OOP
Design by Contract

Carsten Schuermann

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(){…}
    …
}
```

A Crash Course in Propositional Logic

- Boolean values
- Boolean expressions
- Boolean algebra
- Truth assignements
- Interpretations
- Models
- Tautology
- Proof rules

Iterator contract I

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

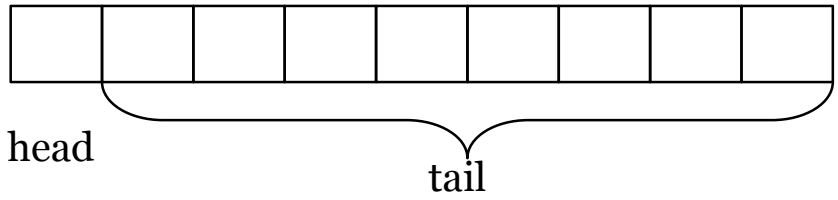
The java.util.Iterator<E> interface has the following methods :

```
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*.

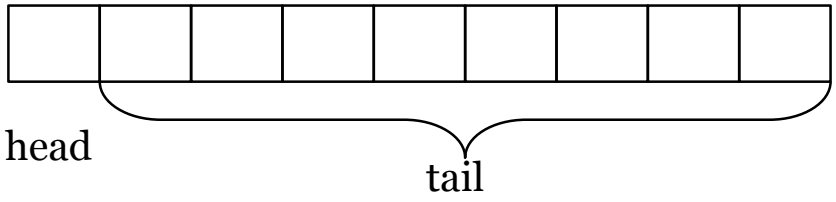
Note, given an iterator itr, itr.tail is an iterator which will return the same elements as itr, except for itr.head.

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();
}
```

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();
    }
}
```

Representation Invariants

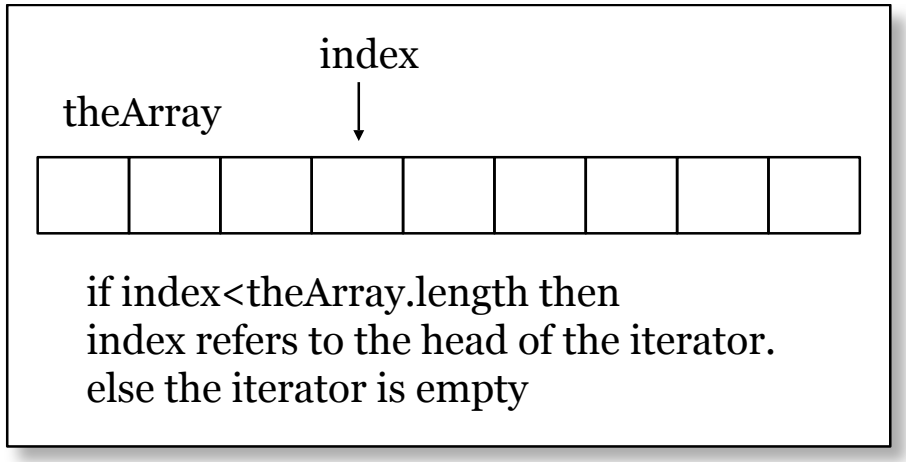
But, we cannot see if the body does the right thing in isolation from the class itself. What is the the value of index, and what is theArray?

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;
}
```



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

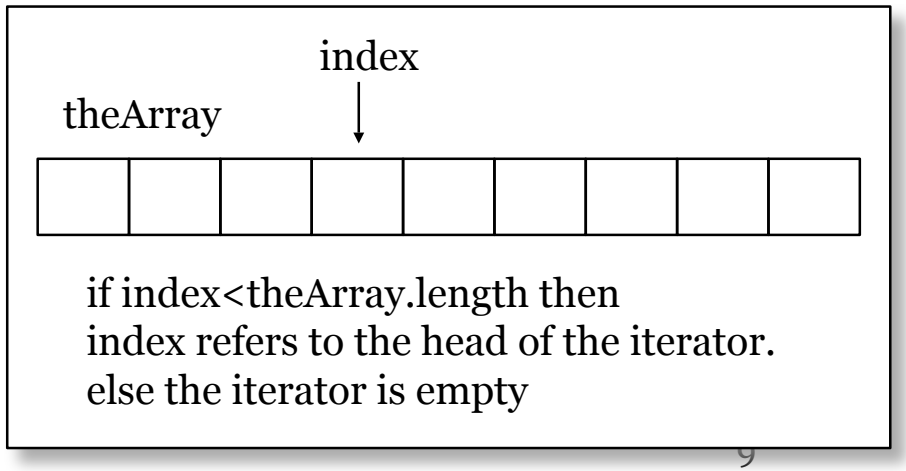
The precondition ensures that we will not attempt to iterate null.

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;
}
```



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.

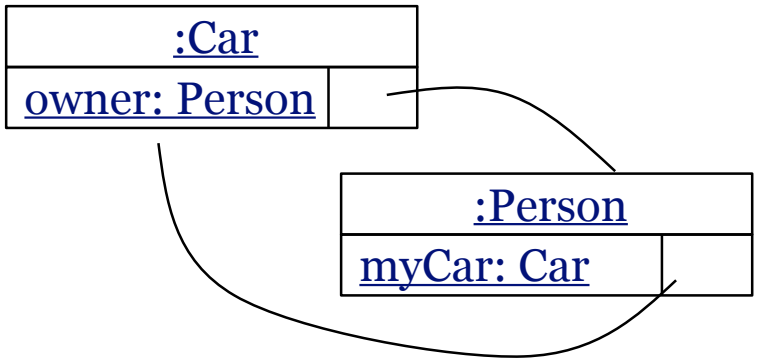
Also notice that one should be very careful with ”setter” methods for variables mentioned in the invariant. A setter for the index would be as bad as making it public

```
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();
   }
}
```

An invariant on Persons and Cars

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

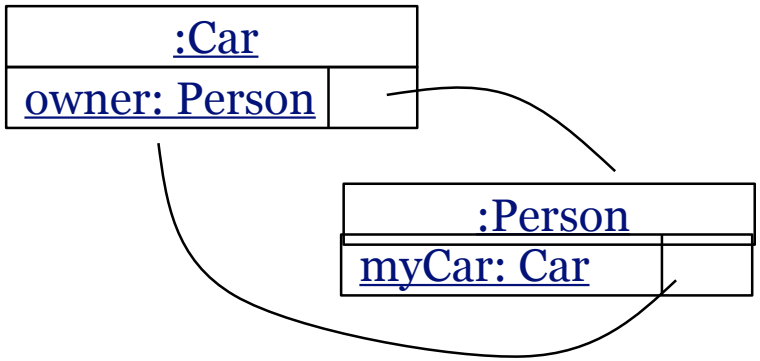
1. Each car is owned by exactly one person, that is, no car is without an owner, and no car is owned by more than one person.
2. If a person owns a car, that car is owned by that person, and vice versa.
3. A Person can own at most one car




```
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);.
    }
}
```

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;
    }
}
```

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();
    }
}
```

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

Contracts and Inheritance

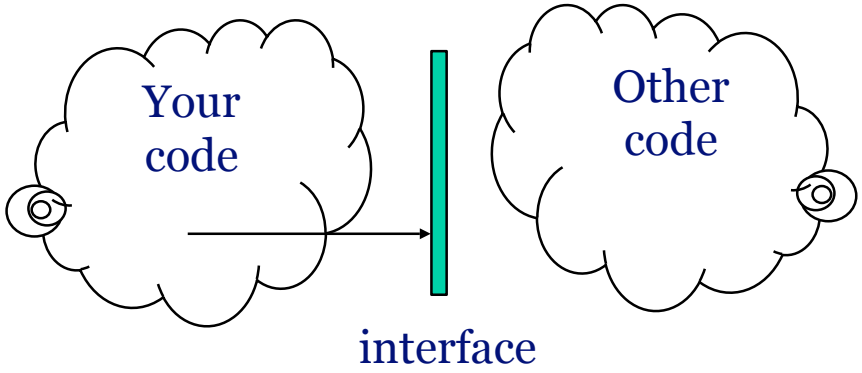
```
class SuperClass {
    int f (int y) {
            /* Pre: Psuper(y)
               Post: Qsuper(old_y, new_y, result) */
            ....
    }
}
class SubClass extends SuperClass {
    int f (int y) {
            /* Pre: Psub(y)
               Post: Qsub(old_y, new_y, result) */
            ....
    }
}
```

- Covariant in y: Psub (y) must imply Psuper(Y)
- Contraviarant in old_y, new_y, result

Qsuper(old_y, new_y, result) must imply Qsub(old_y, new_y, result)

Interfaces

- An interface I declares a set of methods which a class must implement in order to be of type I.
- 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

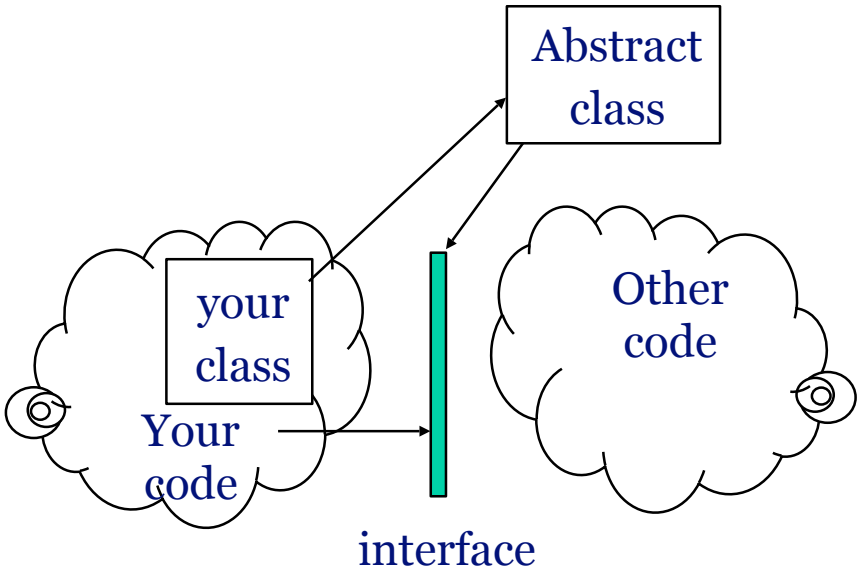


- Important: The advantage of using interfaces is not for the programmer of ”Other code”, but in your code.
- Important: It is the programmer of ”Other code” which implement interfaces for you to use.

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.

JML [Slides by Erik Poll, Nijmegen]

Formal specification language for Java

- **to specify behaviour of Java classes**
- **to record design/implementation decisions**

by adding annotations (aka assertions) to Java source code, eg

- **preconditions**
- **postconditions**
- **class invariants**

as in programming language Eiffel, but more expressive

JML (cont'd)

To make JML easy to use:

- **JML annotations are added as comments in .java file, between `/*@ ... @*/`, or after `//@`.**
- **Properties are specified as Java boolean expressions, extended with a few operators**

  ```
  \result, \forall, \old,  ==>, ...
  ```

  and a few keywords

  ```
  requires, ensures, invariant, ...
  ```

Using JML we specify and check properties of *the Java program itself*, not of *some model of our Java program*. Ie. the Java program itself *is* our formal model.

Pre and Post Conditions

Pre- and post-conditions for methods, eg.

```
/*@ requires amount >= 0;
    ensures  balance == \old(balance)-amount &&
             \result == balance;
  @*/
 public int debit(int amount) {
   ...
 }
```

Here `\old(balance)` refers to the value of `balance` before execution of the method.

Pre and Post Conditions (cont'd)

JML specs can be as strong or as weak as you want.

```
/*@ requires amount >= 0;
    ensures  true;
  @*/
 public int debit(int amount) {
   ...
 }
```

This default postcondition "`ensures true`" can be omitted.

Pre and Post Conditions (cont'd)

Pre- and postconditions define a contract between a class and its clients:

- **Client must ensure precondition and may assume postcondition**
- **Method may assume precondition and must ensure postcondition**

Eg, in the example spec for `debit`, it is the obligation of the client to ensure that `amount` is positive.

The `requires` clause makes this explicit.

Signals

`signals` clauses specify when exceptions may be thrown

```
/*@ requires amount >= 0;
    ensures  true;
    signals (ISOException e)
              amount > balance          &&
              balance == \old(balance) &&
              e.getReason()==AMOUNT_TOO_BIG;
  @*/
 public int debit(int amount) throws ISOExceptio
  ...
 }
```

Signals (cont'd)

Again, specs can be as strong or weak as you want.

```
/*@ requires amount >= 0;
    ensures  true;
    signals  (ISOException) true;
  @*/
 public int debit(int amount) throws ISOExceptio
```

NB this specifies that an `ISOException` is the *only* exception that can be thrown by `debit`

Signals (cont'd)

Exceptions mentioned in throws clause are allowed by default, i.e. the default signals clause is

```
signals (Exception) true;
```

To rule them out, add an explicit

```
signals (Exception) false;
```

or use the keyword normal_behavior

```
/*@ normal_behavior
      requires ...
      ensures  ...
  @*/
```

Signals (cont'd)

There is often a trade-off between precondition and exceptional postcondition

```
/*@ requires amount >= 0 && amount <= balance;
    ensures  true;
    signals  (ISOException e) false;
  @*/
 public int debit(int amount) throws ISOExceptio
  ...
 }
```

*Maybe "**throws ISOException**" should now be omitted.*

Invariants

Invariants (aka *class* invariants) are properties that must be maintained by all methods, eg

```
public class  Wallet {
 public static final short MAX_BAL = 1000;
 private short balance;
  /*@ invariant 0 <= balance
                  && balance <= MAX_BAL;
    @*/
 ...
```

Invariants (con'td)

Invariants document design decisions.

```
private final Object[] objs;
/*@ invariant
    objs != null
    &&
    objs.length == CURRENT_OBJS_SIZE
    &&
    (\forall int i; 0 <= i && i <= CURRENT_OBJS_SIZE
                  ; objs[i] != null);
  @*/
```

Making these design decisions explicit helps in understanding the code.

Null Objects

Many invariants, pre- and postconditions are about references not being `null`. non_null is a convenient short-hand for these.

```
public class Directory {

  private /*@ non_null @*/ File[] files;

  void createSubdir(/*@ non_null @*/ String name){
    ...
  Directory /*@ non_null @*/ getParent(){
    ...
```

Assertions

An `assert` clause specifies a property that should hold at some point in the code, eg.

```
if (i <= 0 || j < 0) {
      ...
} else if (j < 5) {
      //@ assert i > 0 && 0 < j && j < 5;
      ...
} else {
      //@ assert i > 0 && j > 5;
      ...
}
```

Assertions (cont'd)

JML keyword `assert` now also in Java (since Java 1.4).

Still, assert in JML is more expressive, for example in

```
  ...
  for (n = 0; n < a.length; n++)
       if (a[n]==null) break;
 /*@ assert (\forall int i; 0 <= i && i < n;
                                 a[i] != null);
   @*/
```

Mutable Store

Frame properties limit possible side-effects of methods.

```
/*@   requires amount >= 0;
     assignable balance;
        ensures balance == \old(balance)-amount;
 @*/
public int debit(int amount) { }
  ...
```

E.g., `debit` can *only* assign to the field `balance`.
NB this does *not* follow from the post-condition.

Default assignable clause: `assignable \everything`.

Side Effects

A method without side-effects is called pure.

```
public /*@ pure @*/ int getBalance(){...

Directory /*@ pure non_null @*/ getParent(){...}
```

Pure method are implicitly `assignable \nothing`.

Pure methods, and only pure methods, can be used *in* specifications, eg.

```
//@ invariant 0<=getBalance() && getBalance()<=MAX_BALANCE
```

JML Summary

This covers all you need to know to start using JML!

The JML keywords discussed so far:

```
requires ensures signals invariant
non_null normal_behavior assignable
pure
```

and JML operators

```
\old, \forall, \exists, \result
```

There are many more features in JML, but these depend on which tool for JML you use.

Extended Static Checking (Cok,Kiniry,Poll)

ESC/Java(2)

- **extended static checking = fully automated program verification, with some compromises to achieve full automation**
- ***tries* to *prove* correctness of specifications, at compile-time, fully automatically**
- ***not sound*: ESC/Java may miss an error that is actually present**
- ***not complete*: ESC/Java may warn of errors that are impossible**
- **but finds lots of potential bugs quickly**
- **good at proving absence of runtime exceptions (eg Null-, ArrayIndexOutOfBounds-, ClassCast-) and verifying relatively simple properties.**

Extended Static Checking

One of the assertions below is wrong:

```
if (i <= 0 || j < 0) {
    ...
} else if (j < 5) {
    //@ assert i > 0 && 0 < j && j < 5;
    ...
} else {
    //@ assert i > 0 && j > 5;
    ...
}
```

Runtime assertion checking *may* detect this with a comprehensive test suite.
ESC/Java2 *will* detect this at compile-time.

Extended Static Checking

Important differences:

- **ESC/Java2 checks specs at compile-time, jmlrac checks specs at run-time**
- **ESC/Java2 proves correctness of specs, jml only tests correctness of specs. Hence**
  - **ESC/Java2 independent of any test suite, results of runtime testing only as good as the test suite,**
  - **ESC/Java2 provides higher degree of confidence.**

  The price for this: you have to specify all pre- and postconditions of methods (incl. API methods) and invariants needed for modular verification

ESC/Java2 Example

```
class Bag {
  int[] a;
  int n;

  Bag(int[] input) {
    n = input.length;
    a = new int[n];
    System.arraycopy(input, 0, a, 0, n);
  }

  int extractMin() {
    int m = Integer.MAX_VALUE;
    int mindex = 0;
    for (int i = 1; i <= n; i++) {
      if (a[i] < m) {
        mindex = i;
        m = a[i];
      }
    }
    n--;
    a[mindex] = a[n];
    return m;
  }
}
```

ESC Java 2/Example (con't)

```
class Bag {
  /*@ non_null */ int[] a;
  int n;
  //@ invariant 0 <= n && n <= a.length;
  //@ ghost public boolean empty;
  //@ invariant empty == (n == 0);

  //@ requires input != null;
  //@ ensures this.empty == (input.length == 0);
  public Bag(int[] input) {
    n = input.length;
    a = new int[n];
    System.arraycopy(input, 0, a, 0, n);
    //@ set empty = n == 0;
  }

  //@ ensures \result == empty;
  public boolean isEmpty() {
    return n == 0;
  }

  //@ requires !empty;
  //@ modifies empty;
  //@ modifies n, a[*];
  public int extractMin() {
    int m = Integer.MAX_VALUE;
    int mindex = 0;
    for (int i = 0; i < n; i++) {
      if (a[i] < m) {
        mindex = i;
        m = a[i];
      }
    }
    n--;
    //@ set empty = n == 0;
    //@ assert empty == (n == 0);
    a[mindex] = a[n];
    return m;
  }
}
```

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.