# 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

- Iteration Abstraction: Ch. 10.1.4, 10.1.5, in Eck
- Iterator Design Pattern: Ch. 3 in Burris
- Nested Classes: Ch. 5.7 in Eck

## Iteration Abstraction: Specify, Use, Implement

Iteration Abstraction: Facility for iteration that abstracts from

- type of collection
- type of items in collection
- implementation details of how to accomplish iteration
  e.g. choosing an order

Iteration abstraction provides a uniform* solution.

*Unfortunatly, details of iteration abstraction are programming-language specific.

## Iteration Abstraction: Specify, Use, Implement

- How to specify it: Java incorporates a standard specification
- How to use it (as a client): relatively easy
- How to implement it (as a provider): can be tricky

## Enhanced `for` Statement, or the 'for-each' Loop

New in Java 5.0:

**`for`** ( *Type Identifier* **`:`** *Expression* ) *Statement*

*Identifier* is a local variable, whose values are of the declared *Type*, iterating over the collection of *Type* items defined by *Expression*.

Read as 'for each . . . in . . . '.

*Expression* can be

- an array, e.g. obtained through method `values()` from an **`enum`**
- a type that implements interface `Iterable` (a subtype of `Iterable`)

## 'for-each' Loop Examples

- Iterate over an array:

```
float[] a;

for (float f : a) { ... f ... }
```

- Iterate over the values in an **enum**:

```
private enum PrimColor { RED, GREEN, BLUE }

for (PrimColor c : PrimColor.values()) { ... c ... }
```

- Iterate over a type that implements `Iterable`

```
HashSet<Card> cards;

for (Card card : cards) { ... card ... }
```

## Semantics of enhanced `for` Statement: `for ( T v : E ) S`

- if `E` is an array:

```
T[] r = E;  // fresh identifier r; E evaluated only once
    // r is a reference, and not a copy
for (int i = 0; i != r.length; ++ i) {
    T v = r[i];
    S  // operates on v  (N.B. i is invisible)
}
```

- if `E` is subtype of `Iterable<T>` (i.e., `E` provides a default iterator):

```
for (Iterator<T> iter = E.iterator(); iter.hasNext(); ) {
    T v = iter.next();
    S  // operates on v
}
```

## Interfaces `Iterable` and `Iterator`

```
public interface Iterable<T> {
    /** Returns an iterator for a collection over type T. */
    Iterator<T> iterator();
}

public interface Iterator<T> {
    /** Returns whether a next item is available. */
    boolean hasNext();

    /** Returns the next item in the iteration.
     *  @throws NoSuchElementException if not available */
    T next();

    void remove();  // optional (not treated here; see book)
}
```

N.B. Cannot invoke `remove()` in for-each loop (anonymous iterator).

# Iterator and `iterator` method that creates an iterator

```
 1 import java.util.Iterator;
 2
 3 /** C is collection over type T (comparable to ArrayList<T>) */
 4 public class C implements Iterable<T> {
 5     private ... r;  // data representation of the collection
 6
 7     public Iterator<T> iterator() { return new ForIterator(...); }
 8
 9     private class ForIterator implements Iterator<T> {
10         private ... s; // private state of iterator
11
12         ForIterator(...) { ... s ... } // constructs initial state
13
14         public boolean hasNext() { ... r, s ... }
15
16         public T next() throws NoSuchElementException { ... r, s ... }
17     }
18 }
```

## Iterators in general

- Standard iterator to be used in for-each loop:
  - Collection class must implement interface `Iterable<T>` and define method `iterator()` returning an `Iterator<T>`.
  - See example `Range.java`
- Iterator in general (not usable in for-each loop):
  - Collection class must define one or more methods returning an `Iterator<T>`.
  - The name of the iterator constructor is not prescribed.
  - See example `DownRange.java`

## Iterators: Loose Ends

- Iterators cannot be reused.

  Each iteration involves a new iterator object.

- Multiple iterators over the same collection can be active at the same time.

- E.g. iterations over the same collection can be nested.

  This involves multiple iterator objects.

## Design Issues

- Data types that store a collection of items typically offer one or more iterators.
- Mutable collections require that the loop using an iterator does *not* change the collection 'outside' the iterator.
  Doing so will result in a `ConcurrentModificationException`.
- The `Iterator` interface offers one safe modification operation: `remove()`, but it is not always implemented.
- For use in the for-each statement, the collection should implement `Iterable<...>` and provide the iterator constructor `iterator()`.
- Other iterators must be used through standard loops using `hasNext()` and `next()`, or wrapped in an `Iterable<...>`.

## Worse Alternative (Anti-Pattern)

- Return a (simplified) copy of the collection, so that the client can iterate over it using the iterator of the copy
- ```
  class CardPile {
      private List<Card> pile;

      public List<Card> getPile() { return pile; }
  }

  ...

      CardPile deck;
      ....
      for (Card card : deck.getPile()) { ... }
  ```
- Why is this potentially dangerous?
  ```

## Worse Alternative (Anti-Pattern)

- Breaks encapsulation!
- Danger: Client can abuse data rep and break the rep invariant

```
deck.getPile().get(0).turnOver()
```

- Better, but still potentially dangerous:

```
class CardPile {
    private List<Card> pile;

    public Iterable<Card> getPile() { return pile; }
}
```

Client can cast: `((List<Card>)getPile()).get(0).turnOver()`

- Also: Performance penalty, when copying data to collection

# How to Offer Multiple Standard Iterators?

- Problem: Provide standard up and down iterators for `Range`
- Solution: Return an iterable that provides a standard iterator
- Example: `DualRange`

# Iterator as Abstract Data Type

```
1  class MyIterator implements Iterator<E> {
2
3      /** Constructs initialized iterator: @post nothing visited. */
4      public MyIterator() { .... }
5
6      /** @pre true
7        * @post \result == not all items have been visited */
8      public boolean hasNext() { ... }
9
10     /** @pre hasNext()
11       * @post \result == an unvisited item, now marked as visited
12         @throws NoSuchElementException  if pre violated */
13     public E next() { ... }
14 }
```

N.B. `next()` is query-and-command (side-effect)

Cleaner would be: query `current()`, command `step()`

## Testing an Iterator

- Test that iterator visits each item in collection exactly once:
  - Iterate over the collection,
  - check that each item returned by `next()` belongs to collection
  - and was not visited before (need to keep track of visited items).
  - When done: Check that all items were visited, e.g. by counting.
- If the collection is mutable, check that it was not modified.
- Check that `next()` throws `NoSuchElementException` when `! hasNext()`

## Dynamic Structure of a Running Java Program

- Collection of classes, and objects instantiated from classes
- 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`

# What Nested Classes Look Like

```
1  class TopLevel {
2
3      static class StaticNested { ... }
4
5      class NonStaticNested { ... }
6
7      void method() {
8          class NamedLocal { ... }
9
10         Iterable<Integer> iter = new Iterable<>() {
11             // anonymous inner class
12             public Iterator<Integer> iterator() { ... }
13         };
14     }
15 }
```

## Why Nested Classes, and How to Play with Them

- Why: For convenience only; they can be eliminated (at a cost)
- Useful refactoring operations on source code:
  - Given two separate classes: nest one inside the other

    NetBeans: right-click > Refactor > Move Class...
  - Given two classes, one nested inside the other: move up/out

    NetBeans: right-click > Refactor > Move Inner to Outer Level...
  - Given an anonymous class: turn into named local class

    NetBeans: right-click > Refactor > Convert Anonymous Class to Inner...

## `static` member classes

```
1 public class TopLevel {
2
3     static class Nested {
4         ....
5     }
6 }
```

is equivalent to

- putting it in the same file *below* the enclosing class
  in which case it cannot be **`public`**
- putting it in a separate file `Nested.java`
  in which case it must be **`public`**

## non-`static` member classes already encountered

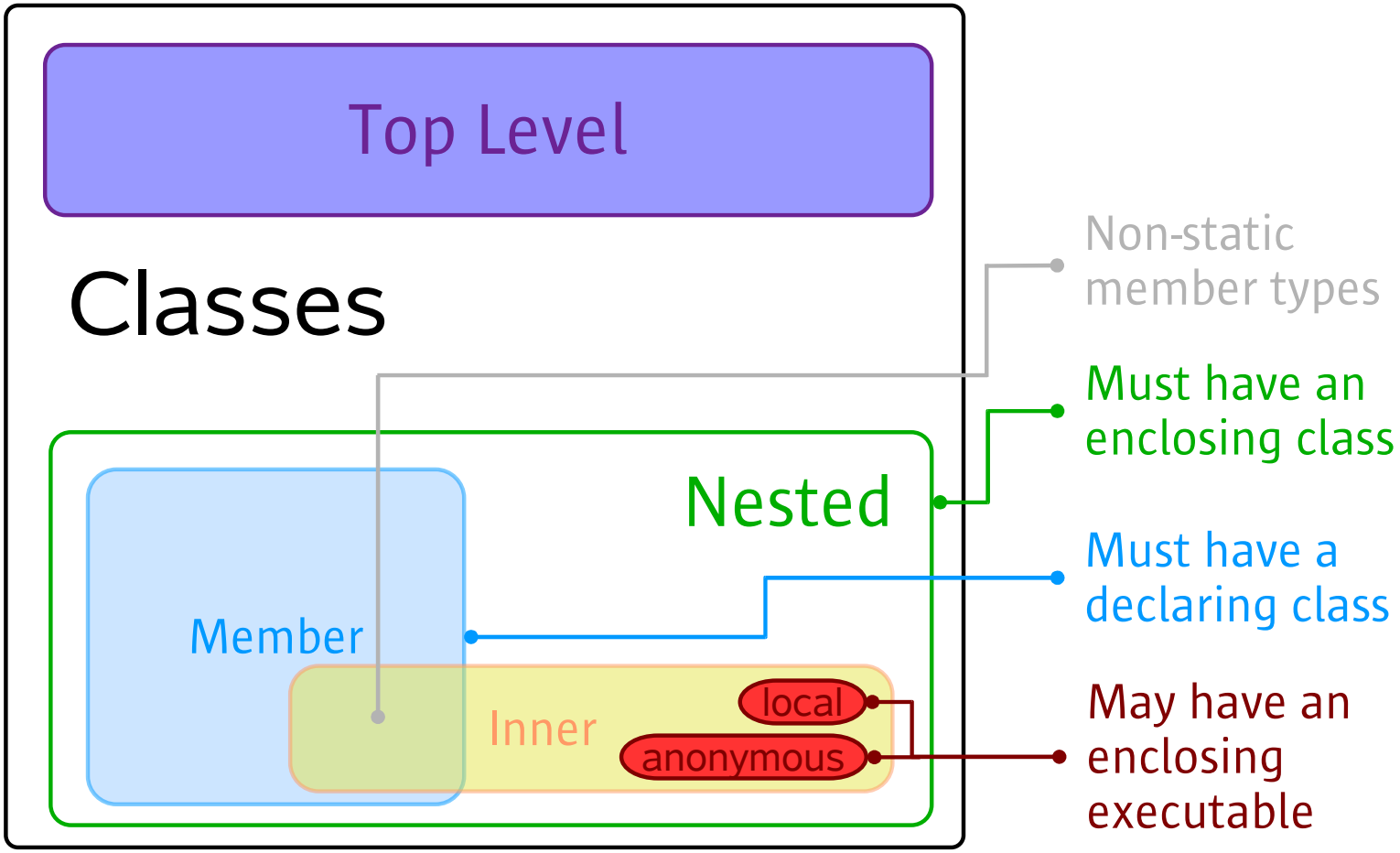
`RangeIterator` inside class `Range`

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

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

Additional advantage: keeps (nested) class simpler.

# 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.
- Inner class can refer *directly* to members of enclosing class, without qualifying the name.
- Inner classes cannot contain **`static`** members, except **`static final`** constants.

See: `NonStaticMemberClassExample`, `UnnestedNonStaticMemberClassExample`

## 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 given a name, turning them into a (named) local class.

See: `LocalClassExample`, `AnonymousClassExample`, `UnnestedLocalClassExample`; `Anonymous` (in `IteratorExamples`)

## 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

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

## Dangers of Nested Classes (and Interfaces)

- It makes the enclosing class less readable.

  So, only do this in when both classes are small.

- Enclosing class and nested class are tightly coupled.

  Harder to (re)use the nested class.

## 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 {<br>  **public** T x...<br>}<br><br>**class** B {<br>  ... A obj ...<br>  ... obj.x ...<br>} | **public class** A {<br>  **private** T x...<br><br>  **class** B {<br>  ... x ...<br>  }<br><br>} |
| 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.

## Assignments Series 3

- Read in Eck: 5.3, 5.5, 10.1.(4, 5), 10.2.1; browse 5.7
- Provide iterators for traversing the grid of a loop puzzle

## Summary

- An *iterator object* provides *iteration abstraction*:

  visit each element of a collection exactly once,

  without worrying about the underlying traversal mechanism.

- An `Iterator<T>` object provides methods `hasNext()` and `next()`.
- The for-each loop uses `iterator()` from `Iterable<T>`.