2IP15 Programming Methods

From Small to Large Programs

Tom Verhoeff
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

www.win.tue.nl/~wstomv/edu/2ip15

Overview

- Java **`class`** mechanism (basics)
- Data Types
- Data Abstraction

A program that involves relations

```
1     boolean[][] relation = new boolean[75][75];
2         ...
3         ...
4     relation[1][42] = true;
5         ...
6         ...
7     if (relation[i][j]) {
8         relation[j][i] = false;
9     }
10        ...
11        ...
12    boolean otherRelation[][] = new boolean[10000][10000];
13        ...
14        ...
```

Java `class` mechanism

The **`class`** concept plays a central role in the Java programming language.

It is very general and "powerful", and can easily be abused.

Initially, you used a **`class`** to collect *static variables and methods*.
It can also involve *instance variables and methods*, and
`extends` (inheritance), **`abstract`**, **`interface`**, **`implements`**, generics

In this lecture, we will learn to use it to define some common data types:

- Enumeration type
- Record type
- Abstract Data Type (ADT)

Modularization: Top-down and Bottom-up Design

Design is an activity, to find and structure a solution.

- Top-down design
  - Given one (big) problem, split it into (smaller) subproblems
  - Continue until reaching trivially solvable problems
- Bottom-up design
  - Given many small solutions, combine them into bigger solutions
  - Continue until reaching a solution of the main problem

Solutions could be: expressions, statements, data definitions, ...
methods, classes, interfaces, packages, programs, ...

In practice: Yo-yo design (top-down and bottom-up combined)

Modularization for Actions

Procedural abstraction enables one to abstract from details within

- an expression
- a (possibly *compound*) statement

In Java, this is done by means of *methods* that

- operate on parameter objects, and/or
- return a primitive value, or (a reference to) an object, or **`void`**

A *method call* acts as an expression or statement.

(Limits to) Procedural Abstraction in Java

If expressions like `p*p*p*p` and `(x+1)*(x+1)*(x+1)*(x+1)/2` occur in various places, then this invites the procedural abstraction:

```
1     /** @return a^4 */
2     int pow4(int a) {
3         int h = a * a;
4         return h * h;
5     }
6
7     .... pow4(p) ... pow4(x+1)/2 ....
```

(N.B. This implementation is also more efficient, in multiplications. How to generalize that for raising to the power $n \geq 0$ efficiently?)

However, in Java it is impossible to define a procedural abstraction `inc(v)` for `v = v + 1`, where `v` is a parameter of type "**int** variable".

Modularization for Data: Data Abstraction (Bottom-up)

Combine variables of primitive types that frequently occur together into a single abstraction, instead of handling them separately.

- The numerator and denominator of a *fraction*
- The coefficients of a *polynomial*
- A pair (or triple) of coordinates of a *point* in the plane (in space)
- The given name, family name, and email address of a *person*

A data abstraction needs to be given a single name: variable vs type

In Java, this can be accomplished by `String`, array, and **`class`**, ...
However, only a **`class`** introduces a name for a new type.

Class as Bundle of Variables: Example

```
1  /** Data Type for fractions (simplistic). */
2  public class Fraction {
3
4      /** The numerator. */
5      long numerator;
6
7      /** The denominator. denominator > 0 */
8      long denominator;
9
10     // Operations on objects of this type.
11     ...
12 }
13
14 ... Fraction q = new Fraction(2, 3);
15 ... Fraction r = q.add(q);
```

Modularization for Data: Data Abstraction (Top-down)

What data items to **distinguish** and put in separate 'modules'?

Which concepts play a role?
Nouns in requirements serve as a hint for data abstractions.
Verbs hint at operations on the data.

Hierarchic decomposition ...
Possibly recursive, i.e. in terms of itself ...

E.g.: a `Rectangle` involves `Points`, each `Point` has `Coordinates`.

A `BinaryTree` is either empty, or has a root and two `BinaryTree`-s.

Type

A (data) type is a *set of values* and *accompanying operations*.

Primitive data types in Java: **`int`**, **`double`**, **`char`**, **`boolean`**, ...
Values in subset of $\mathbb{Z}$, subset of $\mathbb{R}$, {`...,'a',...`}, {`false, true`}

Type usage:

```
1     T v; // declares variable v of type T
2          // during execution, v has one value of type T
3
4     ... v ... // usage of v in operations; depends on T
5     ... ++ v ... // for an int
6     ... v = Math.sqrt(v); // for a double
7     ... ! v && w ... // for a boolean
8     ... v[2] ... // for an array
```

Cf. *Type Theory* and *Type System*

Data Type Definitions in Java: Enumerations

```
public class PrimaryColor {
    final static int RED = 0;
    final static int GREEN = 1;
    final static int BLUE = 2;
}

int v = PrimaryColor.RED; // a variable v having a primary color as value
```

or (why useful?):

```
    final static int RED = 1;
    final static int GREEN = 2;
    final static int BLUE = 4;
```

Better (since Java 5.0):

```
public enum PrimaryColor { RED, GREEN, BLUE }
PrimaryColor v = PrimaryColor.RED; // ...
```

Java Enumeration Types

```
public enum T { NAME1 , NAME2 , ... , NAMEn }
```

Set of values: the set of listed *names*

Operations (involving `v`, `w` of type `T`):

- constants (by their name): `NAME1`, `...`
- iteration: `for (T v : T.values()) { ... v ... }`
- selection: `switch (v) { case NAME1: ...; break; ... }`
- conversion to/from string: `v.name()`, `v.toString()`, `T.valueOf(s)`
- ranking: `v.ordinal()` (ranks start at 0), `v.compareTo(w)`

Data Type Definitions in Java: 'Records'

```
/** Record type for
  * the coordinates of a field on a chess board.
  */
public class ChessBoardField {
    public char line; // 'a' .. 'h'
    public int row; // 1 .. 8
}

ChessBoardField f; // value: the coordinates of a chess field

... f.line ... f.row ...  // using variable f
```

(Also see `DivisionResult` in `Candy` example)

Java 'Record' Types via Classes

```
public class T {
    public Type1 fieldName1 ;
    ... ;
    public TypeN fieldNameN ;
}
```

Set of values: *Labeled product* of the value sets of the types, mapping `fieldName1` to `Type1`, etc.

Operations:

- field access (projection on field name): `v.fieldName1`, `...`

Abstract Data Type (ADT): Definition of Concept

An Abstract Data Type is a type whose *specification* and *usage* abstracts from (i.e. does not depend on) *implementation* details.

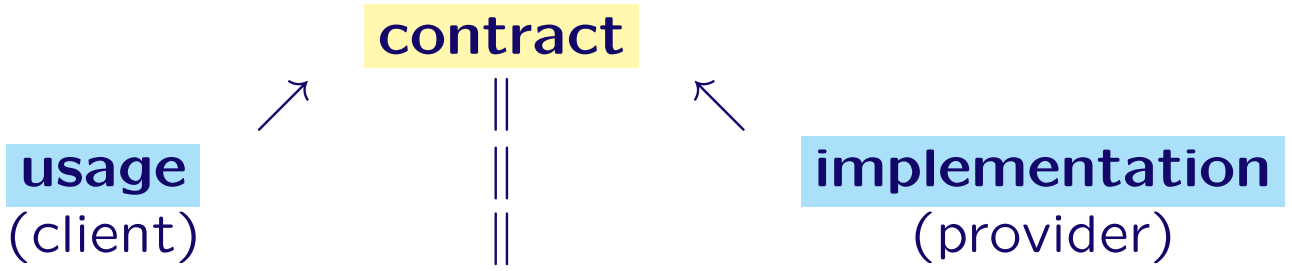
The implementation deals with the choice of data representation and algorithms for operations on that representation.

The implementation of an ADT can be changed, without affecting the *usage occurrences*, provided it adheres to the ADT's *specification*.

This is called implementation hiding, also known as encapsulation.

An ADT can serve as a module of a program.

Abstract Data Type: Specification Plays Central Role



- Relate usage to contract and

  relate implementation to contract.

- *Never* relate usage and implementation directly.

  That way, 'divide' would fail,
  leading to complexity and errors, en hence not to 'conquer'.

Example: Java Collections Framework (JCF)

- Abstract Data Types for various containers and mappings
- Concepts of a `Collection`, `Map`
- Concepts of a `List`, `Set`, `Queue`
- Implementations: `ArrayList`, `HashSet`, `HashMap`
- Involves **`interface`**s and **`class`**es

Abstract Data Type: Specification of Syntax

- Type name
- Operations (methods) with names and typed parameters:
  - Constructor, destructor (memory management)

    N.B. In Java, object destruction is implicit and automatic via garbage collection
  - Queries (state inspection), with return type
  - Commands (state change), **`void`**

Abstract Data Type: Specification of Semantics = Contract

- Set of (abstract) values
  - given *explicitly* (through *model variables*; we do this), or
  - given *implicitly* (through postulated properties of operations)
- Contracts for individual operations
  - **precondition**, **modifies** clause, **postcondition**/**returns**, thrown **exceptions**
- Public invariants: guaranteed relationships between model variables and basic queries

In Java, specification elements are stated in Javadoc comments

Example of ADT Syntax: `HashSet<E>`

- Model: (mathematical) set over type `E` (no order, no duplicates)
- Client does not (need to) know how this is stored/implemented
- Constructor `HashSet<E>()`: returns empty set over type `E`
- Query **`int`** `size()`: number of elements in **`this`**
- Query **`boolean`** `contains(Object o)`: whether `o` $\in$ **`this`**
- Command `add(E e)`: adds `e` to **`this`**
- Command `remove(Object o)`: removes `o` from **`this`**, if present

Abstract Data Type: Implementation in Java (in brief))

- Provide a data representation, a representation invariant, and an abstraction function.
- For each operation, provide a method implementation adhering to the contract.

Abstract Data Type: Implementation

Data representation is defined in terms of instance variables that represent intended abstract values of the ADT.

```
class Fraction { private int numerator, denominator; ... }
```

Representation invariant (short: *rep invariant*) is condition to be satisfied by the instance variables, in order to make sense as a representation of an abstract value. Can be implemented in method **boolean** `isRepOk()`, to support unit testing of the ADT.

```
// rep inv I0: 0 < denominator
```

Abstraction function maps each *representation that satisfies the rep invariant* to the represented abstract value.
Can be implemented (in a way) in method `String toString()`.

```
// AF(q) = q.numerator / q.denominator
```

ADT Specification (...) & Implementation (___) in Java

```
1  /** An ADTname object provides ...
2    * model ...
3    * @inv public invariant ...
4    */
5  public class ADTname {
6      /** Representation ___ */
7      private ___ ___;
8      /* ___ private invariant ___ abstraction function ___ */
9
10     /** Constructs ...  @pre ... @post ... @throws ... */
11     public ADTname() { ___ }
12
13     /** Queries ...  @pre ... @return ... @throws ... */
14     public ReturnType queryName(...) { ___; return ___; }
15
16     /** Changes ...  @pre ... @modifies ... @post ... @throws ... */
17     public void commandName(...) { ___ }
18 }
```

Abstract Data Type: Usage in Java

- Declare variable: `ADTname v = new ADTname(...);`

  E.g. `Fraction f = new Fraction(1, 2), g = new Fraction(2, 3);`

- Apply operations: `... v.operationName(...) ...`,
  in a state where the corresponding precondition holds

  E.g. `f.add(g)`

- Syntactically, this looks the same for all ADTs.

  (Recall that usage syntax for primitive types wildly varies.)

Data Abstraction in Java: References, Sharing, Aliasing

- A variable of a primitive type has as value a value of that type.
- A variable of **`class`** type T does *not* have an object of type T as value, but
  - either *the* **name** *of an object of type* T,
  - or the special value **`null`**.

  This name is a *unique identifier* of that object (a.k.a. *reference*).
- This allows *sharing*, a.k.a. *aliasing*: two distinct variables name the same object. It can help improve (memory and time) efficiency by avoiding the copying of data. But it complicates reasoning.
- Operator "==" on expressions of a class type compares object *names*, and not the object states: `"abc" != "abc"`

Data Abstraction in Java: Aliasing Example

```
Card u = new Card(Rank.Ace, Suit.Spades);
Card v = new Card(Rank.Ace, Suit.Spades);
Card w = new Card(Rank.Queen, Suit.Hearts);
Card y = u;

// u, v, and w name (refer to) distinct objects:
//   u != v  &&  v != w  &&  w != u

// u and v refer to objects having the same value (state)

// u and w refer to objects having different values

// u and y name the same object (aliasing): u == y
```

Data Abstraction: Mutability

- A data type T implemented as a class is said to be immutable when none of its methods **modifies `this`** or a parameter of type T. I.e., the state of a T object cannot change after construction.

  N.B. A T method could still modify a parameter of another type.

  E.g.: `String`; `immutable/Fraction`

  Immutable types need various/parameterized constructors.

- It is said to be mutable otherwise.

  E.g.: arrays; `mutable/Fraction`

Changeability of variables versus objects

Note the difference between (changing)

- the state of a variable
  (i.e., its primitive value or the name of an object it referes to) and
- the state of an object.

These can change independently:

- A variable can be made to refer to another object (unless it is **`final`**), without changing the state of the objects involved.
- The state of an object can be changed (unless its class is immutable), without changing the variables that refer to it.

Data Abstraction: equality and similarity

Two objects are —at the time of comparison— said to be

- equal when they are *behaviorally indistinguishable*:
  - every sequence of operations (queries and commands), when applied to both objects, yields the same result.
  - `a.equals(b)`; e.g. `"abc".equals("abc")`
- similar when they are *observationally indistinguishable*:
  - every sequence of *queries* only (no commands), when applied to both objects, yields the same result.
  - `a.similar(b)`

Data Abstraction: Comparison of (im)mutable objects

- In general: `a == b` implies `a.equals(b)`, but not vice versa.
- Mutable objects are equal when they are the same object (==):
  - Otherwise, you can change one without changing the other.
  - `a.equals(b)` inherited from `Object` (returning `a == b`) suffices
- In general: `a.equals(b)` implies `a.similar(b)`, but not vice versa.
- Immutable objects are equal when they are similar (same state):
  - Immutable types must override implementation of `equals()` to coincide with `similar()`.

Data Abstraction: `equals()`

- `equals()` must be

  reflexive: `a.equals(a)`

  symmetric: `a.equals(b) == b.equals(a)`

  transitive: if `a.equals(b) && b.equals(c)` then `a.equals(c)`

- A class can have several (overloaded) `equals()` methods.

Data Abstraction: `clone()`

- To offer method `clone()`, a class must implement `Cloneable`; otherwise, `clone()` will throw `CloneNotSupportedException`.
- `clone()` returns a copy of **`this`**.
- The copy should be similar but not identical to **`this`**.
- The default implementation inherited from `Object` constructs a new object and copies all instance variables (shallow copy).
- This is sufficient for immutable objects.
- Mutable objects should implement their own `clone()` (doing a deep copy)
- However, it is better to avoid `clone`.

Java Limitation

Unfortunately, in a Java **`class`**, *specification* and *implementation* are combined into a single interleaved description.

Javadoc helps extract just the specification.

Alternative: Put specification in an **`abstract class`** or **`interface`**

- Java **`abstract class`** defines a *type* with partial implementation;

- Java **`interface`** defines a *type* without any implementation.

Inheritance

- A new class can be defined by extending an existing class:

```
public class RuntimeException extends Exception
public class StatisticsWithVariance extends Statistics
```

  Default: `extends Object`

- Terminology: `StatisticsWithVariance` is a subclass of `Statistics`; `Statistics` is the superclass of `StatisticsWithVariance`

- Subclass inherits all instance variables and methods in superclass, both method *signatures* and method *implementations* (if present).
  It is strongly recommended to inherit also the method *contracts*.
  The compiler does not enforce inheritance of method contracts.

- A subclass can extend only *one* class; it can add new methods and instance variables, and override inherited methods.

Polymorphism

- Each variable has a compile-time type: primitive or reference.
- During execution, each (initialized) variable has a value.
- "A variable of a primitive type always holds a value of that exact primitive type."
- "A variable of a *class type* T can hold a **`null`** reference or a reference to an instance of class T or of any class that is a *subclass* of T."
- "A variable of an *interface type* can hold a **`null`** reference or a reference to any instance of any class that *implements* the interface."

Liskov Substitution Principle

Let `U` be a subclass (possibly in multiple steps) of `T`.

Type `U` is called a subtype of type `T`, when

> In each place where an object of type `T` can be used,
>
> you can substitute an object of type `U`,
>
> without affecting the correctness of the program.

It is good practice to ensure that subclasses are also subtypes.

Abstract Classes (no details, just the concept)

- A method can be declared **`abstract`**.
- An abstract method does not have a complete implementation:

  Instance variables and/or method bodies can be missing.
- A class containing abstract methods must be declared **`abstract`**.
- An abstract class cannot be instantiated.
- A subclass can define (additional) instance variables and implementations for inherited **`abstract`** methods, by *overriding*.

  It is recommended to annotate such methods:

```
@Override
public String stringOf() { ... }
```

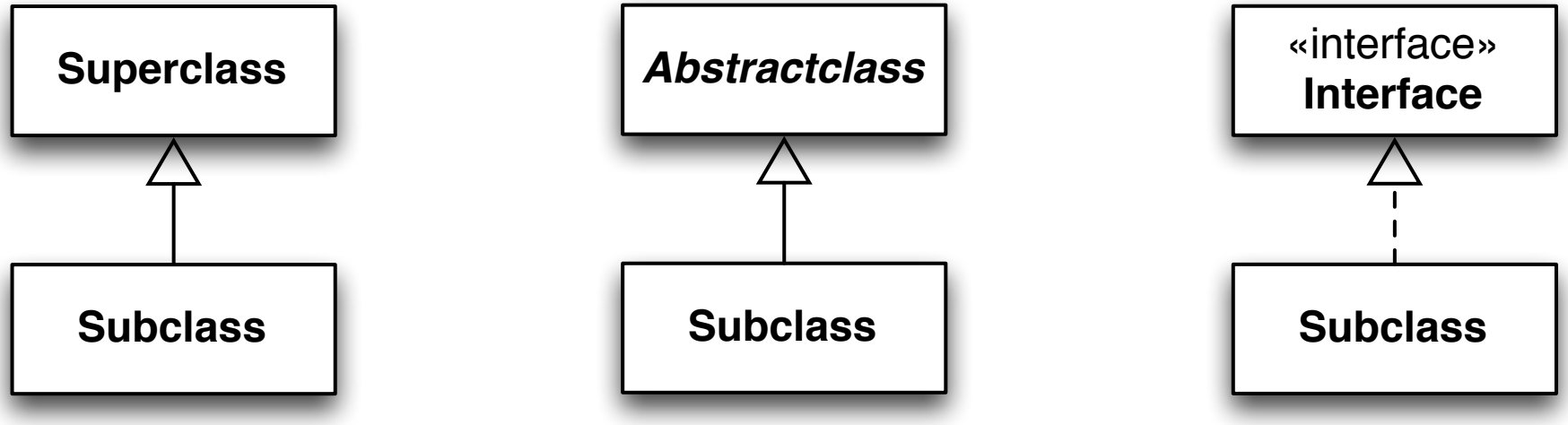
Interfaces (no details, just the concept)

- An **`interface`** defines a collection of method headers, with contracts
- An interface has neither instance variables, nor method bodies. It can define constants with **`public static final ...`**
- An interface is somewhat like a class with only abstract methods.
- A class can implement one or more interfaces via **`implements`**, by implementing each of the methods in the interface(s).
- An interface can be viewed as a union type: Its values are the values of all classes that implement the interface.

Generic Classes and Interfaces (no details, just the concept)

- A type can be defined to have one or more type parameters.
- Example: `List<E>`, `ArrayList<E>`
- The generic type parameter must be replaced by a specific type.
- Example: `ArrayList<String>`

UML Class Diagrams



Strategy Design Pattern

- Problem: Accommodate multiple implementations of same ADT
- Problem: Possibility to select implementation at runtime
- Solution:
  - Put specification of ADT in (abstract) class or interface.

    ```
    IntRelation
    ```

  - Put implementations in subclasses of specification.

    ```
    IntRelationMapOfSets extends IntRelation
    ```

  - Declare variable with specification type.

    ```
    IntRelation friendship
    ```

  - Assign to variable a class of an implementation type.

    ```
    friendship = new IntRelationMapOfSets();
    ```

Strategy Design Pattern: Example

```
 1 public class FaceBook {
 2
 3     private IntRelation friendship; // accommodates all implementations
 4
 5     public FaceBook(final IntRelation friendship) {
 6         this.friendship = friendship; // should be a concrete implementation
 7     }
 8
 9     public void invert() {
10         ...
11             friendship.add(a, b); // still open which implementation it uses
12         ...
13     }
14 }
15
16 ...
17     FaceBook myFB = new FaceBook(new IntRelationMapOfSets());
18     // myFB.invert() uses add() as implemented in IntRelationMapsOfSets
```

Programming techniques: General versus Java-specific

The aim of this course is to teach *general* programming techniques.

That is, techniques that are useful for many programming languages.

However, each programming language requires its own 'mind set' (often, that is why it was invented).

There is a danger that Java-specific matters take the upper hand.

Java details are hard to avoid:

- Concrete applications of techniques help to understand them.
- 'Real' programs are written in a concrete programming language.

Assignments Series 2

- Refresher: book by Eck: §2.3.3 (Enums), and §5.1, §5.2, §5.5
- Implement `IntRelation` with `Map<Integer, Set<Integer>>`

Summary (1)

- Java **`class`** can be used for

  1. Library of static constants and methods (no state)
  2. Defining an enumeration type: a set of related constants
  3. Defining a record type: a labeled-product type
  4. Defining an abstract data type (ADT)

- ADT encapsulates/hides the *representation* of the abstract data and the implementation of *operations* on that data.

- It separates the *use* of and the *implementation* of a data type, through a *specification* in the form a of a two-sided contract.

- This allows modification of the implementation without requiring modification of the using environment, and vice versa.

Summary (2)

- Immutable versus mutable types
- `equals()`, `similar()`, `clone()`
- Single inheritance: **`class`** `SubClass` **`extends`** `SuperClass`
- **`abstract class`**, with **`abstract`** methods
- **`interface`**
- **`class`** `MyClass` **`implements`** `InterfaceA, InterfaceB`
- Polymorphism: *runtime* type of a variable value can be a subtype of the variable's *compile-time type*

Summary (3)

Strategy Design Pattern

- Problem:
  - Allow runtime selection among multiple ADT implementations
- Solution:
  - Put the ADT *specification* in a superclass, or interface
  - Put each ADT *implementation* in a subclass that extends the specification class, or a class that implements the specification interface; add representation and method implementations
  - Declare variables/parameters with the specification type
  - Assign value of the selected implementation type