KATHOLIEKE UNIVERSITEIT LEUVEN

Department of Computer Science

Iterators & Streams

- Anonymous classes
- Streams
- Lambda Expressions

Assignment: Binary Trees

KATHOLIEKE UNIVERSITEIT LEUVEN

- Develop a method returning an iterator for binary trees and binary search trees
  - Elements will be returned in the order corresponding to a left-to-right recursive descent traversal
    - For a sorted tree, that order returns the elements in ascending order
  - The iterator will implement the interface `Iterator` of the Java API
    - The interface of binary trees will extend the interface `Iterable`
      - The enhanced for statement can then be used to iterate over the elements of a tree
- Experiment
  - Create a search tree containing some of the Fibonnaci numbers
    - The elements of the search tree will be objects of the wrapper class `Integer`
  - Compute the sum of the squares of all Fibonacci numbers in the tree that can be divided by some factor
    - The factor is read from the standard input stream

Task 1

Overview

- Static member classes
  - Definition of a class as a static member of another class
- Non-static member classes
  - Definition of a class as an instance member of another class
- Local classes
  - Definition of a class as a local element of a method
- Anonymous classes
  - Definition of a class combined with the creation of a single element as part of an expression
- Streams
  - Set up pipelines in which objects are manipulated in successive steps
- Lambda Expressions
  - Anonymous methods that implement functional interfaces

Static Member Classes

- Classes can be defined as static members of other classes
  - Static member classes have access to the private members of their enclosing classes and twin member classes
    - Enclosing classes also have access to the private members of their nested classes
  - Nested classes introduce a new scope, in which names of members of enclosing classes can be re-used for other purposes
    - A nested class may introduce a new method with the same name and the same argument list as an enclosing method

```
class EnclosingClass {

  private int x;

  protected static class NestedClass {
    private int y;
  }

}
```

Static Member Classes

KATHOLIEKE UNIVERSITEIT LEUVEN

- Outside their enclosing class, static member classes are identified by their fully qualified name
  - The fully qualified name of a nested class consists of the fully qualified name of its enclosing class, followed by its own name
    - Import statements may be used to avoid fully qualified names for nested classes
  - In the scope of its enclosing class, the fully qualified name of a nested class must not be used as long as there is no ambiguity

Task 2

Overview

KATHOLIEKE UNIVERSITEIT LEUVEN

- Static member classes
  - Definition of a class as a static member of another class
- Non-static member classes
  - Definition of a class as an instance member of another class
- Local classes
  - Definition of a class as a local element of a method
- Anonymous classes
  - Definition of a class combined with the creation of a single element as part of an expression
- Streams
  - Set up pipelines in which objects are manipulated in successive steps
- Lambda Expressions
  - Anonymous methods that implement functional interfaces

Inner Classes

- Classes can be defined as instance members of other classes
  - The definition of an inner class cannot include any static members (non-final static variables, static methods or static member classes)
    - Java suggests to introduce static members for inner classes at the level of an enclosing class
  - Classes defined as instance members of other classes are referred to as inner classes

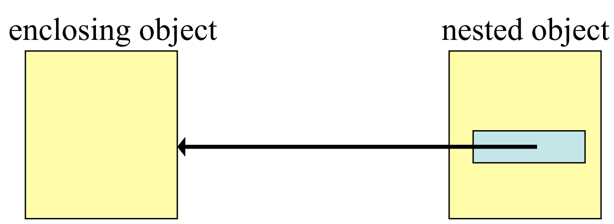
```
class EnclosingClass {

  private void f();
  public static void g();

  public class NestedClass {
    public static int nbInstances;
    public static void st() { … }
  }
}
```

Inner Classes

- Each object of an inner class is at all times associated with an object of its enclosing class
  - The association is initialized at the time the inner object is constructed, and cannot be changed afterwards
    - The association implies a final instance variable implicitly added to the definition of the inner class
  - The implicit object of the enclosing class can be denoted in an explicit way using the notation `EnclosingClassName.this`
    - The implicit object of the inner class is denoted **`this`**, as usual



Inner Classes

- Objects of an inner class are created by applying the operator **`new`** against an object of the enclosing class
  - The creation of a new inner object is denoted in its most general form as `enclosingObject.new InnerClassName(…)`
    - An unqualified creation of an inner object applies to the implicit object **`this`**

```
EnclosingClass enclosingObject = new EnclosingClass();

NestedClass nestedObject =
    enclosingObject.new NestedClass()
```

Inner Classes

- In the body of the inner class, instance methods and static methods of the enclosing class are applicable
  - Instance methods of the enclosing class can be invoked
    - in a qualified way (`EnclosingClassName.this.f()`)
    - or unqualified (`f()`)
      - Unqualified invocations are only possible if no ambiguities arise
  - Static methods of the enclosing class can be invoked
    - in a qualified way (`EnclosingClassName.g()`)
    - or unqualified (`g()`)

Inner Classes

KATHOLIEKE UNIVERSITEIT LEUVEN

```
class EnclosingClass {

  public void enclMethod();

  public static void staticMethod();

  public class NestedClass {

    public void nestedMethod() {
      enclMethod(); EnclosingClass.this.enclMethod();

      staticMethod(); EnclosingClass.staticMethod();

    }

  }

}
```

Task 3

Overview

KATHOLIEKE UNIVERSITEIT LEUVEN

- Static member classes
  - Definition of a class as a static member of another class
- Non-static member classes
  - Definition of a class as an instance member of another class
- Local classes
  - Definition of a class as a local element of a method
- Anonymous classes
  - Definition of a class combined with the creation of a single element as part of an expression
- Streams
  - Set up pipelines in which objects are manipulated in successive steps
- Lambda Expressions
  - Anonymous methods that implement functional interfaces

Local Classes

- Classes can be defined as local elements of methods of other classes
  - Local classes typically implement an interface or extend a class, whose definition is more widely accessible
- Objects of a class local to an instance method have an implicit reference to the prime object of the method
  - By definition, the referenced object is an instance of the enclosing class
  - Objects of classes local to the body of static methods do not have an implicit reference
- Local classes have access to final local variables and final formal arguments of their enclosing method
  - Since Java 8, local classes also have access to effectively final local variables or formal arguments
    - Such variables or arguments are not explicitly qualified final, but their contents does not change during the execution of the method

Local Classes

Task 4

```
class EnclosingClass {

  public void g(int x, final int y) {

    class LocalClass implements SomeInterface {

      public void nestedMethod() {
        y = 100;
        System.out.println(x);
      }

    }

    SomeInterface some = new LocalClass();
    …
  }

}
```

Overview

- Static member classes
  - Definition of a class as a static member of another class
- Non-static member classes
  - Definition of a class as an instance member of another class
- Local classes
  - Definition of a class as a local element of a method
- Anonymous classes
  - Definition of a class combined with the creation of a single element as part of an expression
- Streams
  - Set up pipelines in which objects are manipulated in successive steps
- Lambda Expressions
  - Anonymous methods that implement functional interfaces

Anonymous Classes

- The definition of a class and the construction of a single object can be combined in an anonymous class
  - Anonymous classes always implement an interface or extend an existing class
    - The definition of that interface or that superclass is typically more widely accessible
    - Because only a single object is needed, the nested class must not be given a name
- The definition of an anonymous class combined with the creation of an object is an expression
  - For interfaces, the entire construct has the general form:
    `new InterfaceName() {…}`
    - No arguments can be involved in the construction, because interfaces do not have any constructors
  - For classes, the general form is:
    `new ClassName (args) { … }"`
    - The name of the class identifies the superclass; the arguments must correspond to one of the constructors of that superclass

Anonymous Classes

```
new SomeInterface() {
  public void interfaceMethod () {
     …
  }
}


new SomeClass(arg1, arg2, …) {
  public void classMethod () {
     …
  }
}
```

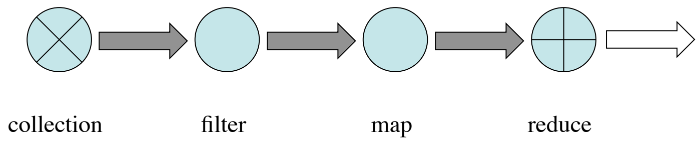
Task 5+6+7

Overview

- Static member classes
  - Definition of a class as a static member of another class
- Non-static member classes
  - Definition of a class as an instance member of another class
- Local classes
  - Definition of a class as a local element of a method
- Anonymous classes
  - Definition of a class combined with the creation of a single element as part of an expression
- Streams
  - Set up pipelines in which objects are manipulated in successive steps
- Lambda Expressions
  - Anonymous methods that implement functional interfaces

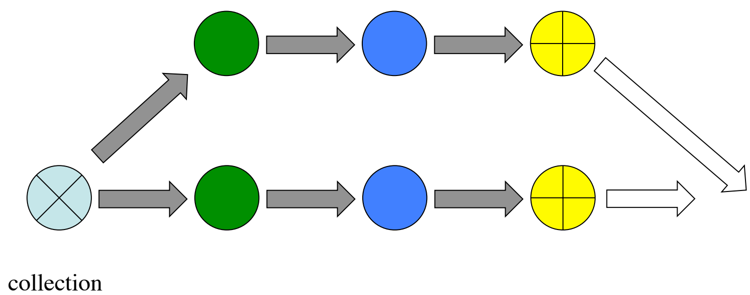
Streams

- Streams offer facilities to set up pipelines in which objects are manipulated in successive steps
  - Objects flow into the stream and are manipulated
    - Intermediate computations push their objects into another stream
    - Final computations deliver their result
  - Streams differ from collections in that they do not store their elements
    - Streams support a functional style of programming



Parallel Streams

- Parallel streams set up several pipelines in which objects flow concurrently
  - The Java Virtual Machine may use the number of cores (processors) in deciding how many pipelines to set up



Stream Methods

KATHOLIEKE UNIVERSITEIT LEUVEN

- Intermediate methods keep the stream open for further processing
  - `Stream<T> filter(Predicate<? super T > pred)`
    - Return a stream of all elements in this stream that satisfy the predicate
  - `<R> Stream <R> map (Function<? super T,? extends R> mapper)`
    - Return a stream of the results of applying the given function to all elements in this stream
- Terminal methods finish the processing of the stream
  - `boolean allMatch(Predicate<? super T> predicate)`
    - Return whether all elements in this stream satisfy the predicate
  - `void forEach(Consumer<? super T> action)`
    - Perform the action on each element in this stream
  - `Optional<T> reduce(BinaryOperator<T> accumulator)`
    - Return a reduction of the elements of this stream using the given accumulator

External versus Internal Iteration

KATHOLIEKE UNIVERSITEIT LEUVEN

- External iteration
  - The client is offered instruments to iterate over collections of objects
    - Optimizations are the client's responsibility
- Internal iteration
  - The client offers snippets of code to be executed at various stages of the computation
    - Library methods control the iteration and are able to work out optimizations such as lazy evaluation, short-circuit evaluation and parallel execution

Task 8

Overview

- Static member classes
  - Definition of a class as a static member of another class
- Non-static member classes
  - Definition of a class as an instance member of another class
- Local classes
  - Definition of a class as a local element of a method
- Anonymous classes
  - Definition of a class combined with the creation of a single element as part of an expression
- Streams
  - Set up pipelines in which objects are manipulated in successive steps
- Lambda Expressions
  - Anonymous methods that implement functional interfaces

Functional Interfaces

- Functional interfaces define exactly one abstract method
  - Lots of existing interfaces in the Java API are functional
    - Examples are `Runnable`, `Comparable<T>` and `ActionListener`
  - In Java 8, interfaces in general may in addition introduce a series of default methods
    - A default method is a method in an interface with an implementation
  - Functional interfaces may in addition override methods defined at the level of the root class `Object`
- The annotation `@FunctionalInterface` can be used to qualify interfaces as being functional
  - The annotation is not required for an interface to be functional
    - Interfaces with the annotation are checked to have only one abstract method

Standard Functional Interfaces

- The package `java.util.function` offers several predefined functional interfaces
  - `Predicate<T>` to check whether some object `t` satisfies some condition
    - Method: `boolean test(T t)`
  - `Function<T,R>` to compute some result `r` out of a given object `t`
    - Method: `R apply(T t)`
  - `Consumer<T>` to invoke a void method against a given object `t`
    - Method: `void accept(T t)`
  - `Supplier<T>` to produce some object `t`
    - Method: `T get()`
- Lots of both specialized and more general versions of the basic functional interfaces exist
  - Examples are `BiFunction<T,U,R>` and `DoubleFunction<R>`

Lambda expressions

- A lambda expression is some kind of anonymous function
  - It definition starts with a comma-separated list of formal arguments enclosed in parenthesis
    - Each formal argument involves a type and a name for the argument
  - The body of a lambda expression is either a single expression or a piece of code enclosed in curly brackets
    - An arrow (->) separates the argument list from the body
- Lambda expressions can be assigned to variables of a functional interface type
  - The lambda expression provides the implementation of the single method of the functional interface

```
(Long a, Long b) -> a+b

(Long amount)    -> { System.out.println(amount); }
```

Lambda Expressions: Shorthand

- Types of arguments may be omitted in argument lists of lambda expressions
  - The enclosing parenthesis may be omitted for lambda expressions with a single argument
- A single void statement as the body of a lambda expression must not be enclosed in curly brackets

```
(~~Long~~ a, ~~Long~~ b) -> a+b
~~(Long~~ amount~~)~~     -> ~~{~~ System.out.println(amount)~~; }~~
```

Capturing Lambda Expressions

- Lambda expressions have access to final local variables and final formal arguments of their enclosing method
  - Accessible local variables are either explicitly qualified **final** or they must be effectively final
    - Variables and arguments are effectively final if their value is not changed in the body of their method
- Lambda expressions that access non-static variables defined outside their body are said to be "capturing"
  - Non-capturing lambda expressions may be compiled in a more efficient way

Task 9