Lecture 3: Design By Contract with JML (Java Modeling Language)

EC3307:Object & Component technology

Outline

- Design by contract (DBC)
- Java Modeling Language (JML)
- DBC with JML
- JML tools – JML compiler (jmlc)

Contracts in Real World

- Contracts specify:
  - Agreements
  - Obligations and rights
- Contracts for buying cars
  - Clients: give money; receive cars
  - Dealers: give cars; receive money

Contracts in Software

```
/** Returns a square root approximation of a non-negative number x.
  *  @param x A non-negative number.
  *  @returns A square root approximation of x.
  */
 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

Design by Contract (DBC)

```
/*@ requires x >= 0.0;
  @ ensures (Math.abs(\result * \result - x) <= 0.00001);
  @*/
 public static double sqrt(double x) { … }
```

Advantages over informal contracts?

- Unambiguous
- Machine manipulation

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;
```

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?

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.

Outline

- ✓ Design by contract (DBC)
- Java Modeling Language (JML)
- DBC with JML
- JML tools – JML compiler (jmlc)

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

Annotations

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

```
//@ …

or

/*@ …
  @ …
  @*/
```

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

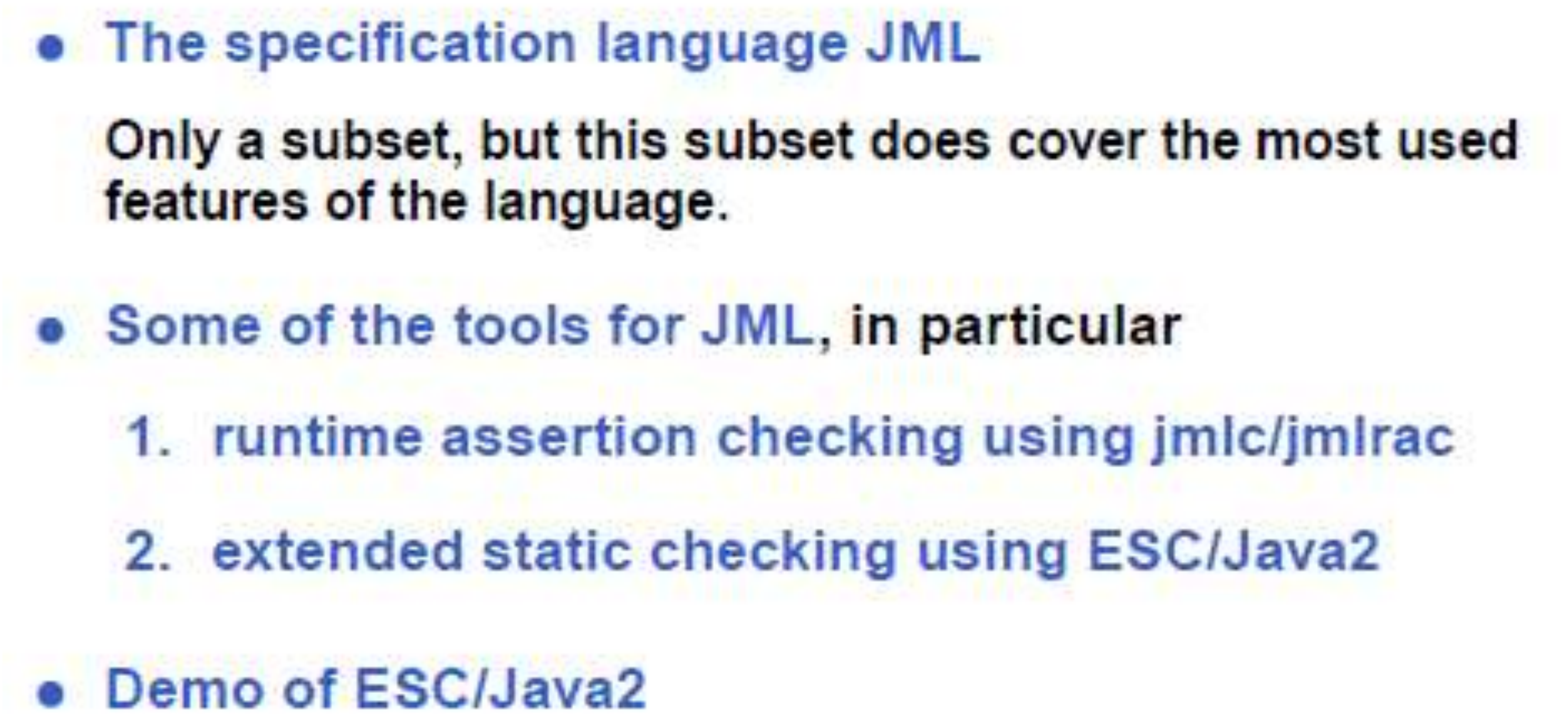
Outline

✓ Design by contract (DBC)

✓ Java Modeling Language (JML)

- DBC with JML
- JML tools – JML compiler (jmlc)

Overview



- The specification language JML

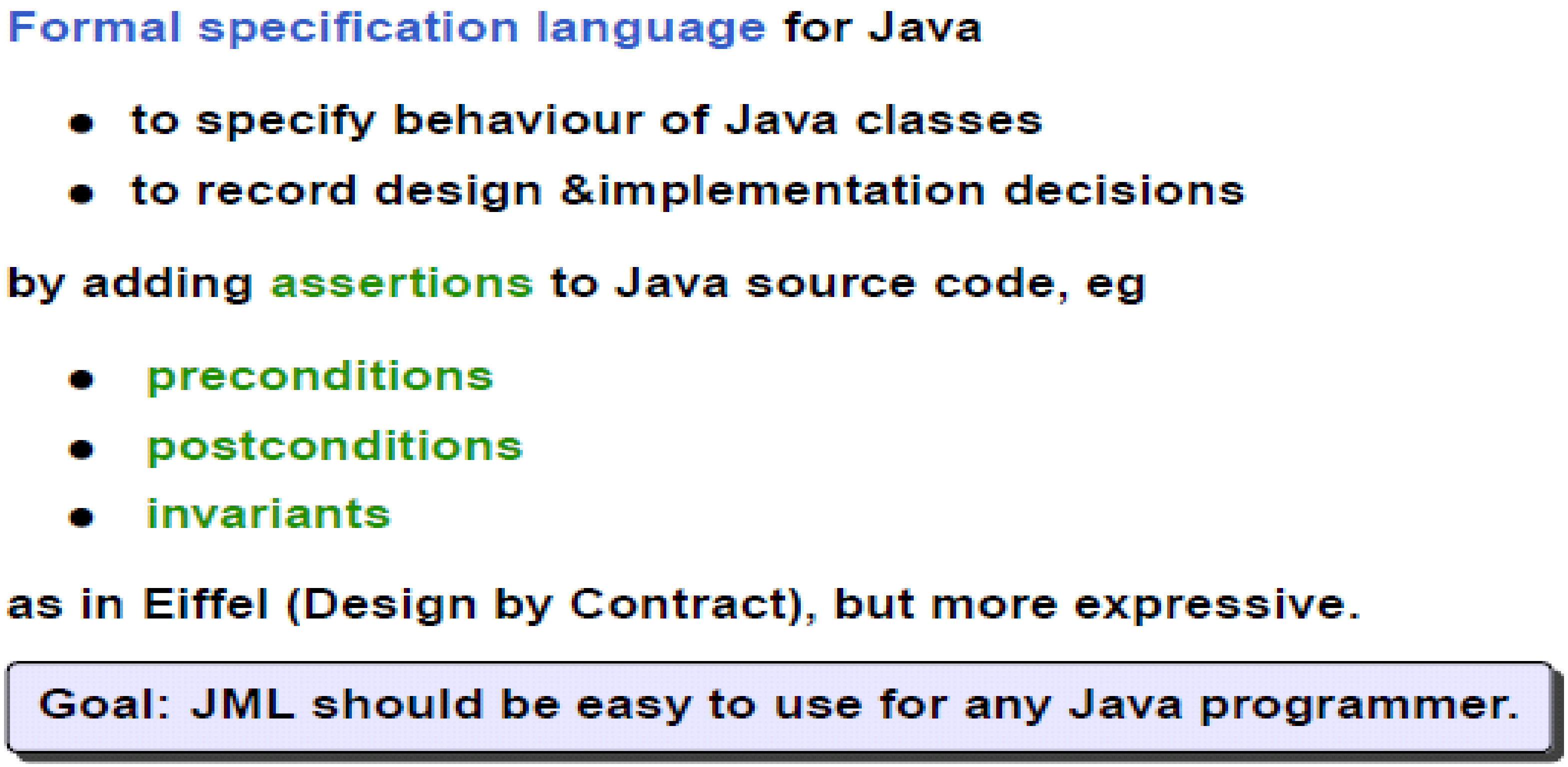
  Only a subset, but this subset does cover the most used features of the language.

- Some of the tools for JML, in particular

  1. runtime assertion checking using jmlc/jmlrac
  2. extended static checking using ESC/Java2

- Demo of ESC/Java2

JML



Formal specification language for Java

- to specify behaviour of Java classes
- to record design &implementation decisions

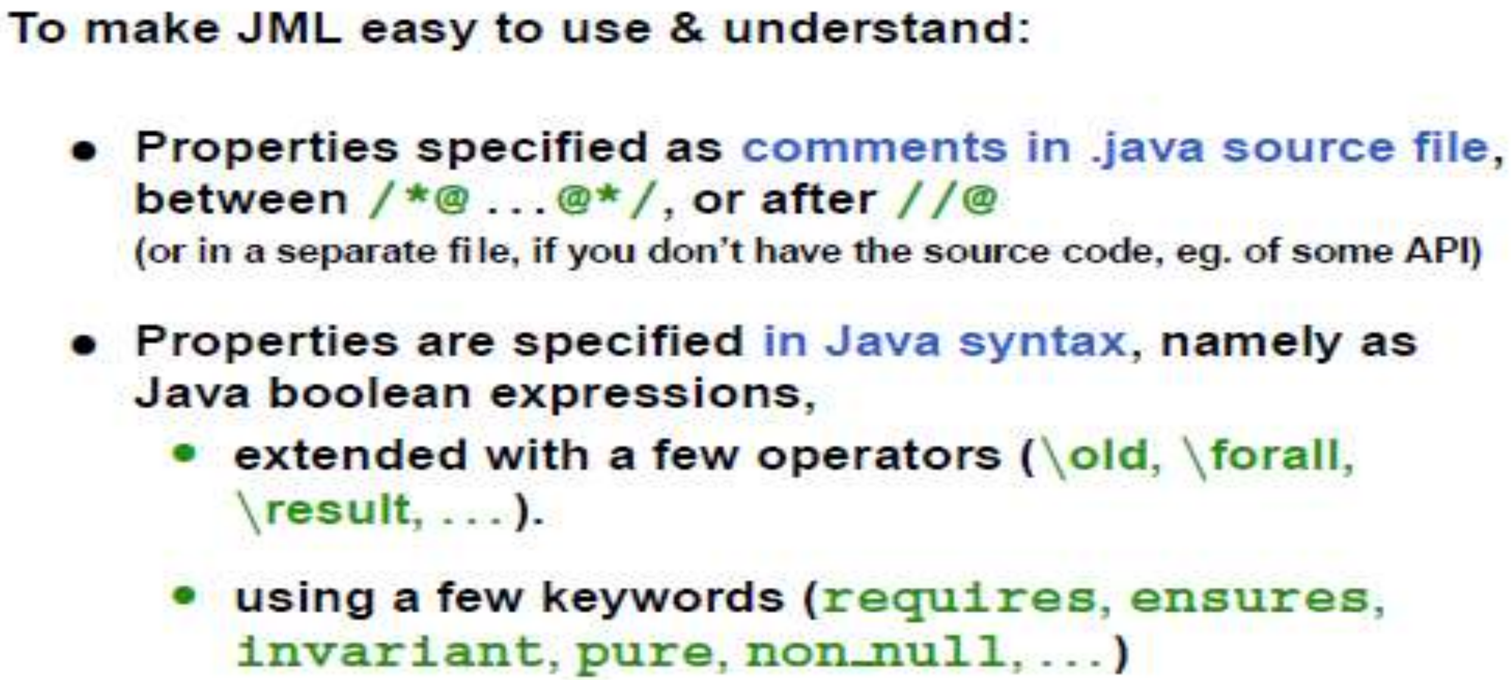
by adding assertions to Java source code, eg

- preconditions
- postconditions
- invariants

as in Eiffel (Design by Contract), but more expressive.

Goal: JML should be easy to use for any Java programmer.

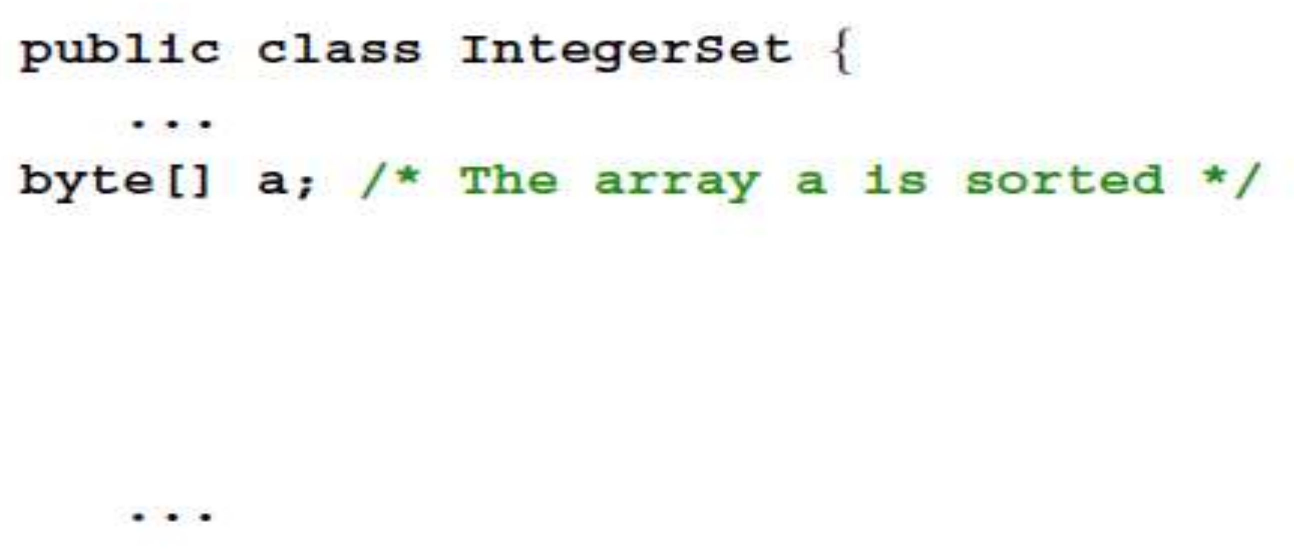
JML



To make JML easy to use & understand:

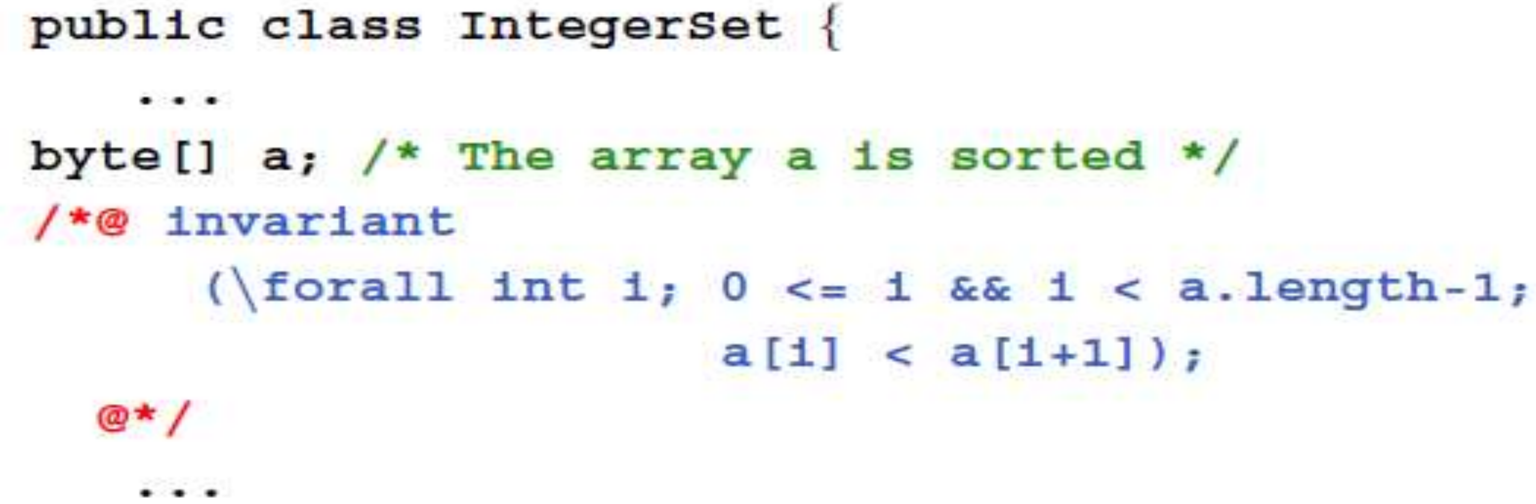
- Properties specified as comments in .java source file, between /*@ . . . @*/, or after //@
  (or in a separate file, if you don't have the source code, eg. of some API)
- Properties are specified in Java syntax, namely as Java boolean expressions,
  - extended with a few operators (\old, \forall, \result, . . .).
  - using a few keywords (requires, ensures, invariant, pure, non_null, . . .)

Example JML Specification



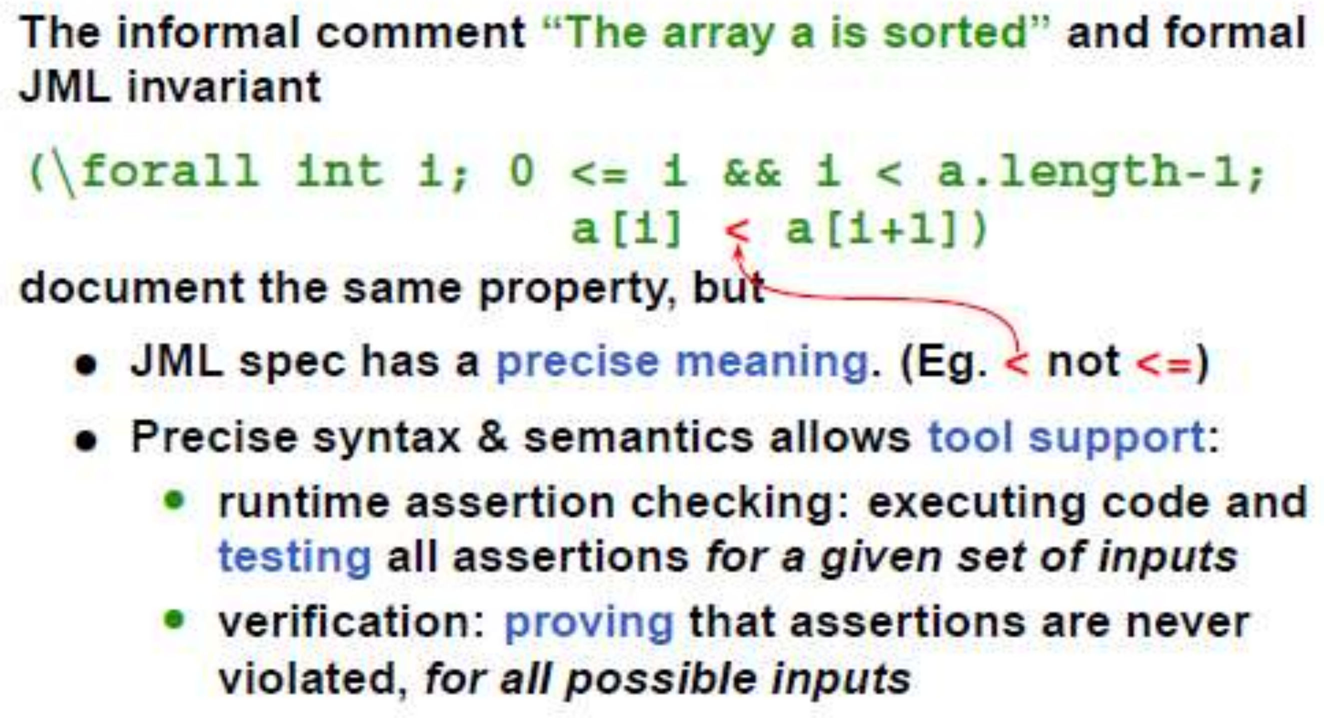
```
public class IntegerSet {
    ...
byte[] a; /* The array a is sorted */




    ...
```

Example JML Specification



```
public class IntegerSet {
    ...
byte[] a; /* The array a is sorted */
/*@ invariant
      (\forall int i; 0 <= i && i < a.length-1;
                              a[i] < a[i+1]);
  @*/
    ...
```

Informal Vs Formal



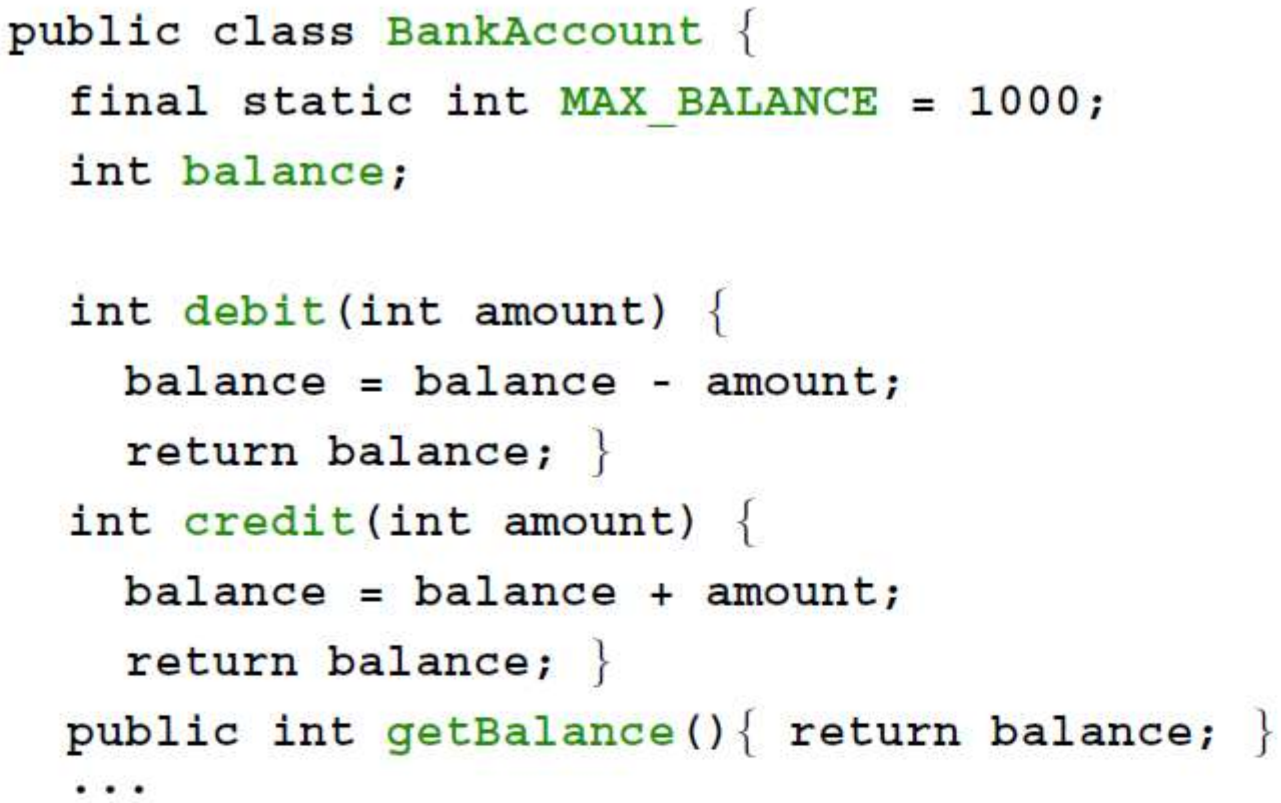
The informal comment "The array a is sorted" and formal JML invariant

```
(\forall int i; 0 <= i && i < a.length-1;
                        a[i] < a[i+1])
```

document the same property, but

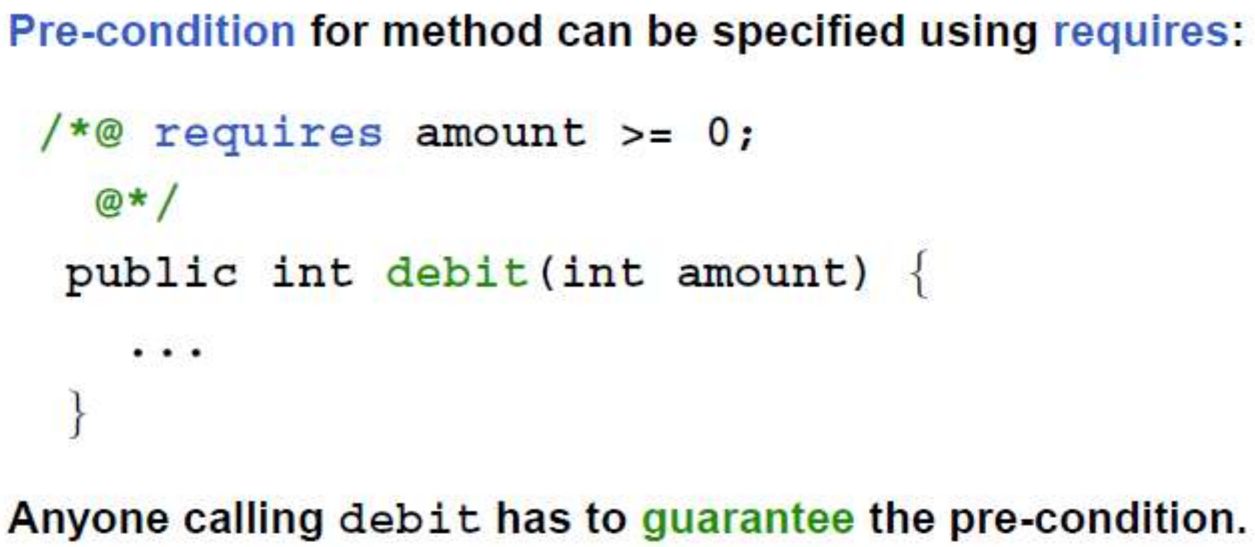
- JML spec has a precise meaning. (Eg. < not <=)
- Precise syntax & semantics allows tool support:
  - runtime assertion checking: executing code and testing all assertions *for a given set of inputs*
  - verification: proving that assertions are never violated, *for all possible inputs*

Example



```
public class BankAccount {
  final static int MAX_BALANCE = 1000;
  int balance;

  int debit(int amount) {
    balance = balance - amount;
    return balance; }
  int credit(int amount) {
    balance = balance + amount;
    return balance; }
  public int getBalance(){ return balance; }
  ...
```

requires

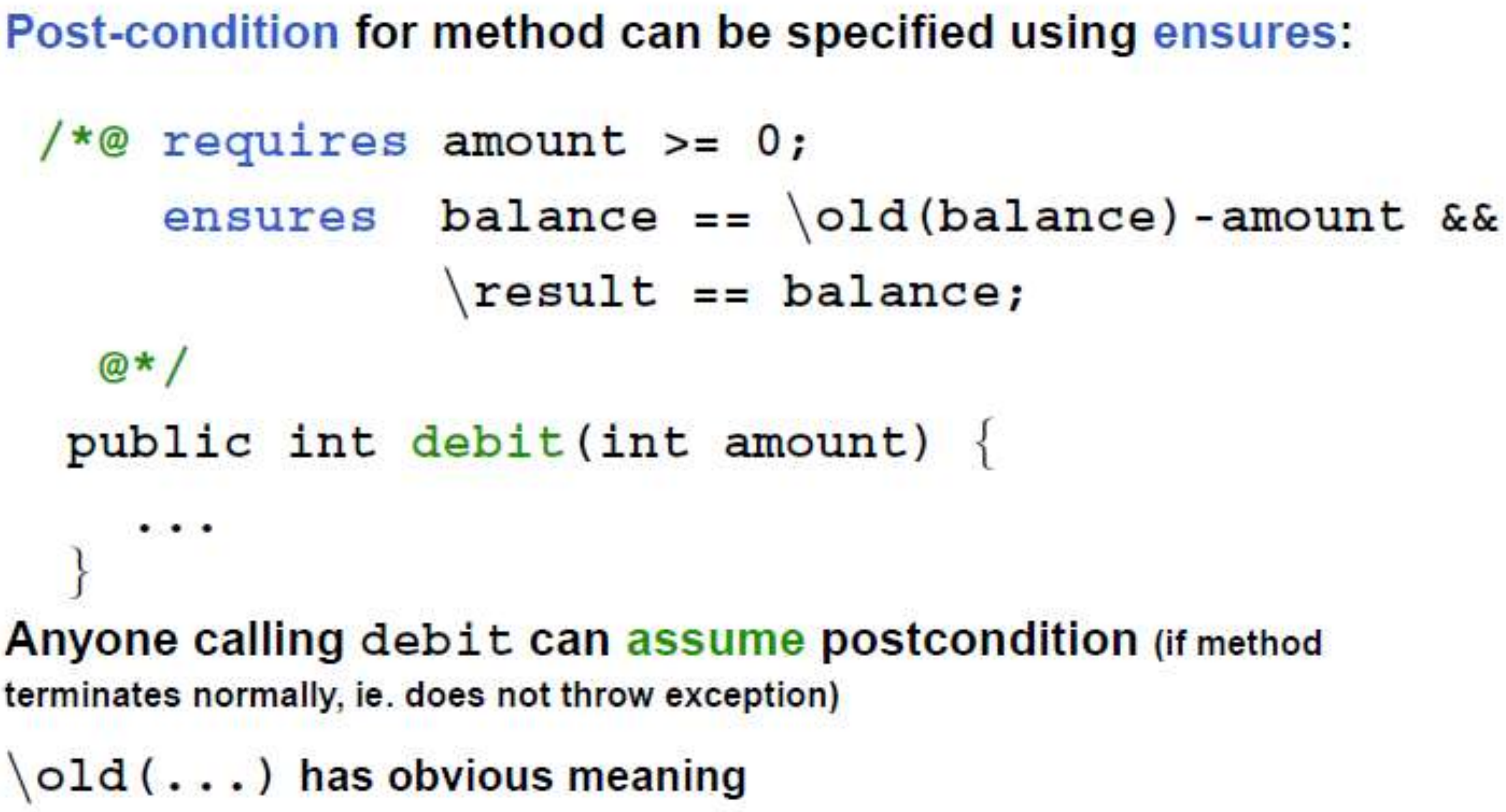


Pre-condition for method can be specified using requires:

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

Anyone calling `debit` has to guarantee the pre-condition.

ensures



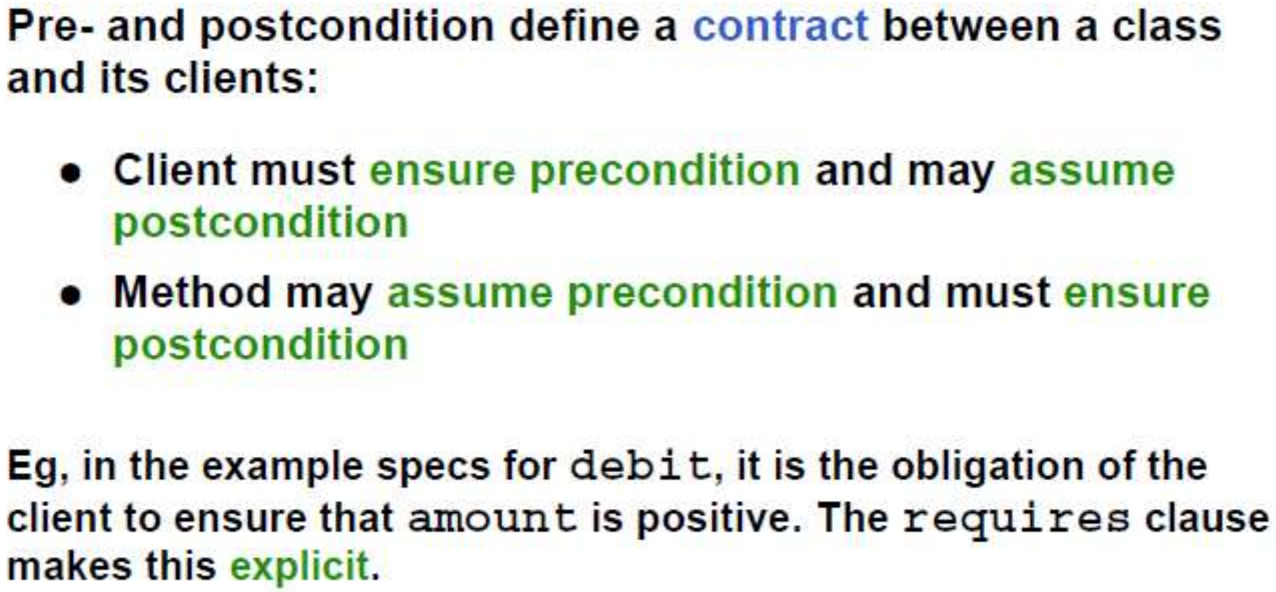
Post-condition for method can be specified using ensures:

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

Anyone calling debit can assume postcondition (if method terminates normally, ie. does not throw exception)

\old(...) has obvious meaning

Design By Contract

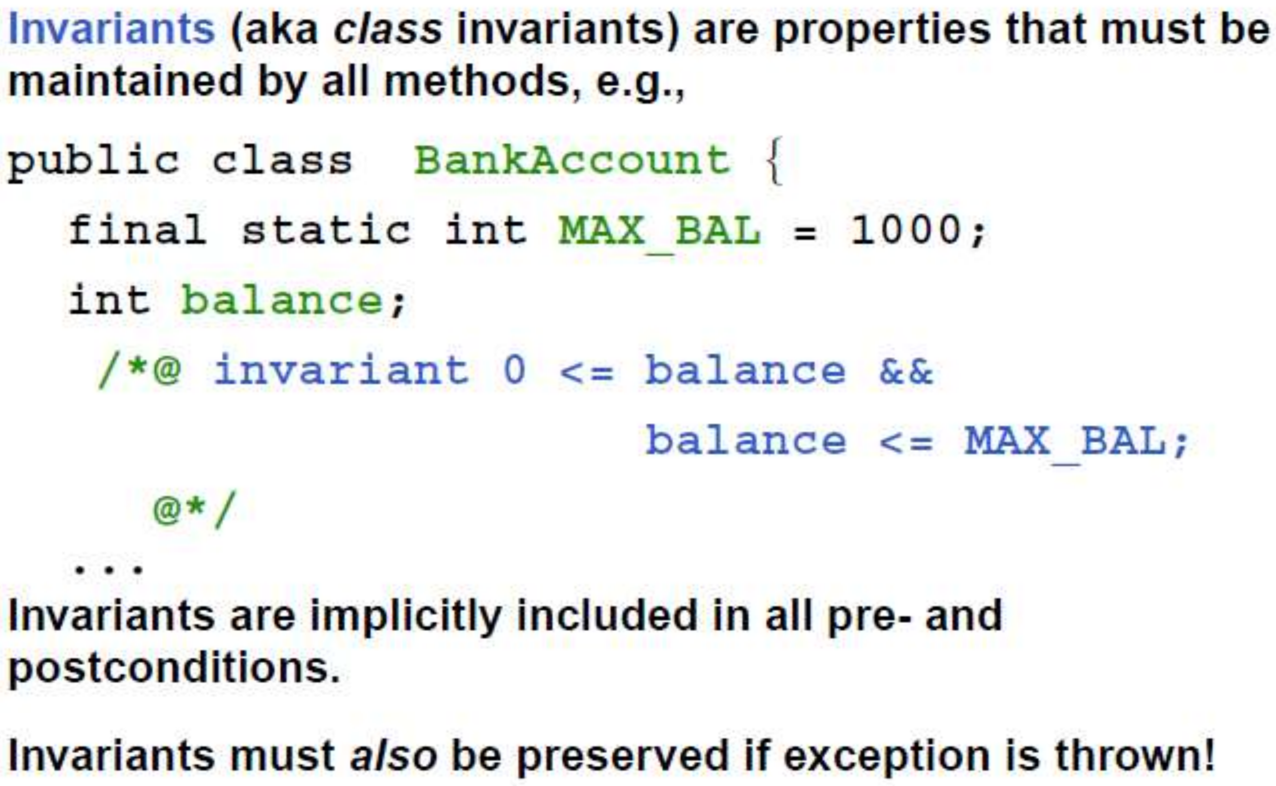


Pre- and postcondition define a contract between a class and its clients:

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

Eg, in the example specs for `debit`, it is the obligation of the client to ensure that `amount` is positive. The `requires` clause makes this explicit.

invariant



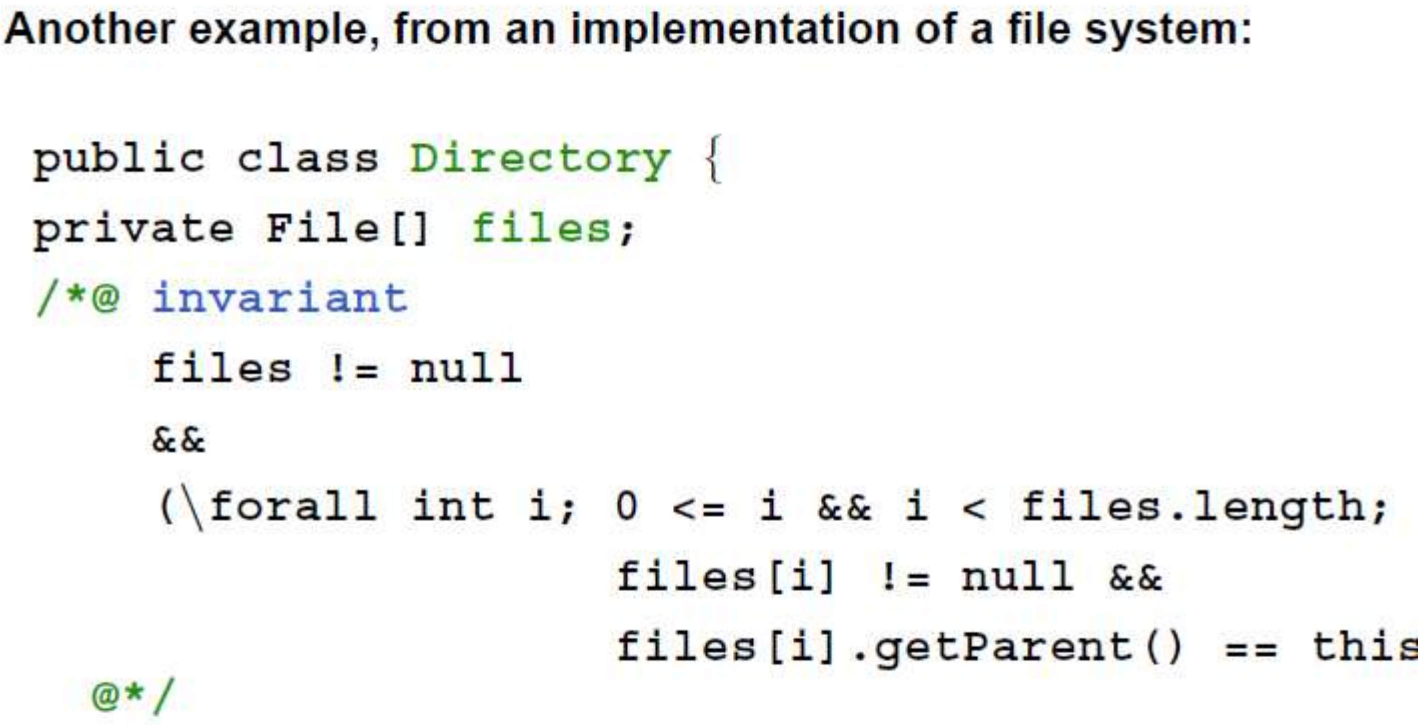
Invariants (aka *class* invariants) are properties that must be maintained by all methods, e.g.,

```
public class  BankAccount {
  final static int MAX_BAL = 1000;
  int balance;
   /*@ invariant 0 <= balance &&
                     balance <= MAX_BAL;
     @*/
  ...
```

Invariants are implicitly included in all pre- and postconditions.

Invariants must *also* be preserved if exception is thrown!

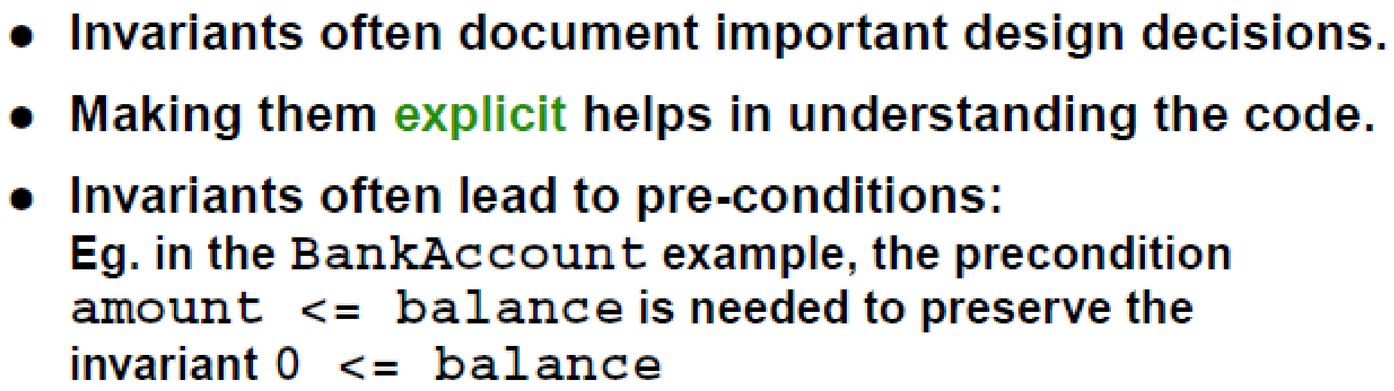
invariant



Another example, from an implementation of a file system:

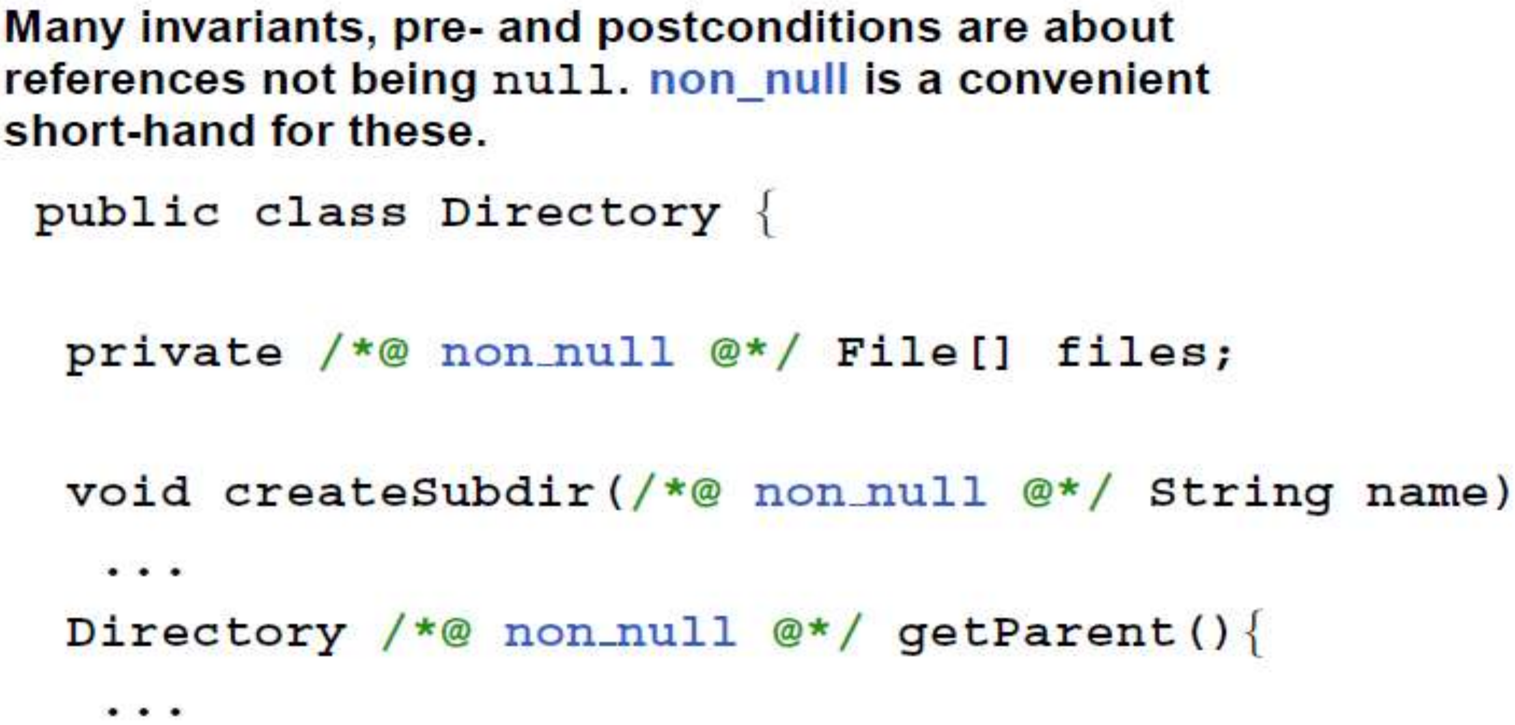
```
public class Directory {
private File[] files;
/*@ invariant
    files != null
    &&
    (\forall int i; 0 <= i && i < files.length;
                      files[i] != null &&
                      files[i].getParent() == this
  @*/
```

invariants



- Invariants often document important design decisions.
- Making them explicit helps in understanding the code.
- Invariants often lead to pre-conditions:
  Eg. in the `BankAccount` example, the precondition `amount <= balance` is needed to preserve the invariant `0 <= balance`

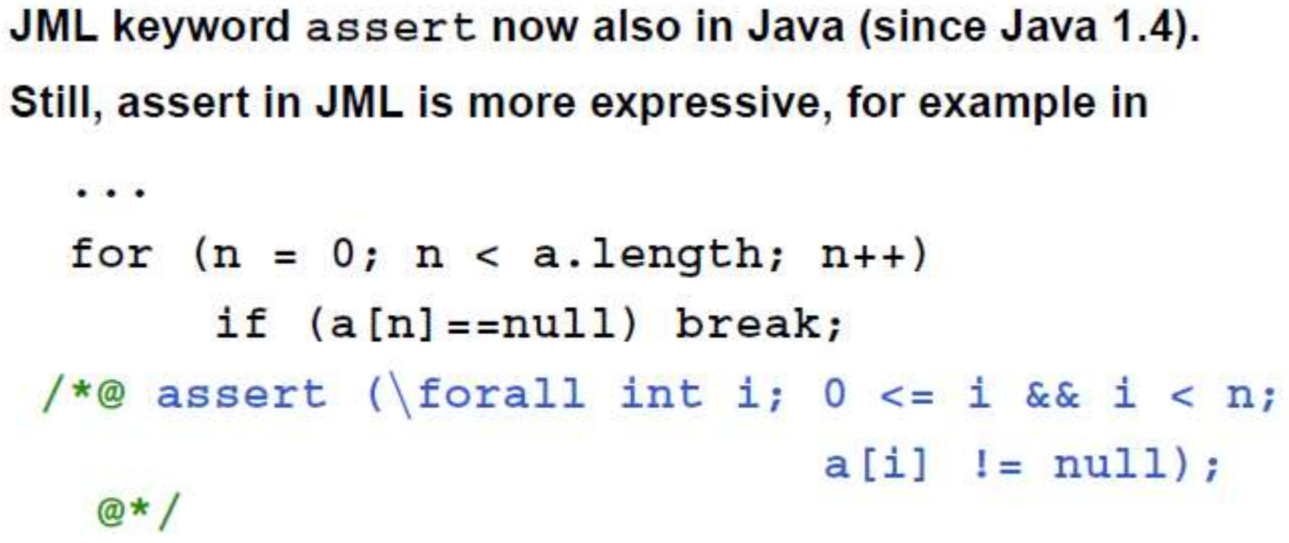
non_null



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(){
   ...
```

assert



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

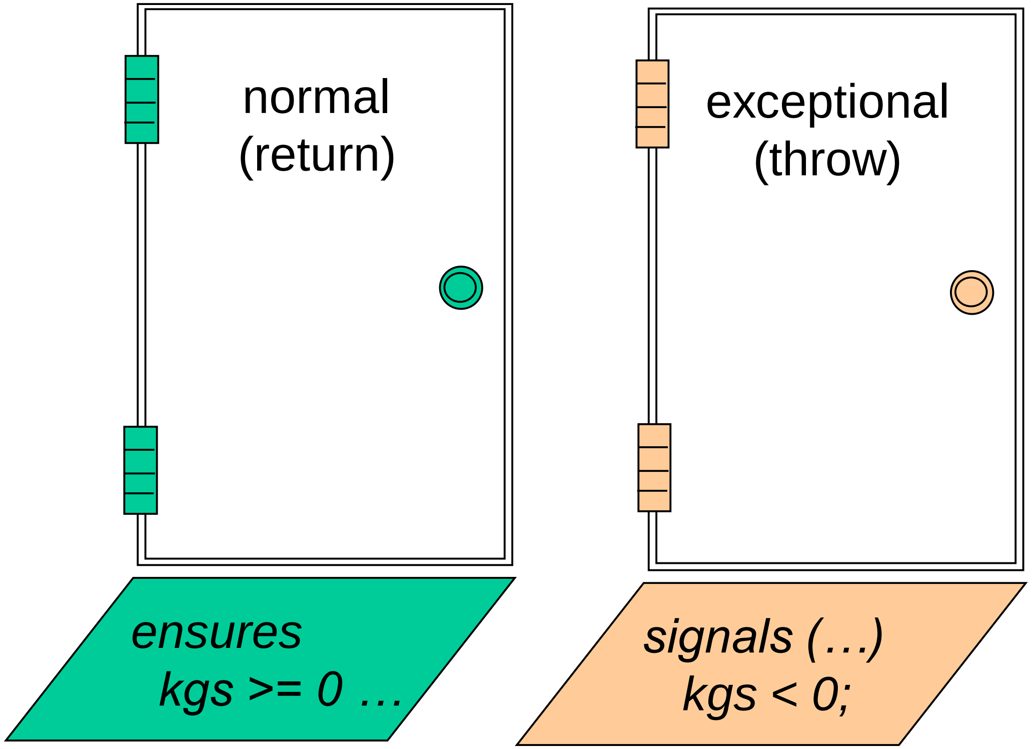
Still, assert in JML is more expressive, for example in

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

Exceptions

- Method has two ways to return
  - Normal return: the postcondition specified by *ensures* holds
  - Exceptional return: an exception is raised and the precondition specified by *signals* holds

Meaning of Postconditions



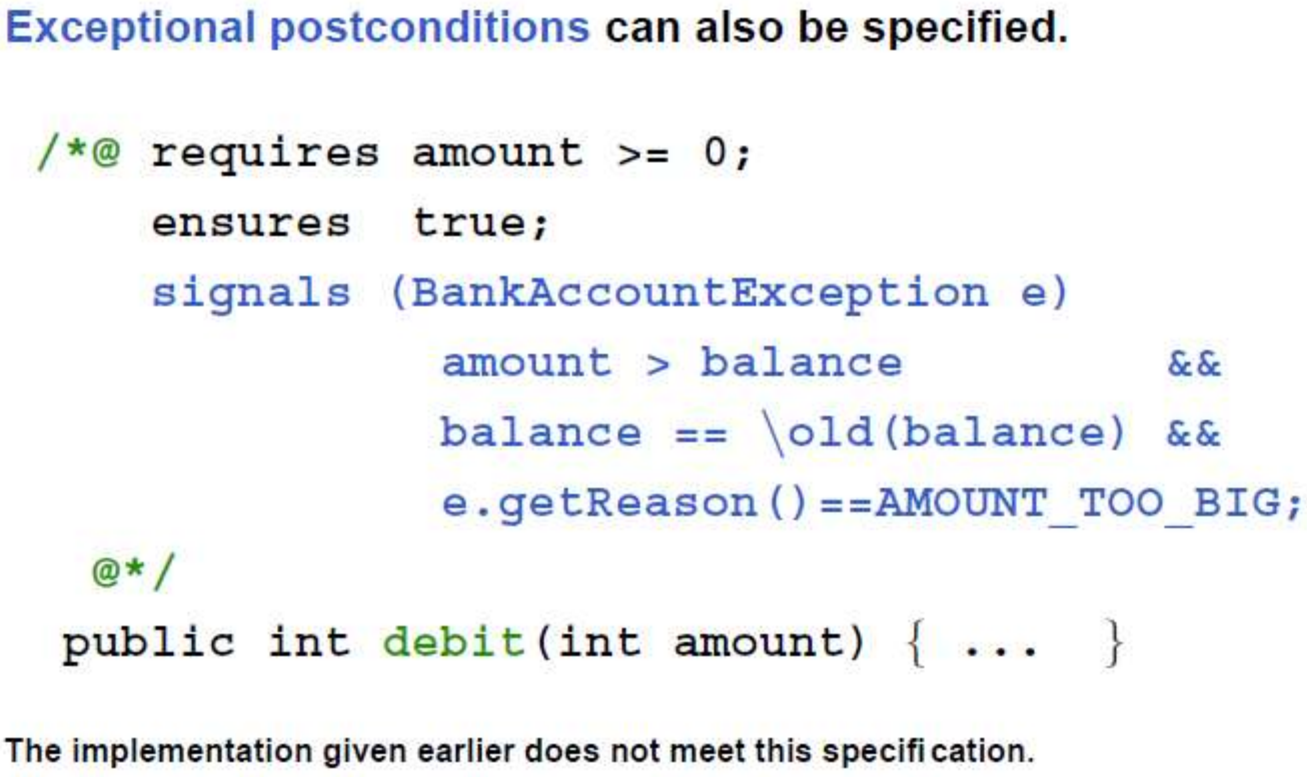
normal
(return)

exceptional
(throw)

ensures
kgs >= 0 …

signals (…)
kgs < 0;

signals

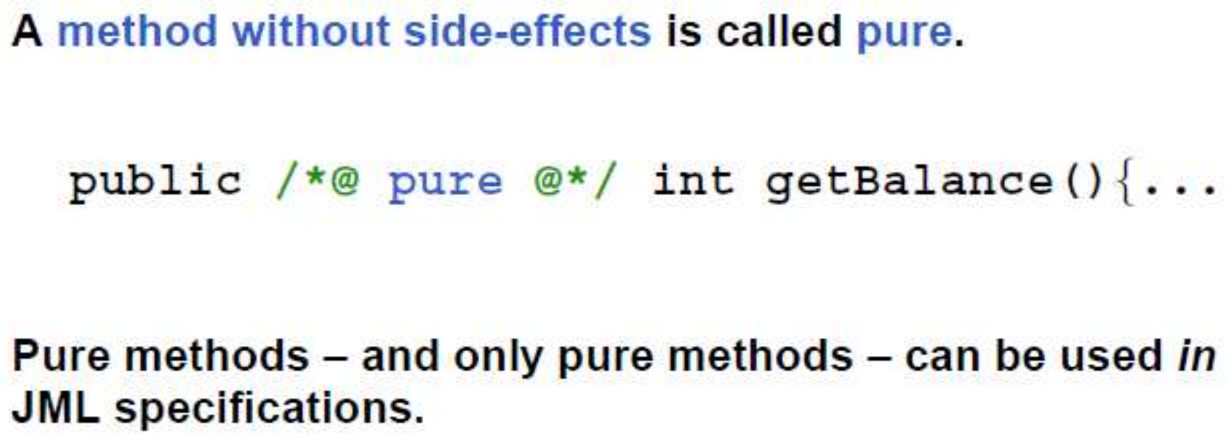


Exceptional postconditions can also be specified.

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

The implementation given earlier does not meet this specification.

pure



A method without side-effects is called pure.

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

Pure methods – and only pure methods – can be used *in* JML specifications.

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) { … }
}
```

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)
```

JML recap

- **The JML keywords discussed so far:**

  requires

  ensures

  signals

  invariant

  non null

  pure

  \old, code\forall, \exists, \result

Examples

```
/*@ requires a != null && a.length > 0;
  @ ensures (\exists int i; 0 <= i && i < a.length; \result == a[i]) &&
  @         (\forall int i; 0 <= i && i < a.length; \result >= a[i]);
  @*/
public static int mystery1(int[] a) { /* … */ }

/*@ requires a != null;
  @ ensures \result == (\sum int i; 0 <= i && i < a.length; a[i]);
  @*/
public static int mystery2(int a[]) { /* … */ }

/*@ requires a != null;
  @ ensures \result == (\num_of int i; 0 <= i && i < a.length; x > a[i]);
  @*/
public static int mystery3(int a[], int x) { /* … */ }
```

Outline

- ✓ Design by contract (DBC)
- ✓ Java Modeling Language (JML)
- ✓ DBC with JML
- JML tools – JML compiler (jmlc)

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(“Yoonsik”));
       System.out.println(new Person(null));
   }
}
```

Example (Formatted)

```
$ jmlc –Q Person.java
$ javac PersonMain.java
$ jmlrac PersonMain
Person(“Yoonsik”,0)
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: Design by Contract with JML

- This document, written by Gary T. Leavens and Yoonsik Cheon, introduces the Java Modeling Language (JML) and explains how it can be used as a Design by Contract (DBC) tool for Java.

Introduction to Design by Contract (DBC)

- DBC is a software development methodology where contracts specify the obligations and guarantees between a class and its clients. It uses:
  - Preconditions (what must be true before calling a method).
  - Postconditions (what must be true after method execution).
  - Invariants (conditions that must always hold for an object).
- JML integrates these contracts directly into Java code, allowing runtime assertion checking.

Benefits of DBC and

- Improved Documentation: JML specifications serve as precise, machine-checkable documentation.
- Error Detection and Blame Assignment: Helps pinpoint whether a bug is due to incorrect client usage or a faulty implementation.
- Increased Efficiency: Reduces unnecessary defensive programming checks, which can slow down execution.
- Modular Reasoning: Developers can understand and verify components based on contracts rather than the entire codebase.

Overview of JML

- JML extends Java’s syntax to provide formal behavioral specifications using annotations. It supports:
  - Preconditions (//@ requires condition;)
  - Postconditions (//@ ensures condition;)
  - Class Invariants (//@ invariant condition;)
  - Quantifiers (\forall, \exists for universal and existential conditions).
- JML specifications are placed in Java comments (//@ or /*@ ... @*/), making them compatible with standard Java compilers.

Examples and Features of JML

- Formal and Informal Specifications: Developers can write both informal descriptions ((* text *)) and formal contracts.
- Information Hiding: Specifications can be public or private to ensure encapsulation.
- Model Fields: Abstractions that allow specification without exposing implementation details.

JML Tools

- JML comes with a set of tools, including:
  - JML Compiler (jmlc): Compiles Java with runtime assertion checks.
  - JML Unit Testing (jmlunit): Integrates with JUnit for automatic contract-based testing.
  - Documentation Generator (jmldoc): Generates HTML documentation combining Javadoc and JML.
  - Static Checker (esc/java2): Detects errors like null pointer dereferences before execution.

Installation and Usage

- JML is freely available at www.jmlspecs.org. The document provides guidance on installing, compiling, and running JML-annotated Java programs.

Summary

- DBC is a way of recording:
  - Details of method responsibilities
  - Avoiding constantly checking arguments
  - Assigning blame across interfaces
- JML is a DBC tool for Java
- For details on JML, refer to its web page at

www.jmlspecs.org