not change as a result of a call to cloneMe(), since nothing is stated to indicate such behaviour




```
public interface Iterator {
    /* pre: none
     * post: return true if the iterator is not empty. */
    boolean hasNext();

    /* pre: hasNext()
     * post: return head of old.iterator
            & this is old.tail. */
    Object next();

    /* pre: next has been called
            & remove has not already been called
            & remove is supported by this iterator
     * post: removes the last element  returned
            by next*/
    void remove();
}
```

5

In the paper by Bertrand Meyer, Meyer uses the programming language Eiffel, which he has designed himself.

The Eiffel language has constructs in the language itself to deal with pre and post conditions. In Java there is no such constructs and we can only write them in the comment of the methods.

There is often a need to refer in the post condition to the state of the class as it was before the operation. In the next() method, we say that this is the old.tail, which means that the iterator now is the tail of what it was before the next() call.

There exists several experimental tools for Java which extends the Java language with support for design by contract. Try ”design by contract java” on Google if you want to try the real thing.

An array iterator

This iterator will iterate over the elements in an array which is given as argument to its constructor.

The iterator can be used as:

```
String[] names ={"Bill", "Ben","Jack"};
ArrayIterator itr = new ArrayIterator(names);
while ( itr.hasNext() )
    System.out.println( itr.next() );
```

But, will this actually print out the right elements?

To examine this, we need to compare the implementation of each method to the contract to see if the ArrayIterator satisfies the Iterator contract, and not just defines methods with the right signature

```
public class ArrayIterator implements Iterator{
    private final Object[] theArray;
    private int index; // index of next element
    public ArrayIterator(Object[] objects){
        theArray = objects;
        index = 0;
    }
    public boolean hasNext(){
        return index < theArray.length;
    }
    public Object next(){
        return theArray[index++];
    }

    public void remove(){
        throw new UnsupportedOperationException();
    }
}
```

6

The point of this, and the next slide is to introduce the notion of a class invariant.

The reason we need this concept is that we cannot reason about the individual methods in isolation from the objects the methods works on.

The invariant helps to state something about the relationship between the fields of the object, and the purpose of the object, so we can argue in that each method does indeed do the right thing.

Class invariants

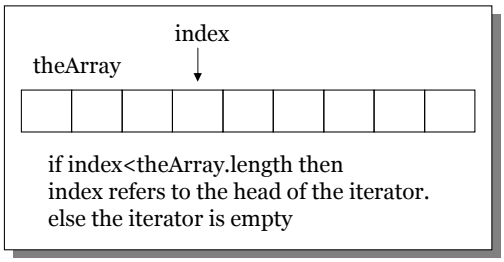
But, we cannot see if the body does the right thing in isolation from the class itself. What is the the value of index, and what is theArray?

A class invariant is a *postulate* on the fields of a class. The postulate is supposed to be true after each method has finished, and after the constructor has finished.

It is the programmers responsibility to make certain that the invariant is maintained.

If we assume that the invariant is as in the drawing, then hasNext() is correct.

```
/* pre: none
 * post: return true if the iterator is not empty. */
public boolean hasNext(){
    return index < theArray.length;
}
```



7

The rule of class invariants is that it is a postulate must be true after each method call and after the constructor has finished. Therefore it must also be true at the beginning of each method call. The invariant cannot be assumed to be true before the constructor has finished.

If we assume the invariant to be true before we call the hasNext() method, we know that either index<theArray.length, in which case index points to the head of the iterator or index >=theArray.length, in which case the iterator is empty.

Thus, we test to see which of the two cases we are in, and if it is not empty, we return true.

Array Iterator, next and constructor

Reg. next.

We assume the invariant is true, and hasNext() is true, that is the iterator is not empty. Thus, we know that index refers to head, which is what we return.

But we also increments index by one, which is what is needed to make sure that the iterator now is its tail.

Reg. the constructor

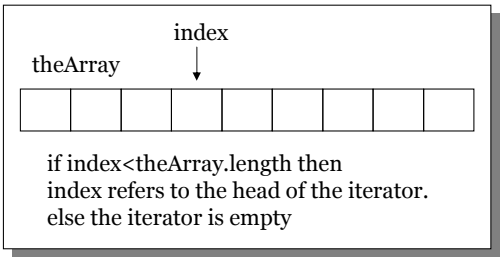
The precondition ensures that we will not attempt to iterate null.

if objects has elements in it, then index=0 is the head of the iterator

if objects it empty, then length is 0, and index is not less than 0, hence the iterator is empty.

In both situations the invariant is true.

```
/* pre: hasNext()
 * post: return head of iterator; this is old.tail.
 */
public Object next(){
    return theArray[index++];
}
/* pre: objects != null
 * post: invariant
 */
public ArrayIterator(Object[] objects){
    theArray = objects;
    index = 0;
}
```



8

I use a dirty trick in the next method. Normally we use the increment operator ”x++” as a statement to add one to a variable x. But really, x++ is an expression, which returns the value of x, and then increments x.

If you look at this code fragment:

int x=8;

System.out.println(x++);

System.out.println(++x);

It will print 8, 10. After the first print, x will have the value 9, but will print the value it had before it was incremented. In the second print, we first increment x, then print its new value.

x++ is called post-increment, and ++x is called pre-increment, see Java Precisely at page 30, section 11.2.

So, return theArray[index++] corresponds to the longer

```
Object o = theArray[index];
index++;
return o;
```

It is often much easier to reason about the invariant being true after the constructor is executed *if one does not use initializers*.

Invariants and encapsulation

Please observe the wording used when arguing for the pre/post conditions.

They all say, ”If the invariant is true, then so and so and so and so, therefore the invariant is still true”.

But what if the invariant is not true?

This is why it in general is a good idea to make fields private.

If the index field was made public, then we could not be sure the invariant was valid, as someone might had changed it from the outside.

Also notice that one should be very careful with ”setter” methods for variables mentioned in the invariant. A setter for the index would be as bad as making it public

```
public class ArrayIterator implements Iterator{
   private final Object[] theArray;
   private int index; // index of next element

   public ArrayIterator(Object[] objects){
        theArray = objects;
        index = 0;
   }
   public boolean hasNext(){
        return index < theArray.length;
   }
   public Object next(){
        return theArray[index++];
   }

   public void remove(){
        throw new UnsupportedOperationException();
   }
}
```

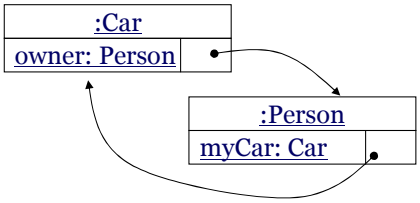
9

At a very high level of abstraction, one can say that one purpose of the encapsulation is to ensure that the invariant of the class can be ensured. There MUST NOT be public methods which break the invariant, as it is then not possible to know what the program does.

An invariant on Persons and Cars

Consider the exercise on Persons and Car from lecture 2. There was three rules:

1. Each car is owned by exactly one person, that is, no car is without an owner, and no car is owned by more than one person.
2. If a person owns a car, that car is owned by that person, and vice versa.
3. A Person can own at most one car




```
public class Car {
    /* inv: owner != null, & owner.myCar == this
    private Person owner;


    /* pre: p.myCar == null
      post: p.myCar = this. */
    public Car(Person p){
        owner = p;
        owner.setCar(this);
    }

    /* pre: p.myCar == null
      post: p.myCar == this  */
    public void setOwner(Person p){
        owner.setCar(null);
        owner = p
        owner.setCar(this);.
    }
}
```

10

The invariant states that the owner is not null, and that the myCar field of the owner refers back to this. That reflects the picture.

Constructor:

A Person cannot own more than one car. We must assume (precondition) that the person given as parameter to the constructor is one who does not own a car.

post condition states that the person p owns the car afterwards.

owner = p, makes the reference from the car to the person p.

owner.setCar(p) makes the reference back again (See next slide).

setOwner:

We must again assume that the new owner p does not already own a car (precondition).

We start by detaching the old owner of the car from this car, so the old owner does not own the car nay more.

We then set the reference from car to person, and

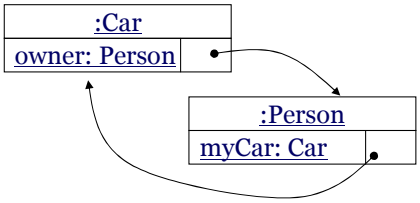
the reference from person to car.

The class Person is simpler, it has the following form

```
class Person{
                /* inv: if myCar != null, myCar.owner = this.
                private Car myCar;
                Person(){};
                /* pre: myCar == null, c.owner = this
                   post: inv
                */
                public void setCar(Car c){
```

An invariant on Persons and Cars II

The class Person is simpler.




```
class Person{
    /* inv: if myCar != null then myCar.owner = this.
    private Car myCar;

    Person(){};

    /* pre: myCar == null, c.owner = this
       post: inv
    */
    public void setCar(Car c){
        myCar = c;
    }
}
```

11

The invariant of Person has to take into account two situations, one where the myCar reference is null, and an other where it is not.

Notice the precondition of setCar.

It assumes that this person does not already own a car, and it assumes that the car c already refers to this person. These two conditions are exactly matched in the call from the method setOwner in class Car.

In the C++ programming language there is a special encapsulation mechanism called “friends”. Rather than letting setCar be a public method, we could have declared it to be a friend of Car. This way the method could only be called from Car objects.

The friend mechanism is very useful for invariant encapsulation which go across several objects as is the case here.

Pre conditions and exception handling

An important property of pre-conditions is to establish under what conditions a method will work.

void sort(int[] a)

Is it acceptable to all with a null reference? An empty array? An array of length larger than 5.000.000 elements?

It is the responsibility of the client to ensure that the method's pre-condition is true.

The contract avoids that both the client and the object performs a test if the argument is null.

```
/* pre: a not null, a not empty
    post: a sorted
*/
void sort(int[] a){
    try{
        … a extraordinary good
        … sorting algorith goes here
    }catch(Exception ex){
        if (a == null || a.length = 0)
            throw new PreConditionVoilation();
    }
}
```

12

On this topic a resign for a better explanation. Please read from page 49 and the rest of Bertrand Meyer's paper, the use of exceptions is very nicely described.

Java assertions

Java has a mechanism called assertions.

```
assert boolean-condition;
```

This statement will throw an exception if the boolean condition is not true.

However, you must tickle java a bit to do so.

```
javac -source 1.4 myfile.java
```

to compile it

```
java -enableassertions myfile
```

to execute and actually test the assertions

Assert statements can be inserted in the beginning of a method to check pre-conditions

Assert statements can be inserted just before return, to check post conditions

Assert statements can be inserted at the end (just before each return) to check that the class invariant is true.

Design by contract is supported in Eiffel.

Assertions lack:

- Systematic support for class invariants
- Support for ”old” variables
- Support for combination of pre and post conditions when using inheritance

13

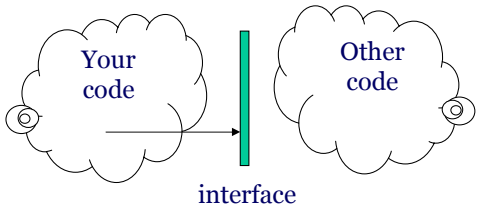
Assertions are described in Java precisely.

Assertions are much better than nothing! So while I am not impressed with the assert mechanism, it is clearly better than not using it.

The three points of critique remains however.

Interfaces

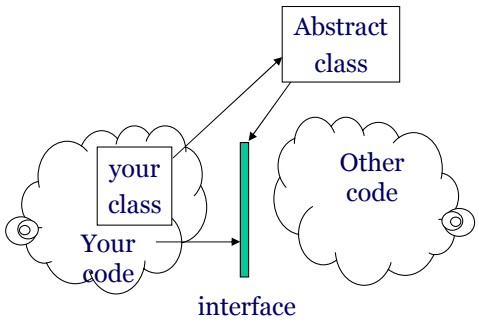
- An interface I declares a set of methods which a class must implement in order to be of type I.
- Interfaces are an important mechanism for achieving low coupling. Declare your variables (especially fields and parameters) of interface type.
- This will make your code more robust to changes in the other code
  - You cannot depend on any specific class
  - You cannot depend on any fields
- Important: The advantage of using interfaces is not for the programmer of ”Other code”, but in your code.
- Important: It is the programmer of ”Other code” which implement interfaces for you to use.



It is typically the programmer of ”Other code” who have designed the interface to be used in ”Your code”.

Abstract classes

- An abstract class is a class which has been designed to be used as a superclass.
- Often an abstract class implements an interface, but leaves out a few details for you to fill in.
  - Your code still use the interface, rather than your class.
  - The abstract class provides useful implementation of interface



As before, the programmer of other code has designed the interface, and also the abstract class.

15

The for loop of Java

As of java 5.0 is it possible to write loops of the form:

```
for( T v: i)
  … do stuff with v
```

Normally, the for loop is used to go through all elements in a collection, but it is far more general.

What the for loop really unfolds to is:

```
Iterator<T> it = i.iterator();
while(it.hasNext()){
  v = it.next();
  … do stuff with v
}
```

This means that the type of i must have an iterator method.

This is ensured by i implementing the Iterable interface.

The iterator method returns an iterator which returns elements of type T.

To let your own classes work with the for loop, several things need to be in place:

- You must define an iterator
- You must create an iterator method that returns an iterator
- You must impement the Iterable interfrace

16

Reading a file using the for loop

```
for( String s: new IterableReader( aReader ) )
    System.out.println( s );
```

```
class IterableReader implements Iterable<String>{
   private Reader reader;
   IterableReader(Reader r){
      reader = r;
   }
   public Iterator<String> iterator(){
      return new MyIterator( reader );
   }
}
```

```
class MyIterator implements Iterator<String>{
   // Inv: br is tail & nextLineToBeReturned is head
   private BufferedReader br;
   private String nextLineToBeReturned;

   private MyIterator(Reader r){
      br = new BufferedReader( r );
      readALine();
   }

   private void readALine(){
      try{
         nextLineToBeReturned = br.readLine();
      }catch(IOException uups){
         nextLineToBeReturned = null;
      }
   }
   public boolean hasNext(){
      return nextLineToBeReturned != null;
   }

   public String next(){
      String ret = nextLineToBeReturned;
      readALine();
      return ret;
   }

   public void remove(){
      throw new UnsupportedOperationException();
   }
}
```

17

The for loop in the beginning will print out all the lines in aReader.

The major part of the slide is the class MyIterator, which will iterate each line in the reader it gets as argument to its constructor.

The implementation strategy of MyIterator is very common. The problem is the following:

With an iterator, you first ask if there are any more elements (using hasNext()), and if there is, you go get it (using next()).

With a Reader, you just try to read, and if there were no elements you get a null or -1 response.

So, to turn a Reader into an iterator, we have to try to read before we answer if there are any elements. Hence, we need to keep the read line waiting for a call to next(). The private method readAline() is doing the bridging from Reader to Iterator style. It try to read a line, and if successful, stores the line so that it later can be returned by next(). If there is no line to be read, the buffered reader will return null, which in hasNext() is used as the test to see if there is a line to be returned.

Notice in particular the next() method. The line to be returned has already been read (using readALine()). Next need to return “nextLineToBeReturned”, and to move to the next line. An auxiliary variable ret is used to hold the return value while we move to the next line.

The moral of the story

- Interfaces and Abstract classes are necessary for the division of labor between
  - A) the programmers of frameworks
  - B) the programmers who use the frameworks
- The B) programmers need to understand this to use frameworks efficiently
- Interfaces are the part of contracts that the compiler understands.
- Contracts are the part you as programmer has to understand.
- Application programmers will rarely need to define interfaces and abstract classes, nor to define contracts.
- Application programmers will often need to implement interfaces or subclass abstract classes to integrate their code into the framework.
- Application programmers need to understand what pre and post conditions are for the methods called.
- Most programmers are application programmers.