Technische Hochschule Rosenheim

Modul - Fortgeschrittene Programmierkonzepte

Bachelor Informatik

02 - Classes und Interfaces

Prof. Dr. Marcel Tilly

Fakultät für Informatik, Cloud Computing

Agenda

Classes and Interfaces Revisited

- information hiding, packages and accessibility
- interfaces revisited
- static classes
- nested classes
- lambda and method references

Information Hiding

- *Information hiding* (or *encapsulation*) is a fundamental concept in object-oriented programming.
- In Java, this is realized using
  - *interfaces* (keyword `interface`): describe the externals of modules
  - *classes* (keyword `class`): encapsulate information (variables) and business logic (methods).
- Also, grouping information and algorithms to coherent modules

Information Hiding

Technische Hochschule Rosenheim

Storing an instance of a class that *implements* (keyword `implements`) an interface in a reference

Example of *information hiding*:

```
interface Intfc {
    void method1();
}
```

```
class Klass implements Intfc {
    void method1() {
        System.out.println("Hello, World!");
    }
    void method2() {
        System.out.println("Uh, might be hidden");
    }
}
```

Information Hiding

Technische Hochschule Rosenheim

```java
Klass inst1 = new Klass();
Intfc inst2 = new Klass();  // ok-- since Klass implements Intfc

inst1.method1();
inst2.method1();  // ok-- method guaranteed

inst1.method2();  // ok-- reference of type Klass
inst2.method2();  // Does this work?
```

Information Hiding

```java
Klass inst1 = new Klass();
Intfc inst2 = new Klass();  // ok-- since Klass implements Intfc

inst1.method1();
inst2.method1();  // ok-- method guaranteed

inst1.method2();  // ok-- reference of type Klass
inst2.method2();  // Does this work?
```

As you can see, regardless of the actual implementation, you can "see" only what's on the class or interface definition.

Packaging

In Java, you can organize your code and structure your project with modules.

- Group your classes and interfaces into coherent *modules*, the packages.
- Packages are organized in a hierarchical way, similar to a filesystem:
  - while the identifier uses `.` as a separator, each level "down" will be in the according directory.
- For example, the package `de.thro.inf.fpk` would correspond to the directory `de/thro/inf/fpk`
- Java files inside that directory need to have the preamble `package de.thro.inf.fpk` to alert the compiler of the package this class belongs to.

Visibility

Recall the [visibility modifiers that are defined in Java](#):

- `public`: visible everywhere (apply to class, attributes or methods)
- `private`: visible only within the class (apply to attributes or methods)
- `protected`: visible within the class, package **and** in derived classes (apply to attributes or methods; more next week)
- *(no modifier)*: visible within the *package*, but not visible outside of the package (apply to class, attributes or methods)

Visibility and Packaging

Both of these features combined yield excellent information hiding:

```java
package de.thro.inf.fpk;
public interface Itfc {
    void method();
}
```

```java
package de.thro.inf.fpk;
class SecretImpl implements Itfc {
    public void method() {  // note: interface --> public
        System.out.println("Hello, World!");
    }
    void secret() {
        System.out.println("Only accessible within this package!");
    }
}
```

Scope?

```java
package de.thro.wif.oop;      // note: different package...
import de.thro.inf.fpk.Itfc;  // ...thus must import!

Itfc itfc = ...;   // we'll come to this later!
itfc.method();

// de.thro.inf.fpk.SecretImpl not visible
// only methods of .Itfc are accessible
```

Interfaces I

- Prior to Java 8, interfaces were limited to (public) functions.
- Since Java 8, interfaces can provide `default` implementations for methods (used to maintain backwards compatibility on extended interfaces) which are available on every resource, and can implement `static` methods, which can be used without instances.

Reconsider the above code example:

```java
package de.fhro.inf.prg3;
public interface Itfc {
    void method();
    static Itfc makeInstance() {
        return new SecretImpl();
    }
    default void method2() {
        System.out.println("Ah, seems not implemented!");
    }
}
```

Interfaces II

```java
package de.fhro.wif.prg3;
import de.fhro.inf.prg3.Itfc;

Itfc itfc = Itfc.makeInstance();
itfc.method();    // provided by (hidden) SecretImpl
itfc.method2();   // provided by default implementation
```

Use `static` on interface methods just like you would on class methods. Use `default` to provide a default implementation, which can be overridden by the implementing class.

> Note: Depending on your JVM security settings, you can use reflection to get around information hiding and to inspect the actual class of the instance; we'll cover that later in this class.

Interface II

Since Java 9, you can use the `private` keyword to implement regular and static helper functions

```
interface Itfc {
    default void a() {
        System.out.println("Hello, I'm (a)");
        c();
    }
    private void c() {
        System.out.println("Yay, (c) only implemented once!");
    }
    // same for static
    static void d() {
        System.out.println("Hello, I'm (d)");
        f();
    }
    private static void f() {
        System.out.println("Yay, (f) only implemented once!");
    }
}
```

Name Conflicts

Since in Java classes can implement multiple interfaces, you may end up with a name conflict:

```
interface Itfc1 {
    default void greet() { System.out.println("Servus"); }
}
interface Itfc2 {
    default void greet() { System.out.println("Moin"); }
}

// must implement greet() to resolve name conflict
class Example implements Itfc1, Itfc2 {
    public void greet() {
        Itfc2.super.greet();  // use super to specify which implementation
    }
}
```

Classes and `static`

- Recall the `static` modifier, used inside class definitions.
- In the following example, all instances of class `Klass` share the very same `n`; this variable "lives" with the class definition.
- Each instance will have its own `m`, since it is *not* static.

```
class Klass {
    private static n = 0;
    static int nextInt() {
        return n++;
    }
    private int m = 0;
    void update() {
        m = nextInt();
    }
}
```

Classes and `static`

Calling `nextInt()` anywhere will return the current value and then increment by one. The `update()` method can only be called on instances, but will use the very same `nextInt`.

```
int n1 = Klass.nextInt();  // n1 == 0, Klass.n == 1
Klass k = new Klass();
k.update();                // k.m == 1, Klass.n == 2
int n2 = k.nextInt();      // n2 == 2, Klass.n == 3
```

Note that static attributes and methods can be called from both the class and the instance. To avoid misunderstandings, use the class when accessing static members.

Typical use cases for static members are

- constants,
- shared counters,
- or the Singleton pattern.

Static Initializers

Static attributes are typically immediately initialized (particularly if they're `final`).

If the value is not just a simple expression, you can use a *static initializer block* `static { /* ... */ }` to do the work:

```
class Klass {
    static final int val;

    static {
        // do what you like...
        val = Math.sqrt(3.0);
    }
}
```

Inner Classes

Consider the following example of a simple binary tree: every node has a left and a right child; the tree is defined by its root node.

- The class that represents the node is very specific to the `BinaryTree` (and presumably not useful to other classes), thus its is a good candidate of an inner class:

```
class BinaryTree {
    private class Node {
        Object item;
        Node left, right;
    }
    Node root;
}
```

Inner classes can have accessibility modifiers (`private`, `protected`, `public`), and are defined within the enclosing class's `{}` (the order of attributes is irrelevant).

> Inner classes are also compiled, and stored as `BinaryTree$Node.class`.

Inner Classes

- All attributes and methods of the outer class are available to the inner class -- regardless of their accessibility level!
- This is also the reason that instances of (regular) inner classes can only exist with an instance of the corresponding outer class.
- Potential shadowing of variables by inner class can be resolved by using the class name:

```
class Outer {
    int a;
    class Inner {
        int a;
        void m() {
            System.out.println(a);  // Outer.Inner.a
            System.out.println(Outer.this.a);
        }
    }
}
```

Static Inner Classes

Like other members, inner classes can also be `static`; in this case, the inner class can be used without an instance of the enclosing class:

```
class Outer {
    static class StaticInner {

    }
    class Inner {

    }
}

Outer.StaticInner osi = new Outer.StaticInner();  // ok
```

To instantiate the inner class outside of the outer class, instantiate the outer class first:

```
Outer.Inner oi = new Outer.Inner();  // error: must have enclosing instance
Outer.Inner oi = new Outer().new Inner();
```

Anonymous Classes

Far more often, you will be using anonymous inner classes.

Recall the sorting function `java.util.Collections.sort(List<T> list, Comparator<? super T> c)` (ignore the `<...>` for now).

You might have used this as follows:

```
class MyComparator implements Comparator {
    public int compareTo(Object o1, Object o2) {
        // ...
    }
}
```

```
Collections.sort(mylist, new MyComparator());
```

Anonymous Classes

While this works just fine, you have one more extra class, just to carry the actual comparison code.

Anonymous classes help keeping your class hierarchy clutter-free:

```
Collections.sort(mylist, new Comparator() {
    public int compareTo(Object o1, Object o2) {
        // ...
    }
});
```

Note that it says `new Comparator() {}`: While it is true that you cannot instantiate interfaces, this syntax is shorthand for creating a new class that implements the `Comparator` interface. This is also works for an anonymous derived class (`new Klass() {}`).

Anonymous Classes

The syntax is compelling, but comes with one major drawback: anonymous classes cannot have a constructor. Instead, Java replicates the current scope, that is: all *effectively* final variables are available within the class.

```
final Object ref;
Collections.sort(mylist, new Comparator() {
    {
        System.out.println(ref);  // anonymous initializer block
    }
    public int compareTo(Object o1, Object p2) {
        if (o1.equals(ref))
            // ...
    }
})
```

Similar to the static initializer block (`static {}`), you can use an anonymous initializer block (`{}`).

Local Classes

The last variant is the *local class*, which is essentially the same as an anonymous inner class, but can be defined with a constructor.

```
class Klass {
    void example() {
        class Local {
            int m;
            Local(int m) {
                this.m = m;
            }
        }

        Local l1 = new Local(3);
    }
}
```

> Note that the enclosing class can again be referenced as `Klass.this. ...`.

Functional Interfaces

A *functional Interface* is an interface that has exactly one non-default method and is annotated with `@FunctionalInterface` (since Java 8).

```
@FunctionalInterface
interface Filter {
    boolean test(Object o);
}
```

```
class Klass {
    void filter(Filter f) {
        // f.test(...)
}
```

```
Klass k1 = new Klass();
k1.filter(new Filter() {
    public boolean test(Object o) {
        return o != null;
    }
});
```

Lambda Expression

There is a lot of "boilerplate" code beside the actual `test()` function.

You can write this more compact with a lambda expression:

```
k1.filter(o -> o != null);  // single statement
k1.filter(o -> {  // multiple statements, conclude with return
    return o != null;
})
```

- the lambda expression `x -> ...` refers to a functional Interface, with the non-default function having exactly one argument.
- if you have multiple arguments, the lambda expression becomes for example `(a1, h2) -> ...` (note that the type is inferred automatically).

Method Reference

The third alternative is to use a method *reference*:

```
@FunctionalInterface
interface Filter {
    boolean test(Object o);
    static boolean testForNull(Object o) {
        return o != null;
    }
}
```

```
Filter fi = new Filter() {
    public boolean  test(Object o) {
        return o != null;
    }
}
```

```
k1.filter(fi);
k1.filter(fi::test);
k1.filter(Filter::testForNull);
```

Method References

Method reference is a shorthand notation of a lambda expression to call a method!

Method references (`::`) can be specified in the following ways:

Kind	Example
Reference to a static method	`ContainingClass::staticMethodName`
Reference to an instance method of a particular object	`containingObject::instanceMethodName`
Reference to an instance method of an arbitrary object of a particular type	`ContainingType::methodName`
Reference to a constructor	`ClassName::new`

and their usage can be confusing.

Homework

Does this work?

```
@FunctionalInterface
interface BiFunction {
    Object apply(Object a, Object b);
}
```

```
class SomeObject implements BiFunction {
    public Object apply(Object o) {
        System.out.println(o);
        return null;
    }
    public static void main(String[] args) {
        SomeObject so = new SomeObject();
        BiFunction bf = so::apply;
    }
}
```

Final Thought!