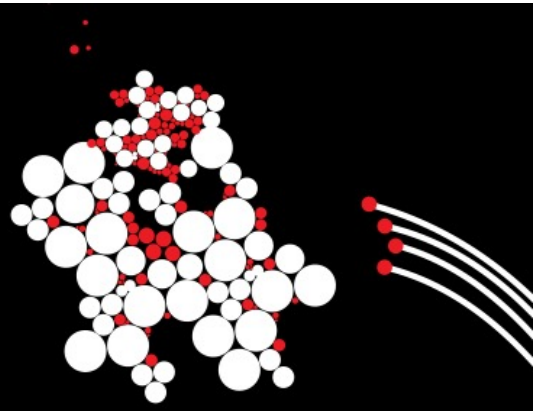
UNIVERSITY OF TWENTE.



JML: Program Specifications

Advanced Programming

Lecturer: Marieke Huisman

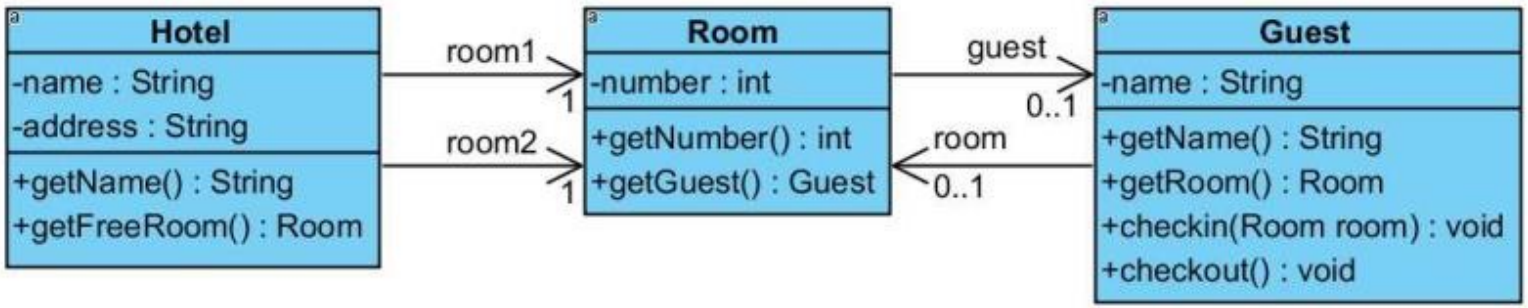
PROGRAMMING OVERVIEW

Week 1 (IP) Procedural programming Arrays Debugging	**Week 2 (IP)** Classes and Objects Documenting and Testing Exceptions (AP)	**Week 3 (IP)** Interfaces, Inheritance, Polymorphism Subtyping Streams and Files (AP)
Week 4 (AP) Collections: Set, List, Map Specification with JML Material: appendix B	**Week 5 (AP)** Design Patterns User Interfaces	**Week 6 (AP)** Basic Concurrency Project kick-off
Week 7 (AP) Basic Networking Defensive programming Security	**Week 8/9 (Project)** Project	**Week 10 (Project)** Project Exam

PROGRAM DESIGN

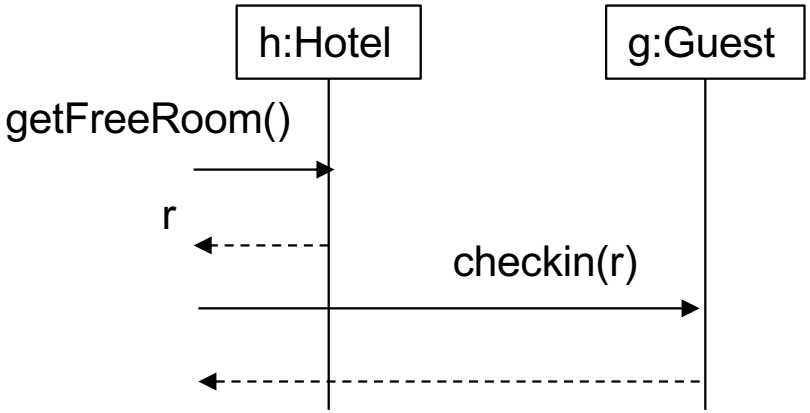
- Consists of classes and their relationships
- How do they interact with each other?

Example: Hotel Information System



OBJECT INTERACTIONS

```
Hotel h = new Hotel("Fawlty Towers");
Guest g = new Guest("Major Gowen");
g.checkin(h.getFreeRoom());
```



Implicit assumptions:

- Result of getFreeRoom is an empty room
- After checkin, guest is signed in into room

CLASS SPECIFICATIONS

PRECONDITIONS, POSTCONDITIONS AND CLASS INVARIANTS

For each class and each method the program designer must specify the conditions for objects of this class to work properly!

- Preconditions: Which conditions should hold before a method is called, for it to work correctly?
- Postconditions: What conditions are satisfied once the method has finished correctly?
- Class invariants: What are the conditions that must always hold in an object of a class?

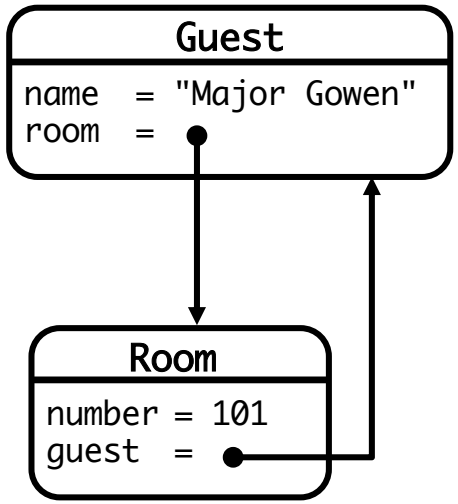
GUEST CLASS SPECIFICATION

- Method `checkin(Room room)`
  - Precondition: Room is empty

    (`room.getGuest() == null`)
  - Postcondition: Guest related to room is this guest

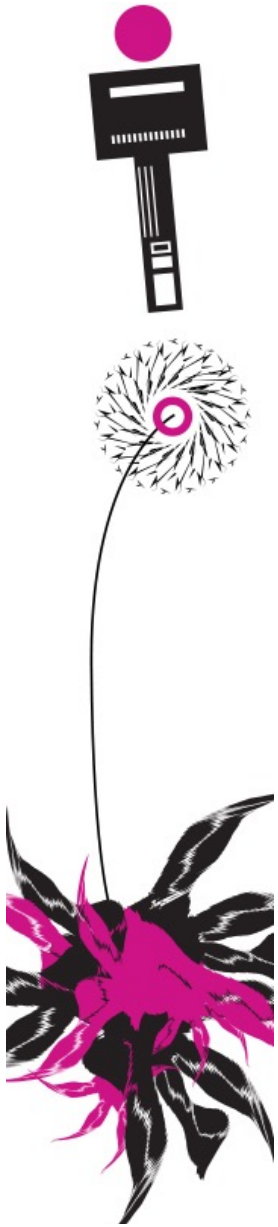
    (`room.getGuest() == this`)
- Class Invariant: If room attribute is not `null` then the guest related to the room is this guest

  (`room != null ==> room.getGuest() == this`)



ADVANTAGES OF FORMAL SPECIFICATIONS

- Makes the implicit assumptions explicit
- Explicit separation between the ‘what’ and the ‘how’
- Unambiguous
- Can be checked by tools
  - During execution,
  - Statically, as separate check
- Specifications independent of implementation!



Preconditions

WHAT IS A PRECONDITION?

RESPONSIBILITY OF THE CALLER

A precondition belongs to a method

- Condition that should always hold when a method is called
  - Caller has to ensure this
  - Method implementation can rely on this
- Condition expressed using method parameters and/or `public` methods
- Typical preconditions
  - Restriction on method arguments
  - Requiring the object to be in a particular state

DIVISION

```
public int myDivision(int x, int y) {
        return x/y;
  }
```

What do we need to make this work?

```
requires y != 0;
```

COURSE EXAMPLE

```
public class Course {

    private int nrOfStudents;

    public Course() {
         nrOfStudents = 0;
    }

    public int getNrOfStudents() {
          return nrOfStudents;
    }

    public void enroll() {
          nrOfStudents++;
    }

    public void batchEnroll(int value) {
              this.nrOfStudents = value;
    }
}
```

The number of students in a course can never be negative. What should be the precondition of `batchEnroll(int value)`?

Answer

```
requires value >= 0;
```

COURSE EXAMPLE EXTENDED

Suppose we add to class `Course` a constant `MAX` to define the maximum number of students enrolled in a class

- How to update the precondition of `batchEnroll()`?

Answer

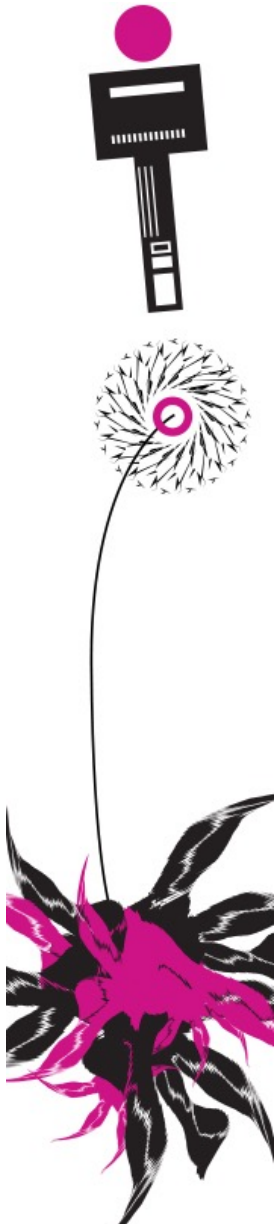
```
requires value >= 0 && value <= Course.MAX;
```

- What is the precondition for `enroll()`?

Answer

```
requires getNrOfStudents() < Course.MAX;
```

`getNrOfStudents()` should be pure!



Postconditions

DEFINITION OF POSTCONDITION

RESPONSIBILITY OF THE CALLED OBJECT (IMPLEMENTER)

A postcondition belongs to a method

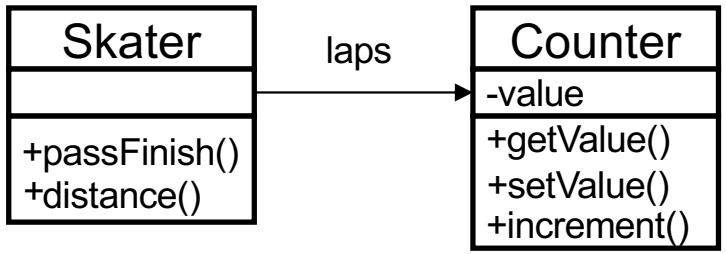
- Condition that should always hold after the method terminates
  - Method implementation should ensure this
  - Caller can rely on this
- Condition expressed using method result, method arguments, and/or pure `public` methods

POSTCONDITION OF MAX

```
public int max (int x, int y) {
    if x >= y {
        return x;
    }
    else {
        return y;
    }
}
```

```
ensures \result >= x && \result >= y;
ensures \result == x || \result == y;
```

POSTCONDITIONS FOR COUNTER



What should be the postcondition of `getValue()`?

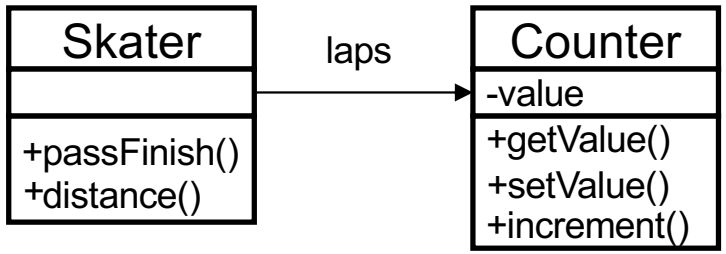
Answer

It should return a positive number!

```
ensures \result >= 0;
```

```
public class Skater {

    private Counter laps;

    public static final int LAPSIZE = 600;

    public void passFinish() {
        laps.increment();
    }

    public int distance() {
        return laps.getValue() * LAPSIZE;
    }
}
```

POSTCONDITION




```
public class Skater {

    private Counter laps;

    public static final int LAPSIZE = 600;

    public void passFinish() {
        laps.increment();
    }

    public int distance() {
        return laps.getValue() * LAPSIZE;
    }
}
```

What should be the postcondition of `increment()`?

Answer

The result of getValue() should equal to the result of getValue() before the call plus 1

```
ensures getValue() == \old(getValue()) + 1;
```

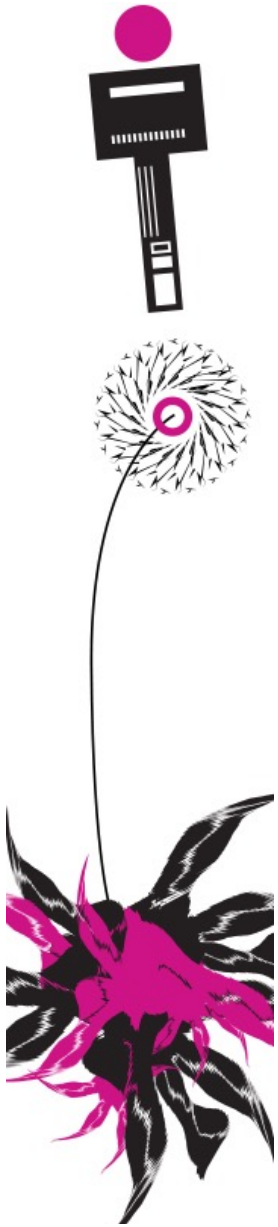
POSTCONDITION EXPRESSIONS

```
//@ ensures (\forall int i; 0 <= i && i < a.length; a[i] == 0);
 public void reset(int a[]) {
   for (int i = 0; i < a.length; i++) {
        a[i] = 0;
   }
}
```

SPECIFICATION EXPRESSIONS

- Preconditions, postconditions and invariants are all boolean conditions
- Some special expression notation
  - `\result` refers to the return value of a method
  - `\old(`*`<expr>`*`)` is the value of *`<expr>`* before calling the method
  - Basic logical operators (`&&`, `||`, `!=`) and implication (`==>`)
  - Universal and existential quantification may be used:

    `(\forall` *`<type>`* `x; ...)`, `(\exists` *`<type>`* `x; ...)`



Class invariants

DEFINITION OF CLASS INVARIANT

- A class invariant is defined in the scope of a class (not a method)
- General idea: a property that always holds
  - Allowed values for instance and static variables
- In practice: might be broken temporarily in the middle of a method (but has to be re-established)

invariant diameter = 2 * radius;

Two usage scenarios

1. Can refer to internal state of the object (useful for implementer)
2. Can also serve as documentation of the behaviour of a class (useful for caller)

COUNTER CLASS EXAMPLE

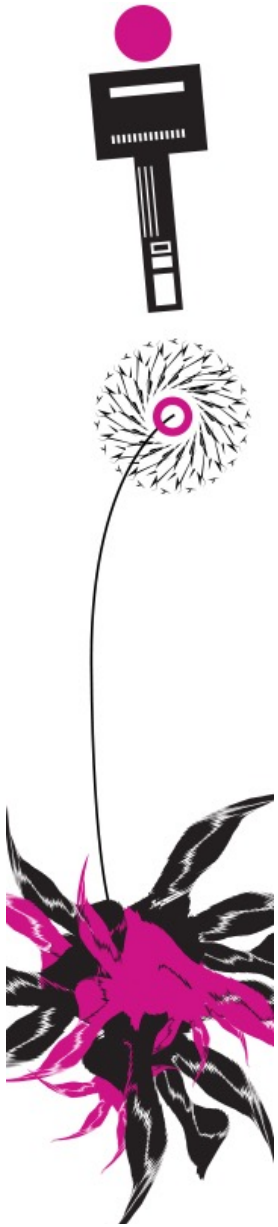
```
public class Counter {

    /*@
     *    invariant value >= 0;
     */

    private int value;

    public Counter() {
         value = 0;
    }

    public int getValue() {
          return value;
    }

    public void setValue(int value) {
          this.value = value;
    }

    public void increment() {
          value++;
    }
}
```

Private invariant: refers to internal state, not visible to caller

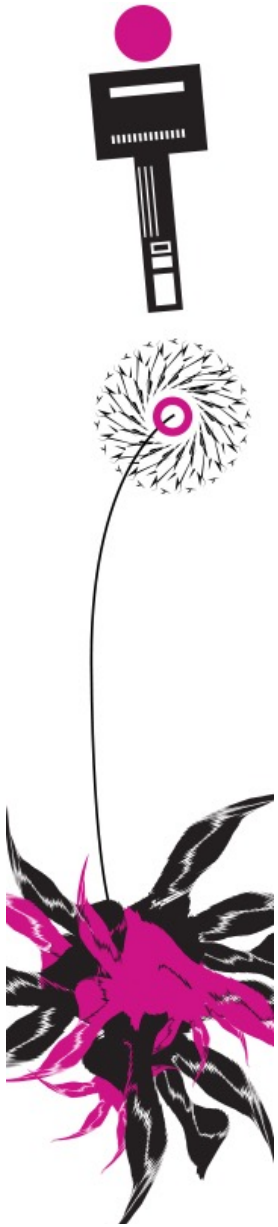
COUNTER CLASS EXAMPLE

```
public class Counter {

 /*@
  *public invariant getValue() >= 0;
  */

   private int value;

   public Counter() {
        value = 0;
   }

   public int getValue() {
         return value;
   }

   public void setValue(int value) {
         this.value = value;
   }

   public void increment() {
         value++;
   }
}
```

Public invariant: refers to publicly visible methods, documents class behaviour



JML



WHAT IS JML?

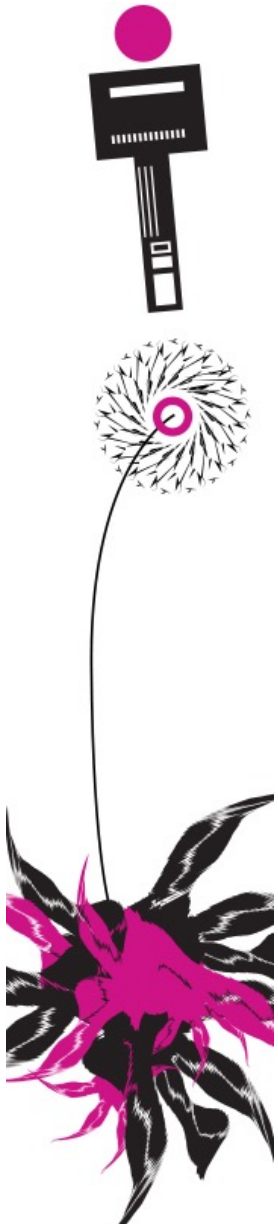
- Java Modeling Language, see jmlspecs.org
- Specifications written as special comments in the code

/*@ … */

//@

No space between the comment start and the @!

- Full language has many more constructs
- Large variety of tools that understand JML (or dialect)



Now it’s your turn!

Participate at

https://www.wooclap.com/OMQRII

CONSIDER THIS STUB FOR CLASS STUDENT

```
public class Student {
            public static final int bachelor = 0;
            public static final int master = 1;

            private String name;
            private int credits;
            private int status;

            public void changeStatus(){
            }
            public void setName(String n) {
            }

            /*@ pure */ public String getName() {
                        return "";
            }
            /*@ pure */ public int getCredits() {
                        return 0;
            }
            /*@ pure */  public int getStatus() {
                        return 0;
            }
```

WHEN CAN CHANGESTATUS() BE CALLED?

```
/*@ requires getCredits() >= 180;
  @ requires getStatus() == bachelor;
  @ ensures getCredits() == \old(getCredits());
  @ ensures getStatus() == master;
*/
public void changeStatus(){
}
```

a. Always
b. Whenever the student has obtained 180 credits or more
c. When the student is a master student, and has obtained 180 credits or more
d. When the student is a bachelor student, and has obtained 180 credits or more

WHAT DOES METHOD CHANGESTATUS() DO?

```
/*@ requires getCredits() >= 180;
  @ requires getStatus() == bachelor;
  @ ensures getCredits() == \old(getCredits());
  @ ensures getStatus() == master;
*/
public void changeStatus(){
}
```

a. Update the credits of the students
b. Change the status of the student to master
c. Status to master, credits unchanged
d. Status to master, credits updated

WHAT IS NOT SPECIFIED HERE?

```
/*@ requires getCredits() >= 180;
  @ requires getStatus() == bachelor;
  @ ensures getCredits() == \old(getCredits());
  @ ensures getStatus() == master;
*/
public void changeStatus(){
}
```

a. Students must always have 180 credits or more
b. After changeStatus(), a student always has 180 credits or more
c. changeStatus() will not change the credits
d. After changeStatus(), a student has master status

WHAT IS CAPTURED BY THIS SPECIFICATION?

```
/*@ requires n != null;
  @ ensures getName().equals(n);
*/
public void setName(String n) {
}
```

a. Field name is always non-null
b. Field name has same contents as string n
c. Field name points to string n
d. Field name is a constant (and can never be changed)

WHAT IS A GOOD SPECIFICATION FOR CONSTRUCTOR?

```
public class Point {
    private int x;
    private int y;

    public Point(int n, int m) {
        x = n;
        y = m;
    }
    public int getX() { return x; }
    public int getY() { return y; }
}
```

a. requires n == 0 && m == 0;
b. requires x < n && y < m;
c. ensures x == n && y == m;
d. ensures x < n && y < m;

WHICH IMPLEMENTATION DOES NOT RESPECT THIS SPECIFICATION?

```
public int x;

/*@ requires n >= 0;
      ensures x >= \old(x);
*/
public void inc(n) {
   // body goes here
}
```

a. x = n;
b. x = x + n;
c. x = x + 1;
d. x = x + 2*n;

WHAT IS A CORRECT SPECIFICATION FOR *POWER*?

```
public int power(int e1, int e2) {
    int res = 1;
    int count = 0;
    if (e1 > 0) {
        while (count < e2) {
            res = res * e1;
            count = count + 1;
        }
        return res;
    } else {
        return 1;
    }}
```

a. ensures \result >= 1;

b. ensures \result == e1 + e2;

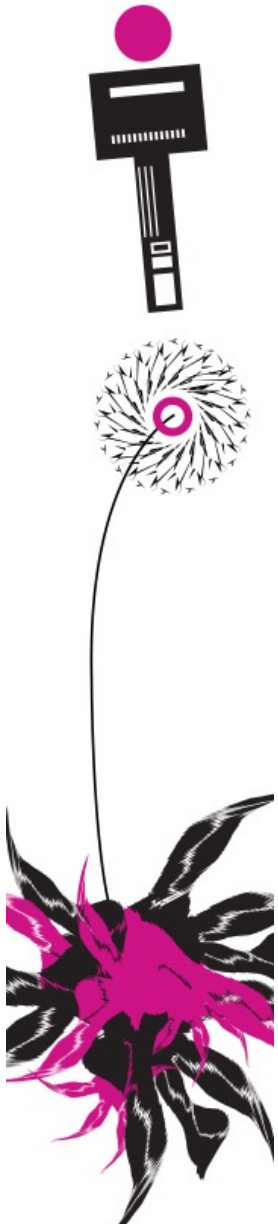
c. ensures res >= 0;

d. requires e1 == e2;

WHAT SPECIFICATION IS CHECKED BY THE ASSERTS?

```
public int compute (int m, int n) {
    assert m >= 0;
    assert n >= 0;
    return <complicated expression using m and n>
}
```

a. requires m >= 0 && n >= 0;
b. ensures m >= 0 && n >= 0;
c. requires m >= 0 || n >= 0;
d. ensures m >= 0 || n >= 0;



List specifications

List<E> QUERIES

PACKAGE java.utils

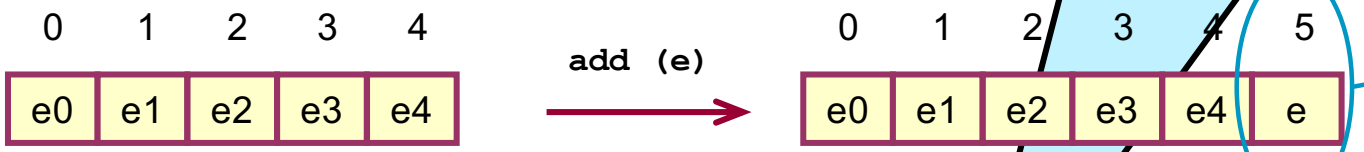
- Number of elements: `public int size()`
  - Postcondition: `ensures \result >= 0;`
- Is list empty? `public boolean isEmpty()`
  - Postcondition: `ensures \result == (this.size() == 0);`
- Get element at index `i`: `public E get(int i)`
  - Precondition: `requires 0 <= i && i < this.size();`

List size cannot be negative

Index *i* should be inside the list!

List<E> COMMANDS: ADD

Append element: `public void add(E e)`



List
- grows one position,
- new element *e* is added to the end of the list
- all other elements remain the same

Extra element!

Postcondition

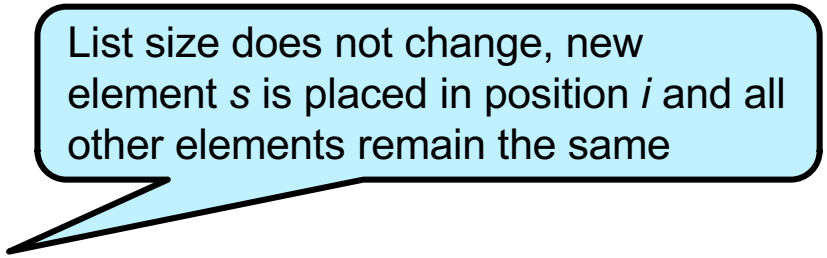
```
ensures this.size() == \old(size()) + 1;
ensures this.get(this.size() - 1).equals(e);
ensures (\forall int i; 0 <= i && i < \old(size());
                this.get(i).equals(\old(this.get(i)));
```

List<E> COMMANDS: SET

Set element at index i: `public void set(int i, E e)`

Precondition:

```
requires 0 <= i && i < this.size();
```



Postcondition:

```
ensures this.size() == old.size();
ensures this.get(i).equals(e);
ensures (\forall int j; 0 <= j & j < this.size() & j != i;
                this.get(j).equals(\old(this.get(j))));
```

List<E> COMMANDS: CONTAINS

Does the list contain element e: `public boolean contains(E e)`

Postcondition:

```
ensures \result == (\exists int j; 0 <= j & j < this.size();
                          this.get(j).equals(e));
ensures (\forall int j; 0 <= j & j < this.size();
                this.get(j).equals(\old(this.get(j))));
ensures this.size() == \old(this.size());
```

Could be captured by specifying that the method is pure

HOW TO SPECIFY SEARCH?

```
public int search(E e)
```

Postcondition:

```
ensures \result == -1 <==>
        (\forall int j; 0 <= j & j < this.size();
                        !this.get(j).equals(e));
ensures 0 <= \result && \result < this.size  <==>
          this.get(\result).equals(e));
ensures (\forall int j; 0 <= j & j < this.size();
             this.get(j).equals(\old(this.get(j)));
ensures this.size() == \old(this.size());
```

HOW TO SPECIFY BINARY SEARCH?

```
public int bsearch(E e)
```

Precondition: `requires sorted();`

Postcondition:

```
ensures \result == -1 <==>
        (\forall int j; 0 <= j & j < this.size();
                        !this.get(j).equals(e));
ensures 0 <= \result && \result < this.size  <==>
           this.get(\result).equals(e));
ensures (\forall int j; 0 <= j & j < this.size();
              this.get(j).equals(\old(this.get(j)));
ensures this.size() == \old(this.size());
```

HOW TO SPECIFY SORTING?

This specification holds for any sorting algorithm

Postcondition sorted property:

```
ensures (\forall int i; 0 <= i && i < this.size();
         (\forall int j; 0 < j && j < this.size();
                      i < j ==> this.get(i) <= this.get(j)));
```

Alternative formulation

```
ensures (\forall int i; 0 <= i && i < this.size() - 1;
                        this.get(i) <= this.get(i+1));
```

Can we specify more?

Permutation property

SORTING: PERMUTATION PROPERTY

```
ensures (\forall int i; 0 <= i && i < this.size();
          (\exists int j; 0 < j && j < this.size();
              (this.get(i)).equals(\old(this.get(j))))) &&
         (\forall int i; 0 <= i && i < this.size();
          (\exists int j; 0 < j && j < this.size();
              (\old(this.get(i))).equals(this.get(j)))));
```

SUMMARY

- JML specifications: make the implicit assumptions explicit
- Preconditions, postconditions, class invariants
- Unambiguous
- Can be checked by tools
  - During execution,
  - Statically, as separate check
- Specifications independent of implementation!