Disclaimer. These notes are derived from notes originally developed by Joseph Kiniry, Gary Leavens, Erik Poll, David Cok, Cesare Tinelli, and Jochen Hoenicke. They are copyrighted material and may not be used in other course settings outside of New York University in their current form or modified form without the express written permission of one of the copyright holders.
Exploiting Design Information

- Alloy provides a means for expressing properties of designs
  - Early design refinement saves time
  - Ultimately, we want this effort to impact the quality of implementations

- How can we transfer design information to the code?
  - State information (multiplicities, invariants, ...)
  - Operations information (pre, post, frame conditions, ...)
Design by Contract

- A method that emphasizes the precise description of **interface semantics**
  - not just syntax, e.g., signatures (names, types, visibility modifiers)
  - but run-time behavior, e.g., effects of a method call
- Supported by tools that
  - allow semantic properties of the design to be propagated to the code
  - support various forms of validation of those properties, e.g., run-time and static checking
History

• Term “Design by Contract” was first coined by Bertrand Meyer in the context of the Eiffel language

• Basic ideas and techniques go back to pioneering work of
  – Alan Turing (1949)
  – Robert Floyd (1967)
  – Tony Hoare (1969)
  – Edsger Dijkstra (1975)
Basic Idea

• Software is viewed as a system of communicating components (objects)
  – all interaction is governed by contracts
  – contracts are precise specifications of mutual obligation between components
Contracts

- Two parties are involved in a contract
  - The supplier performs a task
  - The client requests that the task be performed
- Each party
  - has obligations
  - receives some benefits
- Contracts specify those obligations and benefits
- Contracts are bi-directional
  - both parties are obligated by them
Contract Example: Air Travel

Client (Traveler)
• Obligation
  – check in 30 minutes before boarding
  – <3 small carry-ons
  – pay for ticket
• Benefit
  – reach destination

Supplier (Airline)
• Obligation
  – take traveler to destination
• Benefit
  – don’t need to wait for late travelers
  – don’t need to store arbitrary amounts of luggage
  – money
Contract Example: Air Travel

Client (Traveler)
- **Obligation**
  - check in 30 minutes before boarding
  - <3 small carry-ons
  - pay for ticket
- **Benefit**
  - reach destination

Supplier (Airline)
- **Obligation**
  - take traveler to destination
- **Benefit**
  - don’t need to wait for late travelers
  - don’t need to store arbitrary amounts of luggage
  - money
Contracts

• Specify **what** should be done **not how** it should be done
  – they are implementation independent

• This same idea can be applied to software using the building blocks we have already learned in Alloy
  – pre conditions
  – post conditions
  – frame conditions
  – invariants
Taking a Flight (Java Syntax)

class Flight {
    /*@ requires time < this.takeoff - 30 &&
       @ l.number < 3 &&
       @ p in this.ticketed; 
       @ ensures \result = this.destination;
       @*/
    Destination takeFlight(Person p, Luggage l)
    {...
    }
}
Specification or Implementation Language

• Why not both?
• Refinement methodology
  – rather than develop signatures alone
  – develop contract specification
  – analyze client-supplier consistency
  – fill in implementation details
  – check that code satisfies contract
• Natural progression from design to code
Executable Specifications

• Specification language is a subset of the implementation language
  – contracts are written in the programming language itself
  – and translated into executable code by the compiler
  – enables easy run-time checking of contracts
Java Example: Stack Data Structure

class Mystack {
    private Object[] elems;
    private int top, size;
    public Mystack (int s) {
        ...
    }
    public void push (Object obj) {
        ...
    }
    public Object pop() {
        ...
    }
    public boolean isEmpty() {
        ...
    }
    public boolean isFull() {
        ...
    }
}
Java Example: Stack Data Structure

```java
/*@ invariant top >= -1 &&
    top < size &&
    size = elems.length();
 */

class Mystack {
    private Object[] elems;
    private int top, size;
    ...
}
```
class Mystack {
    private Object[] elems;
    private int top, size;
    ...
    /**< requires !isFull();
     @ ensures top == \old(top) + 1 &&
     @ elem[top] == obj;
     @*/
    public void push (Object obj) { ... } 
    ...
    public boolean isFull() { ... } 
}
class Mystack {
    private Object[] elems;
    private int top, size;
    ...
    /*@ requires !isEmpty();
    @ ensures top == \old(top) - 1 &&
    @    \result == elem[\old(top)];
    @*/
    public Object pop() { ... }
    ...
    public boolean isEmpty() { ... }
}
Java Example: Stack Data Structure

class Mystack {
    private Object[] elems;
    private int top, size;
    ...
    /*@ ensures \result \iff top = -1; */
    public boolean isEmpty() { ... }
}
Source Specifications

• Pre/post conditions
  – (Side-effect free) Boolean expressions in the host language

• What about all of the expressive power we have in, e.g., Alloy?
  – Balance expressive power against checkability
  – Balance abstractness against language mapping

• No one right choice
  – Different tools take different approaches
Important Issues

• Contract enforcement code is executed
  – It should be side-effect free
  – If not, then contracts change behavior!
• Frame conditions
  – Explicitly mention what can change
  – Default: anything can change
• Failed contract conditions
  – Most approaches will abort the execution
  – How can we continue?
Contract Inheritance

• Inheritance in most OO languages
  – Sub-type can be used in place of super-type
  – Sub-type provides at least the capability of super-type

• Sub-types **weaken** the pre-condition
  – Require no more than the super-type
  – Implicit **or** of inherited pre-conditions

• Sub-types **strengthen** the post-condition
  – Guarantee at least as much as the super-type
  – Implicit **and** of inherited post-conditions

• Invariants are treated the same as post-conditions
Languages with DbC Support

- Eiffel
- SPARK (Ada)
- Spec# (C#)
- **Java**
  - Java Modeling Language (JML)
  - iContract, JContract, Jass, Jahob, ...
- .NET languages: Code Contracts
- C/C++: VCC, Frama-C, ...
- Research languages: **Daphne**, Chalice, Hob, ...
- ...
Java Modeling Language (JML)

JML is a behavioral interface specification language (BISL) for Java.


• Combines ideas from two approaches:
  – Eiffel with its built-in language for Design by Contract
  – Larch/C++ a BISL for C++
The Roots of JML

• Ideas from Eiffel:
  – Executable pre and post-condition for runtime assertion checking
  – Uses Java syntax (with a few extensions).
  – Operator `\old` to refer to the pre-state in the post-condition.

• Ideas from Larch:
  – Describe the state transformation behavior of a method
  – Model Abstract Data Types (ADT)
Java Modeling Language (JML)

- Homepage: http://www.jmlspecs.org/
- Release can be downloaded from http://sourceforge.net/projects/jmlspecs/files
- Includes many useful tools for testing and analysis of contracts
  - JML compiler
  - JML runtime assertion checker, ...
- Many additional third party tools available
JML: Tool Support

- Run-time checking and dynamic analysis:
  - JML tools
  - AJML
  - Daikon

- Automated test case generation:
  - JML tools
  - Korat,
  - Sireum/Kiasan
  - KeY/TestGen

- Static checking and static analysis:
  - ESC/Java 2
  - JForge

- Formal verification:
  - JACK
  - KeY

- Documentation generation: jmldoc (JML tools)
JML Example: Factorial

Is this method correct?

```java
public static int factorial(int n) {
    int result = n;
    while (--n > 0)
        result *= n;
    return result;
}
```

We need a specification!
JML Syntax: Method Specifications

In JML a method contract precedes the method in special comments /*@ ... @*/.

• **requires** formula:
  – The specification only applies if formula holds when method called.
  – Otherwise behavior of method is undefined.

• **ensures** formula:
  – If the method exits normally, formula has to hold.
A JML formula is a Java Boolean expression. The following list shows some operators of JML that do not exist in Java:

- **\old(expression):**
  - the value of expression before the method was called (used in ensures clauses)

- **\result:**
  - the return value (used in ensures clauses).

- **F ==> G:**
  - states that F implies G. This is an abbreviation for !F || G.

- **\forall Type t; condition; formula:**
  - states that formula holds for all t of type Type that satisfy condition.
/*@
requires n >= 0;
@ ensures \result == n! ;
@*/
public static int factorial(int n) {
    int result = n;
    while (--n > 0)
        result *= n;
    return result;
}

Is this method correct?

But factorial ! is not an inbuilt operator.
Solutions (1): Weakening the specification

```java
/*@ requires n >= 0;
   @ ensures \result >= 1;
   @*/
public static int factorial(int n) {
    int result = n;
    while (!(--n > 0))
        result *= n;
    return result;
}
```

+ Simple Specification
+ Catches the error
- Cannot find all potential errors
- Gives no hint, what the function computes
JML Example: Factorial

Solutions (2): Using pure Java methods

```java
/*@ requires n >= 0;
  @ ensures (n == 0 ==> \result == 1) &&
  @ (n > 0 ==> \result == n*fact(n-1)); */

public static @pure int fact(int n) {
  return n <= 0 ? 1 : n*fact(n-1);
}
```

Pure methods must not have side-effects and must always terminate. They can be used in specifications:

```java
/*@ requires n >= 0;
  @ ensures \result == fact(n); */

public static int factorial(int n) {
  int result = 1;
  while (n > 0) result *= n--;
  return result;
}
```
Partial vs. Full Specifications

Giving a full specification is not always practical.

• Code is repeated in the specification.
• Errors in the code may also be in the specification.
Semantics of Java Programs

The Java Language Specification (JLS) 3rd edition gives semantics to Java programs

• The document has 684 pages.
• 118 pages to define semantics of expression.
• 42 pages to define semantics of method invocation.
• Semantics is only defined by prosa text.
Example: What does this program print?

class A {
    public static int x = B.x + 1;
}

class B {
    public static int x = A.x + 1;
}

class C {
    public static void main(String[] p) {
        System.err.println("A: " + A.x + ", B: " + B.x);
    }
}
Example: What does this program print?

JLS, chapter 12.4.1 “When Initialization Occurs”: A class T will be initialized immediately before the first occurrence of any one of the following:

• T is a class and an instance of T is created.
• T is a class and a static method declared by T is invoked.
• A static field declared by T is assigned.
• A static field declared by T is used and the field is not a constant variable.
• T is a top-level class, and an assert statement lexically nested within T is executed.
Example: What does this program print?

JLS, chapter 12.4.2 “Detailed Initialization Procedure”:
The procedure for initializing a class or interface is then as follows:

1. Synchronize on the Class object that represents the class or interface to be initialized. This involves waiting until the current thread can obtain the lock for that object.

2. . .

3. If initialization is in progress for the class or interface by the current thread, then this must be a recursive request for initialization. Release the lock on the Class object and complete normally.

4.–8. . .

9. Next, execute either the class variable initializers and static initializers of the class, or the field initializers of the interface, in textual order, as though they were a single block, except that final class variables and fields of interfaces whose values are compile-time constants are initialized first.

10.– . .
Example: What does this program print?

class A {
    public static int x = B.x + 1;
}

class B {
    public static int x = A.x + 1;
}

class C {
    public static void main(String[] p) {
        System.err.println("A: "+A.x + ", B: "+B.x);
    }
}
Example: What does this program print?

If we run class C:
1) main-method of class C first accesses A.x.
2) Class A is initialized. The lock for A is taken.
3) Static initializer of A runs and accesses B.x.
4) Class B is initialized. The lock for B is taken.
5) Static initializer of B runs and accesses A.x.
6) Class A is still locked by current thread (recursive initialization). Therefore, initialization returns immediately.
7) The value of A.x is still 0 (section 12.3.2 and 4.12.5), so B.x is set to 1.
8) Initialization of B finishes.
9) The value of A.x is now set to 2.
10) The program prints “A: 2, B: 1”.
Further Reading Material

• Gary T. Leavens, Yoonsik Cheon. Design by Contract with JML
• G. Leavens et al.. JML Reference Manual (DRAFT), July 2011