2IPC0 Programming Methods

From Small to Large Programs

Loek Cleophas
Eindhoven University of Technology
Department of Mathematics & Computer Science
Software Engineering & Technology Group

http://canvas.tue.nl/courses/473

Overview

- Using Nested classes
- Generic type definitions
- Façade Pattern
- Looking back
- Checklist for design of larger Java programs

Dynamic Structure of a Running Java Program

- Collection of classes, and objects instantiated from classes, both having (static/instance) variables
- Each variable holds a value of a primitive type or a reference to an object, forming a labeled directed graph (network)
- Unreachable objects can be removed by the Garbage Collector
- Stack of nested method invocations (calls) that are active

  At the bottom of the stack is the designated `main` method.
- Each active method invocation has parameters, local variables and a current instruction address in a stack frame
- Instruction at current address of topmost stack frame is executed

Top-level and Nested Classes (or Interfaces)

- Each compilation unit defines one **`public`** class and, next to it, possibly other non-**`public`** classes, called top-level classes.

- A class can also be defined *inside* another class.

  These are called nested classes, coming in four kinds:

  1. **`static`** member class (almost equivalent to top-level class)
  2. non-**`static`** member class
  3. named local class (defined inside a method)
  4. anonymous class (defined in **`new`** expression, without name)

  The latter three are also called inner classes.
  An inner class has access to the members of its outer classes.

`en.wikipedia.org/wiki/Inner_class`

`static` member classes already encountered

In *Powerize*: **`static class`** `Power`

This is equivalent to

- putting it in the same file *above* or *below* **`class`** `MathStuff`, or
- putting it in a separate file `Power.java`

When put in the same file, **`class`** `Power` cannot be **`public`**

Advantage of nested class: keeps things closer together.

Disadvantage of nested classes: outer class becomes less readable
(Code folding in IDE may mitigate this disadvantage.)

Useful for "small" (auxiliary) classes, like records.

static member class example

```java
public class StaticMemberClassExample {
    public static void main(String[] args) {
        TopWithStatic.Nested.print();
    }
} // End of driver

/** A top-level class with static member class. */
class TopWithStatic {
    private static final int N = 42;

    /** A static member class nested inside TopWithStatic. */
    public static class Nested {
        public static void print() {
            // access private field N of enclosing class
            System.out.println(N);
        }
    } // End of nested class
} // End of top-level class
```

`static` member class example: Unnested

```java
public class UnnestedStaticClassExample {
    public static void main(String[] args) {
        Unnested.print();
    }
} // End of driver

/** A top-level class, whose static member class has been "unnested".
  * N.B. Private members cannot be accessed directly from unnested class.
  * @see StaticMemberClassExample
  */
class TopWithUnnestedStatic {
    private static final int N = 42;

    /** Gets N. (NEW) */
    public static int getN() {
        return N;
    }
} // End of top-level class
```

static member class example: Unnested

```java
/** Class unnested from Top. */
class Unnested {
    public static void print() {
        // the next line cannot access private N directly
        System.out.println(TopWithUnnestedStatic.getN());
        // NEW: uses getter for N
    }
} // End of unnested static member class
```

non-`static` member classes already encountered

`RangeIterator` inside class `Range`

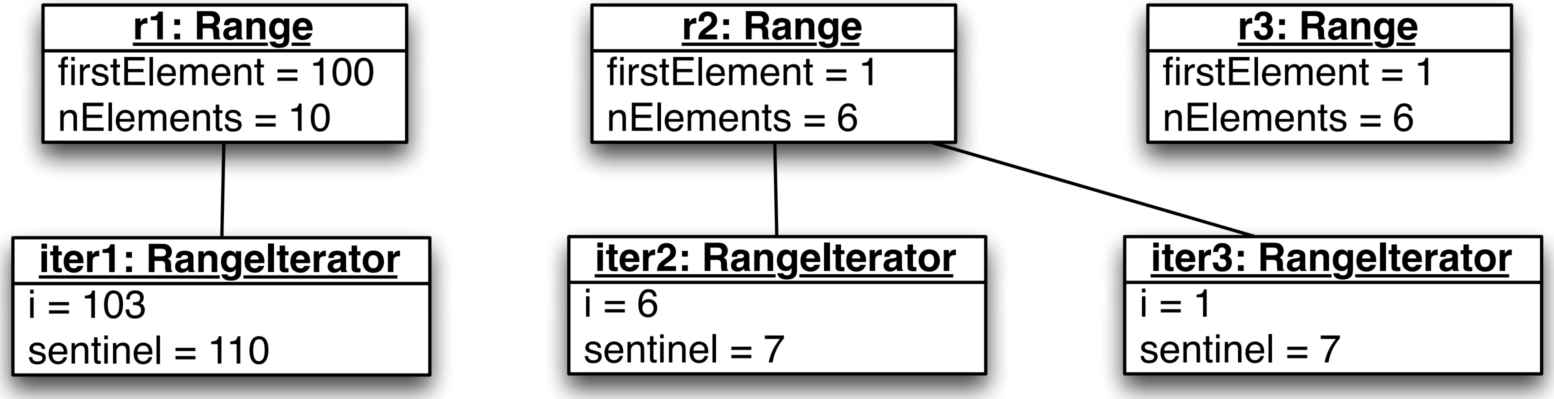
Each instance `iter` of `...Iterator` is *associated* to the instance `r` of `Range`, 'inside' which it was constructed.

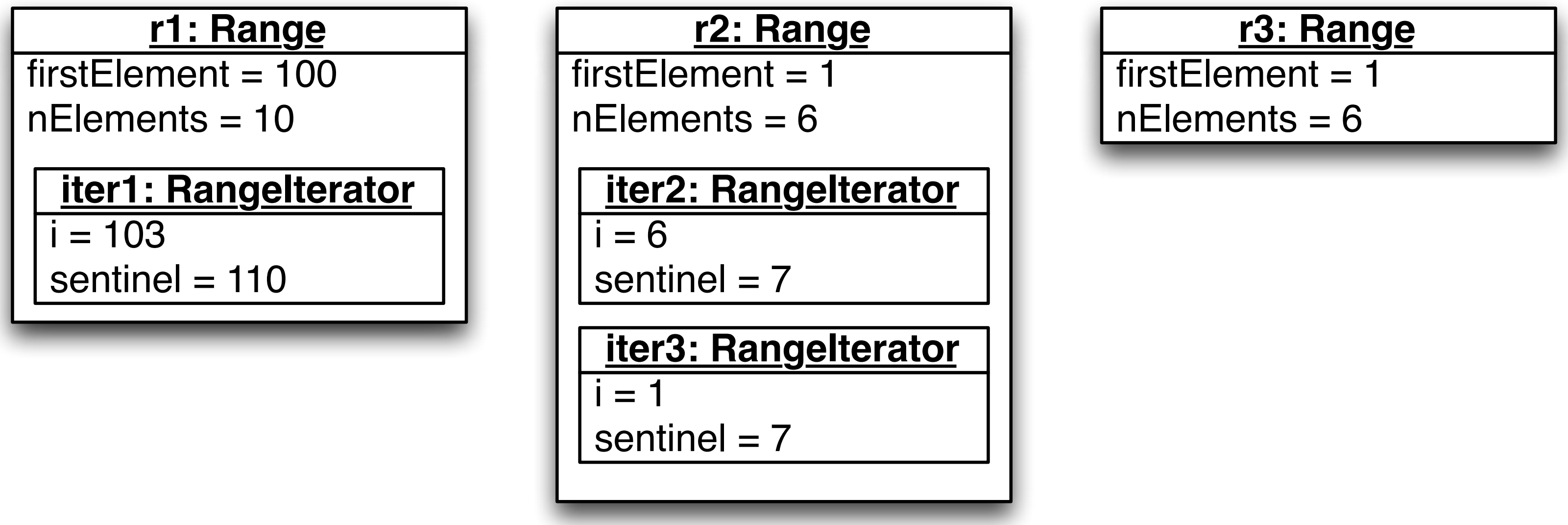
Alternatively, you can imagine that inner objects are located inside their associated outer object.

Methods of `iter` can access all members of `r`,
including **`private`** members.

Additional advantage: keeps (nested) class simpler.

Nested Classes – “Nested” Objects





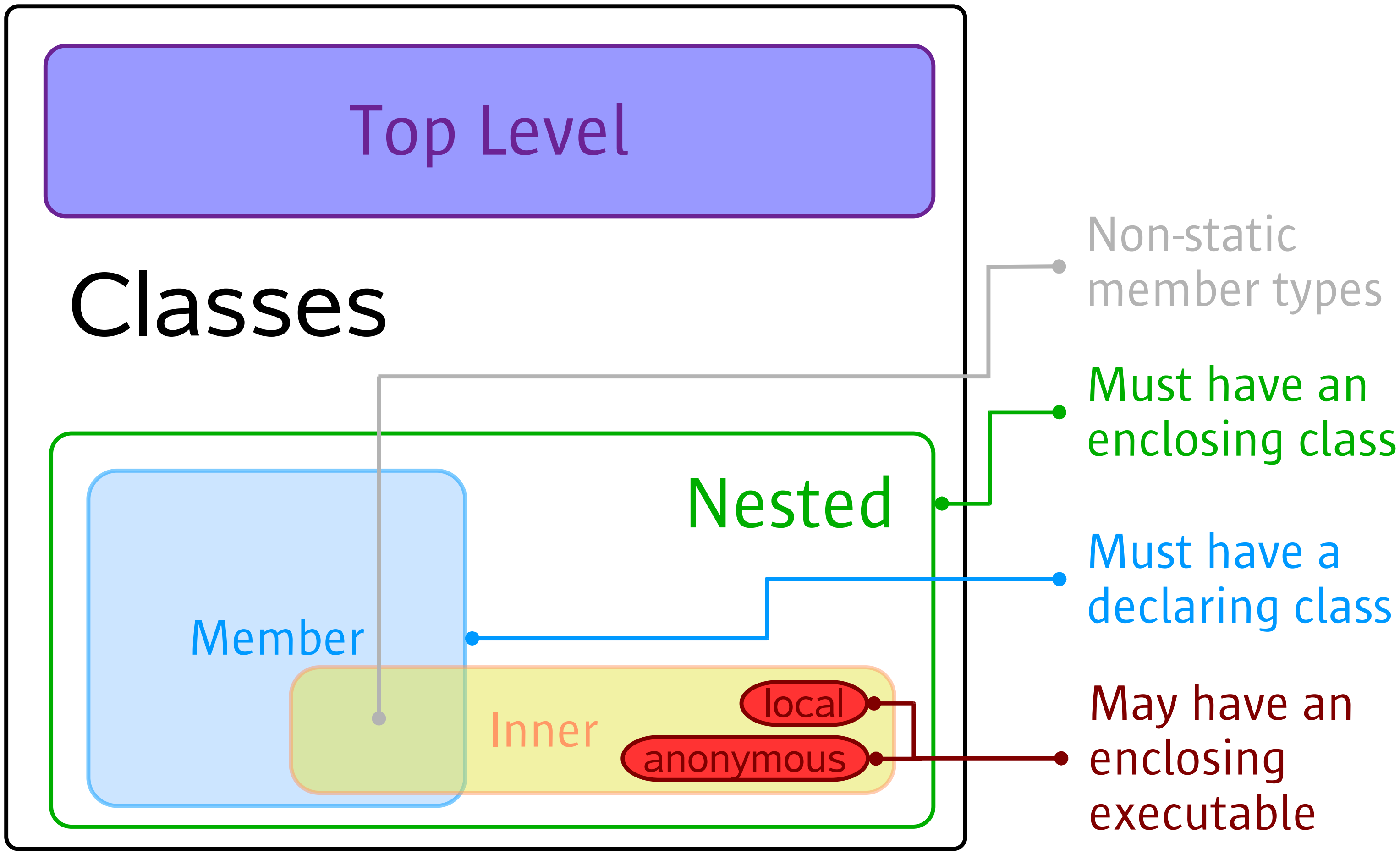
Inner and Outer Classes

```java
/** Driver. */
public class NonStaticMemberClassExample {

    public static void main(String[] args) {
        TopWithNonStatic top = new TopWithNonStatic(42);
        // construct instance of Inner associated with top
        TopWithNonStatic.Inner inner = top.new Inner(); // NOTE the syntax
        inner.print(); // inner "knows" that it is associated with top
    }

} // End of driver


/** A top-level class with inner class. */
class TopWithNonStatic {

    private int c;

```

Inner and Outer Classes

```java
    /** Constructor initializes private int field. */
    public TopWithNonStatic(final int c) {
        this.c = c;
    }

    /** A non-static member class. */
    public class Inner {

        void print() {
            // accesses private field c of enclosing class
            System.out.println(c);
        }

    } // End of inner class

} // End of top-level class
```

Taxonomy for Kinds of Class Definitions



blogs.oracle.com/darcy/entry/nested_inner_member_and_top

Static Member Classes

- Convenient for logical grouping
- Nested class can refer *only* to **`static`** members of enclosing class.
- Nested class can refer *directly* to *all* **`static`** members of enclosing class, without qualifying the name, including **`private`** members.
- Instances of the nested class are *not* automatically associated with objects of the enclosing class.
- It is possible to refer to **`static`** member class from outside the enclosing class, by qualifying its name with name of enclosing class.

See: `StaticMemberClassExample`, `UnnestedStaticClassExample`

Non-Static Member Classes

- Instances of the inner class are automatically associated with one object of the enclosing class.
- New constructor call syntax: `outer.new Inner()`
- Inner class can refer to *all* members of enclosing class, including **`private`** members.
- Enclosing class can refer to *all* members of inner class, including **`private`** members.
- Inner class can refer *directly* to members of enclosing class, without qualifying the name.
- Inner classes cannot contain **`static`** members, except **`static final`**

See: `NonStaticMemberClassExample`, `UnnestedNonStaticMemberClassExample`, `MutualAccess`

Local and Anonymous Classes

- Also see non-static member classes.
- Local classes can access local variables and parameters *that are declared **`final`***.
- Anonymous classes *occur only in a **`new`** expression*; this implicitly involves an **`extends`** or **`implements`** clause.

  Thus, they can *override* methods on-the-fly.
- Anonymous classes can easily be turned into a named local class.

  (Easier than turning member class into top-level class)

See: `LocalClassExample`, `AnonymousClassExample`, `UnnestedLocalClassExample`

Anonymous Classes: Limitations

- Cannot define their own constructors

- Can introduce extra methods

  Usually, these are only called from inside

  (Called as auxiliary method from an overridden method)

  Extra methods are harder to call from outside:

  ```
  new Object() { int sqr(int n) { return n * n; } }.sqr(3)
  ```

  Variable to store this object must be declared with type `Object`

  Hence, method `sqr` cannot be found

See: `AnonymousExtra`

Benefits of Nested Classes (and Interfaces)

- **Logical grouping** (increased coherence)

  Keep related things close together.

- **Encapsulation** (decreased coupling)

  Only provide possibility to couple things that need to be coupled.

- **Improved readability and maintainability** of source code

  A consequence of the preceding two benefits

- **Simpler code**

  Inner class can access members of enclosing class, without needing an explicit reference to it.

`download.oracle.com/javase/tutorial/java/javaOO/nested.html`

Using an Inner Class to Decrease Coupling Possibilities

Suppose **`class`** `B` needs to access member `x` (method, field) of **`class`** `A`.

Solution without inner class	Solution with inner class

`public class A {`
`    public T x...`
`}`
 	
`class B {`
`    ... A obj ...`
`    ... obj.x... ...`
`}`
	`public class A {`
	`    private T x...`
	
	`    class B {`
	`        ... x... ...`
	`    }`
	
	`}`

Everything can access `x` in `A`	Only `A` and `B` can access `x` in `A`

Both "work"; they differ in risks during development and evolution.

Refactorings for Class Nesting

- top-level class $\longleftrightarrow$ **`static`** nested class

- top-level class $\longleftrightarrow$ inner class (non-**`static`** member class)

  $\longleftarrow$: need to add explicit reference to outer class

- inner class $\longleftrightarrow$ named local class

  $\longleftarrow$: need to add explicit instance variables for final local variables accessed by local class

- named local class $\longleftrightarrow$ anonymous local class

  $\longrightarrow$: not always possible (see limitations of anonymous classes)

NetBeans can assist in these refactorings.

– O – O – O – O – O –

NEW TOPIC

Generic Type = Parameterized Type

- Example in JCF: `List<E>` is a generic type

  `E` is a formal type parameter

- **Usage**: Substitute concrete class type for formal type parameter

  Examples: `List<String>`, `List<Set<Integer>>`

- Auto (un)boxing and wrapper classes: **`int`** $\leftrightarrow$ `Integer`

  Makes it possible to use primitive types as actual type parameters

- Benefits: improved readability, resusability, robustness

See: `docs.oracle.com/javase/tutorial/java/generics`

Defining Generic Types

- Formal type parameter can be used as a class type in the definition

```
public class Pair<A, B> {
    public A a;
    public B b;
}
```

- When used as `Pair<Integer, String>` it is equivalent to

```
public class PairIntegerString {
    public Integer a;
    public String b;
}
```

Actual type parameters are substituted for formal type parameters everywhere in the class definition

Generic Types: Historic Motivation

- Pre-Java 5: `List` concerns a list of `Object`

  User responsible for proper typing

```
List list = new ArrayList(); // intended as list of integers
list.add( "okay" ); // no complaint, but unintended
list.add( 42 );
int i = (Integer)list.get(1); // cast needed
```

- Java 5 and beyond: User can indicate intention to compiler

```
List<Integer> list = new ArrayList<Integer>();
list.add( "okay" ); // compile error
list.add( 42 );
int i = list.get(1); // no cast needed
```

Generic Type Definitions: Type Inference

- Recurring code fragments using generics:

```
Set<String> messages = new HashSet<String>();
```

- Can be shortened to

```
Set<String> messages = new HashSet<>();
```

- `<>` is known as the diamond
- Compiler does type inference to determine the missing type

Generic Types: Constraining Actual Type Parameters

- In generic class `C<T>`, objects having type parameter `T` as type can only be used as `Ojbect`

  ```
  class C<T> {
      private T t;;
      public void m() { . . . t.xxx(); . . . }
  }
  ```

  `xxx()` must be known in `Object`

- Actual type parameters can be constrained: `C<T extends U>`, where `U` is a concrete class type

  Inside `C<T extends U>`, variable `T t` can be used as a `U` object

  Cf. preconditions of methods

See: §10.5.4 (Bounded Types) in David Eck's book

Generic Types: Complications with Subtypes

- If `U` is a subtype of `T`, then `C<U>` is *not* a subtype of `C<T>`

  Method **`void`** `m(List<Object> list)` cannot be called as

  `m(`**`new`**` ArrayList<String>)`

  Method **`void`** `m(List<?> list)` can

- Some limitations can be overcome with wildcards:
  - `C<U>` is a subtype of `C<?>`, for any type `U`
  - `C<U>` is subtype of `C<?` **`extends`** `T>`, if `U` **`extends`** `T`
  - `C<S>` is subtype of `C<?` **`super`** `T>`, if `T` **`extends`** `S`

See: `docs.oracle.com/javase/tutorial/java/generics/wildcards.html`
§10.5.3 (Type Wildcards) in David Eck's book

NEW TOPIC

Façade Pattern Motivation

- A class or package of classes offers a large/complex interface:
  - many methods on the interface
  - constraints on order of method calls (usage protocol)
- Many clients of this class/package do not need all functionality

Concern: How to simplify the interface?

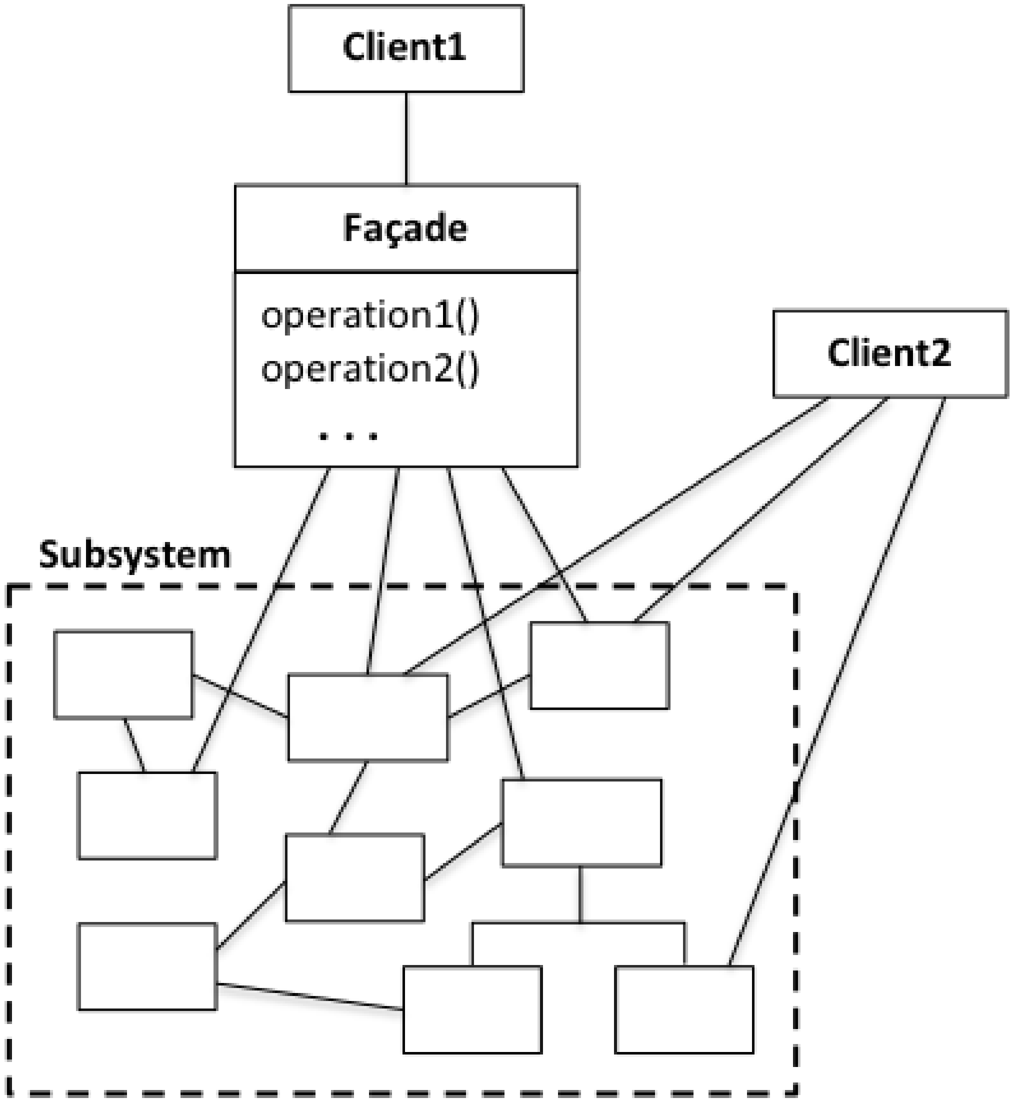
Façade Design Pattern (adapted from Eddie Burris)

Intent

With the Façade design pattern,

- you offer a single point of access for clients;
- you offer a simpler, more abstract, interface for clients;
- you decouple client code from multiple subsystem classes.

Façade Pattern (from Burris)



– O – O – O – O – O –

NEW TOPIC

Looking Back on First Half of the Course

- Write readable code (always)
- Manage Complexity: Divide & Conquer (& Rule), Modularization
- Procedural abstraction, contracts, functional decomposition
- Robustness: dealing with errors
- Data abstraction: decouple use and implementation of data type
- Iteration abstraction
- Test-Driven Development

Source Code Should Be Written with Utmost Care

- Source code is not only intended for the compiler.

  Source code should be readable and verifiable by other engineers.

- Adhere to a coding standard:

  Pay attention to layout, (javadoc) comments, naming, structure.

- This prevents mistakes, and eases finding and repairing of defects.

Manage Complexity: Modularization

- Separation of Concerns: *Divide & Conquer* (and hence *Rule*)

- For each facility (function, type, iterator, package, . . .), separate

$$\begin{array}{ccccc} & & \textbf{Contract} & & \\ & \nearrow & \| & \nwarrow & \\ \textbf{Use} & & \| & & \textbf{Implementation} \end{array}$$

  Use and implementation are based on the (same) contract.

  Use and implementation are not directly 'coupled'.

- Always *program to an* *abstraction* (contract, interface),
  not to a 'concretion' (use, implementation).

- Divide & conquer serves many purposes, including maintainability.

Procedural Abstraction

- Abstract from data operated on (through parameters)
- Abstract from realization of operation (when viewed from usage)
- Abstract from context of use (when viewed from implementation)
- Guidelines for functional decomposition

Robustness: Errors and Exceptions

IEEE terminology: failure, defect (fault, bug), mistake, error

For non-private methods:

- the precondition should be as weak as possible:
  unless unacceptable for performance reasons.

- the contract specifies relevant exceptions and their conditions:
  `@pre P` and `@throw E if ! P`

You are encouraged to check preconditions, e.g. via `assert`,
also in **`private`** methods. Assertion checking is disabled by default!

Data Abstraction

Abstract Data Type = set of (abstract) values and corresponding operations: construct, destroy, query, modify

In Java

Specification: **`class`** name, **`public`** method headers and contracts, public invariant
Optionally: public constant names

Implementation: **`private`** instance variables, private (rep) invariant, abstraction function, public method bodies
Optionally: public constant values, private methods

N.B. Variable of **`class`** type is a reference: *aliasing*!

Java Built-in Protections for Modularization

- Functions (class methods): local variables
- Data types (classes): instance variables, methods

Modifier	Access Level			
	Class	Package	Subclass*	World
`private`	*Yes*	No	No	No
no modifier	*Yes*	*Yes*	No	No
`protected`	*Yes*	*Yes*	*Yes*	No
`public`	*Yes*	*Yes*	*Yes*	*Yes*

*Outside package

Step-by-step (Test-driven) ADT Development Plan

1. Gather and analyze requirements.
2. Choose requirement to develop next.
3. Specify class & methods informally: javadoc summary sentences.
4. Specify formally: model with invariants, signatures, and contracts.
   Class w/o implementation: no data rep, empty method bodies.
5. Create a corresponding unit test class.
6. Implement rigorous tests.
7. Choose data representation and implement class methods.
8. Test the implementation.
9. Refactor and retest.

Iteration Abstraction

- Problem: Visit each item of a collection exactly once
- Abstract from type of collection, type of items, how to iterate
- An *iterator object* maintains the state of the iteration.
- In general, an iterator object implements `Iterator<T>` providing methods **`boolean`** `hasNext()`, `T next()`, and optionally `remove()`.
- To use enhanced **`for`** statement, collection implements `Iterable<T>`, i.e., provides a method `iterator()` returning an `Iterator<T>`.
- **`for`** `(` *Type* *Identifier* `:` *Expression* `)` *Statement*

Summary

- Nested classes, and iterators
- Generic data type definitions
- Summary of first half of the course

  Also see: Checklist for design of larger Java programs