

# Design by Contract with JML

Design by contract

Java Modeling Language (JML)

Formal specifications in JML

JML tools – JML compiler (jmlc)

Thanks to Gary Leavens for allowing us to adapt his lecture notes.

# Design by Contract (DBC)

A way of recording:

- Details of method responsibilities
- Avoiding constantly checking arguments
- Assigning blame across interfaces

# Contracts in Software

```
/*@ requires x >= 0.0;  
  @ ensures JMLDouble.approximatelyEqualTo(x,  
  @       \result * \result, eps);  
  @*/  
public static double sqrt(double x) { ... }
```

	Obligations	Rights
Client	Passes non-negative number	Gets square root approximation
Implementor	Computes and returns square root	Assumes argument is non-negative

# Pre and Postconditions

## Definition

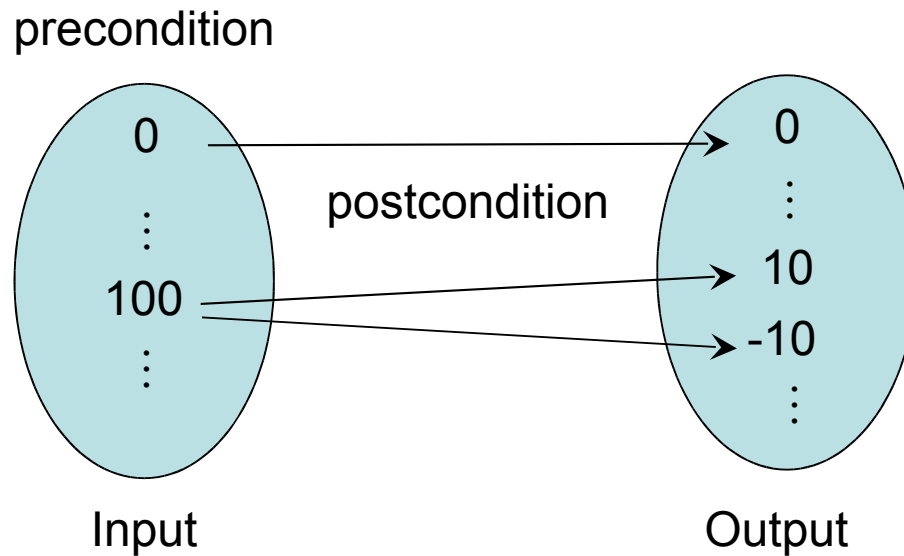
- A method's *precondition* says what must be true to call it.
- A method's *normal postcondition* says what is true when it returns normally (i.e., without throwing an exception).
- A method's *exceptional postcondition* says what is true when a method throws an exception.

```
/*@ signals (IllegalArgumentException e) x < 0;  
@*/
```

# Relational Model of Methods

Can think of a method as a relation:

Inputs  $\leftrightarrow$  Outputs



# Contracts as Documentation

For each method say:

- What it requires (if anything), and
- What it ensures.

Contracts are:

- More abstract than code,
- Not necessarily constructive,
- Often machine checkable, so can help with debugging, and
- Machine checkable contracts can always be up-to-date.

# Abstraction by Specification

A contract can be satisfied in many ways:

E.g., for square root:

- Linear search
- Binary search
- Newton's method
- ...

These will have varying non-functional properties

- Efficiency
- Memory usage

So, a contract abstracts from all these implementations, and thus can change implementations later.

# More Advantages of Contracts

## Blame assignment

- Who is to blame if:
  - Precondition doesn't hold?
  - Postcondition doesn't hold?

## Avoids inefficient defensive checks

```
//@ requires a != null && (* a is sorted *);  
public static int binarySearch(Thing[] a, Thing x) { ... }
```



# Modularity of Reasoning

Typical OO code:

...

```
source.close();
```

```
dest.close();
```

```
getFile().setLastModified(loc.modTime().getTime());
```

...

How to understand this code?

- Read the code for all methods?
- Read the contracts for all methods?

# Rules for Reasoning

## Client code

- Must work for every implementation that satisfies the contract, and
- Can thus only use the contract (not the code!), i.e.,
  - Must establish precondition, and
  - Gets to assume the postcondition

```
//@ assert 9.0 >= 0;  
double result = sqrt(9.0);  
//@ assert result * result ≈ 9.0; // can assume result == 3.0?
```

## Implementation code

- Must satisfy contract, i.e.,
  - Gets to assume precondition
  - Must establish postcondition
- But can do anything permitted by it.

# Contracts and Intent

Code makes a poor contract, because can't separate:

- What is intended (contract)
- What is an implementation decision
  - E.g., if the square root gives an approximation good to 3 decimal places, can that be changed in the next release?

By contrast, contracts:

- Allow vendors to specify intent,
- Allow vendors freedom to change details, and
- Tell clients what they can count on.

Question

- What kinds of changes might vendors want to make that don't break existing contracts?

# JML

## What is it?

- Stands for “Java Modeling Language”
  - A formal behavioral interface specification language for Java
- Design by contract for Java
- Uses Java 1.4 or later
- Available from [www.jmlspecs.org](http://www.jmlspecs.org)

# Annotations

JML specifications are contained in annotations, which are comments like:

//@ ...

or

/\*@ ...

@ ...

@\*/

At-signs (@) on the beginning of lines are ignored within annotations.

## Question

- What's the advantage of using annotations?

# Informal Description

An informal description looks like:

(\* some text describing a property \*)

- It is treated as a boolean value by JML, and
- Allows
  - Escape from formality, and
  - Organize English as contracts.

```
public class IMath {  
    /*@ requires (* x is positive *);  
    @ ensures \result >= 0 &&  
    @ (* \result is an int approximation to square root of x *)  
    @*/  
    public static int isqrt(int x) { ... }  
}
```

# Exercise

Write informal pre and postconditions for methods of the following class.

```
public class Person {
    private String name;
    private int weight;

    /*@ also
     @ ensures \result != null &&
     @ (* \result is a displayable
     @ form of this person *);
    public String toString() {
        return "Person(\" + name +
            "\", " + weight + ")";
    }

    public int getWeight() {
        return weight;
    }
}
```

```
public void addKgs(int kgs) {
    if (kgs >= 0) {
        weight += kgs;
    } else {
        throw new IllegalArgumentException();
    }
}

public Person(String n) {
    name = n; weight = 0;
}
}
```

# Formal Specifications

Formal assertions are written as Java expressions, but:

- Cannot have side effects
  - No use of =, ++, --, etc., and
  - Can only call *pure* methods.
- Can use some extensions to Java:

Syntax	Meaning
\result	result of method call
$a ==> b$	$a$ implies $b$
$a <== b$	$b$ implies $a$
$a <==> b$	$a$ iff $b$
$a <=!> b$	$\neg(a <==> b)$
\old(E)	value of $E$ in pre-state



# Example

```
// File: Person.refines-java
//@ refine "Person.java"
public class Person {
    private /*@ spec_public non_null @*/ String name;
    private /*@ spec_public @*/ int weight;

    /*@ public invariant !name.equals("") && weight >= 0;

    /*@ also
       @ ensures \result != null;
       @*/
    public String toString();

    /*@ also ensures \result == weight;
    public int getWeight();

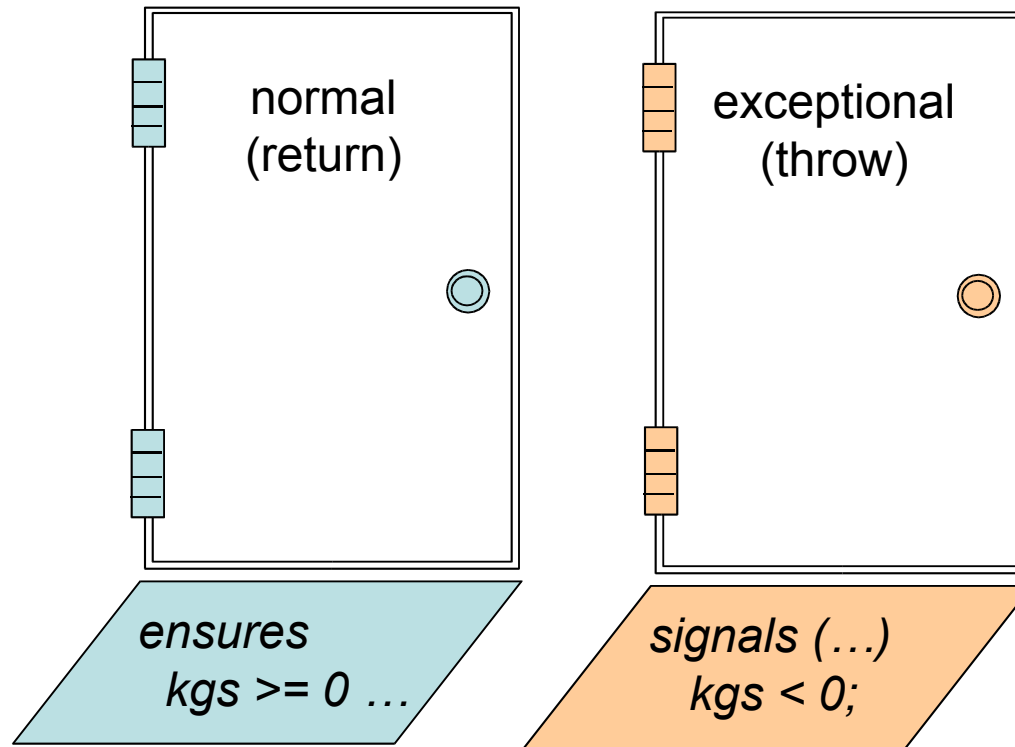
    <<continues to next slide>>
```

# Example (Cont.)

```
/*@ also
  @ ensures kgs >= 0 && weight == \old(kgs + weight);
  @ signals (Exception e) kgs < 0 &&
  @           (e instanceof IllegalArgumentException);
  @*/
public void addKgs(int kgs);

/*@ also
  @ requires !n.equals("");
  @ ensures n.equals(name) && weight == 0;
  @*/
public Person(/*@ non_null @*/ String n);
}
```

# Meaning of Postconditions



# Invariants

## Definition

- An *invariant* is a property that is always true of an object's state (when control is not inside the object's methods).

## Invariants allow you to define:

- Acceptable states of an object, and
- Consistency of an object's state.

```
//@ public invariant !name.equals("") && weight >= 0;
```

# Exercise

Formally specify the following method (in Person)

```
public void changeName(String newName) {  
    name = newName;  
}
```

Hint: watch out for the invariant!

# Quantifiers

JML supports several forms of quantifiers

- Universal and existential (`\forall` and `\exists`)
- General quantifiers (`\sum`, `\product`, `\min`, `\max`)
- Numeric quantifier (`\num_of`)

`(\forall Student s; juniors.contains(s); s.getAdvisor() != null)`

`(\forall Student s; juniors.contains(s) ==> s.getAdvisor() != null)`

# Exercise

Formally specify the missing part, i.e., the fact that `a` is sorted in ascending order.

```
/*@ old boolean hasx = (\exists int i; i >= 0 && i < a.length; a[i] == x);
   @ requires
   @
   @ ensures (hasx ==> a[\result] == x) && (!hasx ==> \result == -1);
   @ requires_redundantly (* a is sorted in ascending order *);
   @*/
public static int binarySearch(/*@ non_null @*/ int[] a, int x) { ... }
```

Hint: use a nested quantification!

# Model Declarations

What if you want to change a `spec_public` field's name?

```
private /*@ spec_public non_null @*/ String name;
```

to

```
private /*@ non_null @*/ String fullName;
```

For specification:

- need to keep the old name public
- but don't want two strings.

So, use a model field:

```
//@ public model non_null String path;
```

and a represents clause

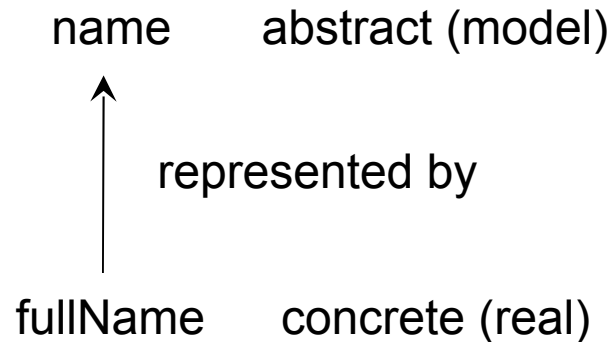
```
//@ private represents path <- fullName;
```



# Model Variables

Are specification-only variables

- Like domain-level constructs
- Given value only by represents clauses:



# Question

What changes would you make to change the representation of a person's weight from kilograms to pounds?

# Tools for JML

JML compiler (jmlc)

JML/Java interpreter (jmlrac)

JML/JUnit unit test tool (jmlunit)

HTML generator (jmldoc)

# JML Compiler (jmlc)

## Basic usage

```
$ jmlc Person.java  
    produces Person.class
```

```
$ jmlc -Q *.java  
    produces *.class, quietly
```

```
$ jmlc -d ../bin Person.java  
    produces ../bin/Person.class
```

# Running Code Compiled with jmlc

Must have JML's runtime classes  
(jmlruntime.jar) in Java's boot class path

Automatic if you use script jmlrac, e.g.,

```
$ jmlrac PersonMain
```

# A Main Program

```
public class PersonMain {  
    public static void main(String[] args) {  
        System.out.println(new Person(null));  
        System.out.println(new Person(""));  
    }  
}
```

# Example (Formatted)

```
$ jmlc -Q Person.java
```

```
$ javac PersonMain.java
```

```
$ jmlrac PersonMain
```

```
Exception in thread "main" org.jmlspecs.jmlrac.runtime.JMLEntryPreconditionError
```

```
: by method Person.Person regarding specifications at
```

```
File "Person.refines-java", line 52, character 20 when
```

```
  'n' is null
```

```
  at org.jmlspecs.samples.jmltutorial.Person.checkPre$$init$$Person(
```

```
    Person.refines-java:1060)
```

```
  at org.jmlspecs.samples.jmltutorial.Person.<init>(Person.refines-java:51)
```

```
  at org.jmlspecs.samples.jmltutorial.PersonMain.main(PersonMain.java:27)
```

# Summary

JML is a powerful DBC tool for Java.

For details, refer to the JML web page at

[www.jmlspecs.org](http://www.jmlspecs.org)