(2) Java Abstractions: More on User defined Types

Nico Ludwig (@ersatzteilchen)

TOC

- (2) Java Abstractions: More in User defined Types
  - Documenting Classes
  - Instance- vs. Class Members
  - Enumerations revised
  - Deprecation
  - Blank Finals
  - Static nested Classes

- Cited Literature:
  - Just Java, Peter van der Linden
  - Thinking in Java, Bruce Eckel

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

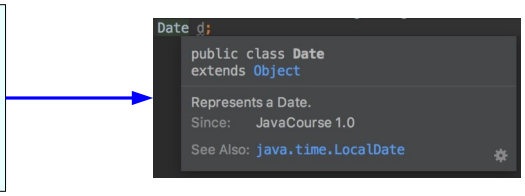
Have fun!

Documenting Java Classes with Javadoc

- When we discussed methods, we learned, that we can document them via Javadoc comments.
  - We can also document classes with Javadoc!

- As programmers we can help Javadoc to generate more detailed documentations by providing special comments:

```
/**
 * Represents a Date.
 * @author      Nico Ludwig
 * @see         java.time.LocalDate
 * @since       JavaCourse 1.0
 */
public class Date {
      // pass
}
```



  - Javadoc comments are fringed with /** */ (not /* */) and allow the usage of @-prefixed tags for special documentation aspects.
    - The first lines of a Javadoc comment can be used for a free prosaic text "Represents a Date.". The following lines can be used for the tags.
    - Important Javadoc (block-)tags for classes: @author, @see and @since.
    - It is also possible to embed HTML 4 markup to style the output documentation (e.g.).
  - The result of these comments can also be used in the IDE directly, without generating HTML pages explicitly.
    - Thus we can use the Javadoc comments to have an inline documentation during development in the IDE.

- We should avoid adding too many @-keywords and embedded markup to make the generated documentation look awesome. Why? Well, because the comment itself should remain readable as well!
- In the example it is shown, how IntelliJ IDEA shows the documentation of a method from the Javadoc comment. This view can be triggered by ctrl+Q on Windows or ctrl+J on OS X.

Instance independent Functionality

- Let's create another method for the UDT *Date*, which retrieves a *Date* object representing "today":

```
// <Date.java>
import java.time.*;

public class Date { // (members hidden)
        public Date today() {
            LocalDate now = LocalDate.now();
            return new Date(now.getDayOfMonth(), now.getMonthValue(), now.getYear());
        }
}
```

  - We use *LocalDate.now()* to get the current date. *LocalDate* is defined in the package *java.time*. It can be qualified as *java.time.LocalTime*.

- The new method *Date.today()* is used like so:

```
Date date = new Date();
Date today = date.today();
today.print();
// >20.12.2012
```

- Fair enough, it works! But it is nasty!
  - It makes no sense to create an "empty" *Date* instance and then call a method like *Date.today()* on that "empty" instance.
  - "Empty" means, that the immediate object *date* is superfluous in the example above!

- We can do better in Java!

Instance independent Functionality with static Methods

- In hindsight, *Date.today()* has two remarkable features:
  - It is independent from a *Date*-instance. I.e. no *Date*-instance is really required to make *Date.today()* work.
  - Nevertheless, it is somehow connected to the UDT *Date*. E.g. it creates an instance of type *Date*.

- If a method is independent of a certain instance, but connected to the UDT in question, it can be defined as static method.
  - A method is independent of a certain instance, if it never references this implicitly or explicitly.

```
// <Date.java>
public class Date { // (members hidden)
        public static Date today() {
            java.time.LocalDate now = java.time.LocalDate.now();
            return new Date(now.getDayOfMonth(), now.getMonthValue(), now.getYear());
        }
}
```

- We can call a static method on the class name, i.e. using '.' to call the method on the class name directly:

```
Date today = Date.today();
today.print();
// >20.12.2012
```

  - Mind, this is exactly the way we have called static methods defined in *Program* from other classes.
  - No instance of *Date* must be created to call *Date.today()*!
  - Therefore static methods are also called class methods (in opposite to non-static instance methods).

Static Methods in Detail – Part 1

- We can bind methods to a UDT instead of an instance: static methods.

```
// <Date.java>
public class Date { // (members hidden)
        public static Date today() {
            java.time.LocalDate now = java.time.LocalDate.now();
            return new Date(now.getDayOfMonth(), now.getMonthValue(), now.getYear());
        }
}
```

```
Date today = Date.today();
today.print();
// >20.12.2012
```

- The static method *Date.today()* has following features:
  - (1) it's independent of a concrete *Date* instance (no *Date* object is needed to call *Date.today()* and *Date.today()* doesn't refer this),
  - (2) because it is public, it can be called "from outside" the class.

- Syntactic peculiarities of static methods:
  - In the definition of a static method, the keyword static is used.
  - static methods are called with the . operator applied on the type name.
  - static methods can also be called on instances of the defining type like non-static methods, but this is discouraged.
    - They can even be called on null-references of the defining type.
  - Java does not allow to have a static and a non-static method with the same name and parameter-set in the same UDT.

```
Date myDate = new Date();
// discouraged: calls static method on instance:
Date today = myDate.today();
```

Static Methods in Detail – Part 2

- Common misunderstandings while using static methods:
  - We can't have a this-reference in static methods, because there's no instance.
  - Remember: in non-static methods, the implicit parameter "this" refers to the current instance, which doesn't exist for static methods.

- Just inspect this example:

```
// <Date.java>
public class Date { // (members hidden)
        private int day;

        public int setDay(int day) {
            this.day = day;
        }

        public static Date today() {
            int theDay = this.day; // (1) Invalid! We cannot access this.day from a static context! There is no "this"!
            setDay(21); // (2) Invalid! We cannot access setDay() from a static context! Here the "this" is just implicit!

            java.time.LocalDate now = java.time.LocalDate.now();
            return new Date(now.getDayOfMonth(), now.getMonthValue(), now.getYear());
        }
}
```

Static Fields

- We can also define static fields in a UDT.
- Assume, we want to count how often the static method *Date.today()* is called. We can use a static field as counter:

```
// <Date.java>
public class Date { // (members hidden)
        public static int nTodayCalls;

        public static Date today() {
             ++nTodayCalls;
             java.time.LocalDate now = java.time.LocalDate.now();
             return new Date(now.getDayOfMonth(), now.getMonthValue(), now.getYear());
        }
}
```

```
Date today = Date.today();
Date today2 = Date.today();
System.out.printf("Date.today() was called %d times.%n", Date.nTodayCalls );
// >Date.today() was called 2 times.
```

- Basically, static fields share the syntactic peculiarities of static methods:
  - In the definition of a static field, the keyword static is used. Like other fields, they default to their default value (0/0.0, null or false).
  - static fields are accessed with the . operator applied on the type name.
  - static fields can also be accessed on instances of the defining type like non-static fields, but this is discouraged: we'll discuss this in a minute.
    - They can even be accesses on null-references of the defining type.
  - Java does not allow to have a static and a non-static field with the same name in the same UDT.

Static Fields are shared among Instances

- Because the static field *Date.nTodayCalls* is public, we can easily assign a value to it:

```
Date.nTodayCalls = 20;
```

- Let's create two *Date* instances, *date* and *birthday* and read the value of the field *nTodayCalls* "from" those instances:

```
Date date = new Date();
Date birthday = new Date();
System.out.printf("Date.today() was called %d times and %d times.%n", date.nTodayCalls, birthday.nTodayCalls);
// >Date.today() was called 20 times and 20 times.
```

  - *date.nTodayCalls*, *birthday.nTodayCalls* both evaluate to 20, because they share the same static field *Date.nTodayCalls*.

- Then we set "only" *birthday.nTodayCalls* to 5 and print out the results to the console once again:

```
birthday.nTodayCalls = 5;
System.out.printf("Date.today() was called %d times and %d times.%n", date.nTodayCalls, birthday.nTodayCalls);
// >Date.today() was called 5 times and 5 times.
```

  - *date.nTodayCalls*, *birthday.nTodayCalls* both evaluate to 5, because they share the same static field *Date.nTodayCalls*.

- However, if we evaluate *Date.nTodayCalls*, we see, that it evaluates to 5 as well!

```
System.out.printf("Date.today() was called %d times.%n", Date.nTodayCalls);
// >Date.today() was called 5 times and 5 times.
```

  - Of course! Because *Date.nTodayCalls*, *date.nTodayCalls* and *birthday.nTodayCalls* all represent the very same static field!

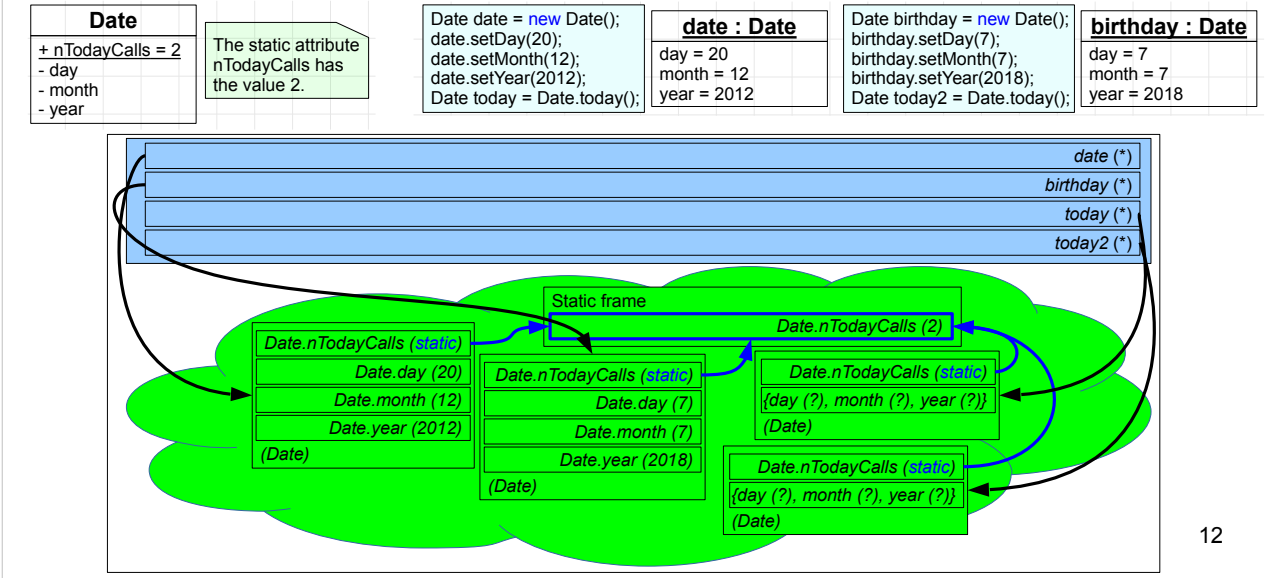
- => It is misleading, discouraged, and dangerous to access static fields via variables instead of class names.

Static Fields in Memory – Part 1

- static members at run time:
  - static fields are held in a so called static frame, which is allocated in the heap!
  - static fields are directly initialized, when the VM loads the enclosing class.
  - I.e. the ctors of objects created as static fields are executed, when the VM loads the enclosing class!

- Mind, that static fields are shared among all instances, we'll inspect this in another memory diagram.

Static Fields in Memory – Part 2



- Questions about when a static field is ending its lifetime are difficult to answer. The memory in question could be gc'd, when the enclosing class is unloaded from memory, but this aspect is heavily dependent on the used "class loader".

Encapsulating static Fields

- To make the static field *Date.nTodayCalls* a little safer, we could encapsulate modification via a clever setter.
  - So we can least avoid, that it is set to a negative value:

```
// <Date.java>
public class Date { // (members hidden)
        private static int nTodayCalls;

        public static void setNTodayCalls(int nTodayCalls) {
            if (0 <= nTodayCalls) {
                    Date.nTodayCalls = nTodayCalls; // Explicit access to shadowed Date.nTodayCalls.
            }
        }

        public static int getNTodayCalls() {
            return Date.nTodayCalls;
        }

        public static Date today() {
            setNTodayCalls(getNTodayCalls() + 1);
            java.time.LocalDate now = java.time.LocalDate.now();
            return new Date(now.getDayOfMonth(), now.getMonthValue(), now.getYear());
        }
}
```

  - As can be seen, we can also have private static fields (and methods).

- Also mind that local variables and parameters of static methods can also shadow static fields!
  - The shadowed static field is reached by using the class-name (i.e. *Date*) as prefix.

public final static Fields

- static public fields are as dubious as non-static public fields are, therefor they're used rarely in Java.
  - Also mind, that their sharing behavior is error prone, although it can be useful!

- public static constants, i.e. public final static fields are far more common in Java!

```
public class Date { // (members hidden)
        // seconds per Gregorian year:
        public static final double S_PER_GREGORIAN_YEAR = 31556952.2;
}

System.out.println(Date.S_PER_GREGORIAN_YEAR);
// >3.15569522E7
```

  - *Date.S_PER_GREGORIAN_YEAR* as public static final makes sense: it is *Date*-related, but not related to a specific *Date* instance!
  - Because final fields cannot be modified, their sharing behavior and public access is less a problem.

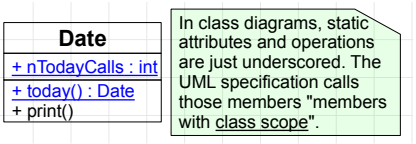
- Actually, we have already used a public static final field all the time: *System.out*!

```
// Somewhere in the JDK, the class System is defined:
public class System { // (declaration simplified)
        public final static PrintStream out;
}
System.out.println("Hello World!");
```

  - So, the public static final field *System.out* is an instance of type *PrintStream*, and we're calling its public method *println()*.

static Members vs. Non-static Members – Part 1

- Some definitions of "being static":
  - A static member belongs to a UDT, but does not belong to a certain instance.
  - static members are shared among all instances of a UDT.



- Interesting facts:
  - static methods and final static fields are used often in Java to implement compile time constants!
  - static methods have access to private fields of the defining UDT!
  - static methods do not overload non-static member functions!

- Terms:
  - Sometimes static members are called class or type members.
  - Sometimes Non-static members (i.e. "normal" members) are called instance members.
    - Each instance has its own copy of, e.g., a non-static field as instance member.

- The idea of static vs. non-static members is handy for useful abstractions.
  - We can find these concepts in basically all programming languages allowing to define abstractions.

15

- A static method can access the private methods (esp. the fields) of a passed object that has the same UDT, in which this static method has been defined. Nevertheless, a static method has no this-reference!

static Members vs. Non-static Members – Part 2

- We cannot access non-static members from a static context.

```
// <Date.java>
public class Date { // (members hidden)
        private int day;

        public static Date today() {
            int theDay = this.day; // Invalid! We can access setDay() from a static context! There is no "this"!

            java.time.LocalDate now = java.time.LocalDate.now();
            return new Date(now.getDayOfMonth(), now.getMonthValue(), now.getYear());
        }
}
```

- But we can access static members from non-static contexts, also private ones.

```
// <Date.java>
public class Date { // (members hidden)
        private int day;

        public int setDay(int day) {
            this.day = day;
            System.out.printf("Log today: %s%n", today());
        }

        public static Date today() {
            // pass
        }
}
```

Revisiting Enumerations – Part 1

- In Java, it's common sense to represent constant values in a class as static final fields of instances of the enclosing class:

```
// <Month.java>
public class Month { // Month fields:
        public static final Month JANUARY = new Month();
        public static final Month FEBRUARY = new Month();
        public static final Month MARCH = new Month();
        public static final Month APRIL = new Month();
        public static final Month MAY = new Month();
        public static final Month JUNE = new Month();
        public static final Month JULY = new Month();
        public static final Month AUGUST = new Month();
        public static final Month SEPTEMBER = new Month();
        public static final Month OCTOBER = new Month();
        public static final Month NOVEMBER = new Month();
        public static final Month DECEMBER = new Month();
}
```

  - Actually, it makes a lot of sense, because *Month.JANUARY* is indeed an object of type *Month* and also a constant one.

- Usage of *Month* is very simple:

```
// myBirthMonth is of type Month:
Month myBirthMonth = Month.OCTOBER;
// Invalid! Can't assign to a final static field:
Month.JANUARY = Month.DECEMBER;
```

Revisiting Enumerations – Part 2

- In an earlier lesson we mentioned enums, enums are basically shortcuts for classes full of "own" constant objects.

```
// <Month.java>
public class Month { // Month fields:
        public static final Month JANUARY = new Month();
        public static final Month FEBRUARY = new Month();
        public static final Month MARCH = new Month();
        public static final Month APRIL = new Month();
        public static final Month MAY = new Month();
        public static final Month JUNE = new Month();
        public static final Month JULY = new Month();
        public static final Month AUGUST = new Month();
        public static final Month SEPTEMBER = new Month();
        public static final Month OCTOBER = new Month();
        public static final Month NOVEMBER = new Month();
        public static final Month DECEMBER = new Month();
}
```

```
// <Month.java>
public enum Month { // Month's enum constants:
        JANUARY,
        FEBRUARY,
        MARCH,
        APRIL,
        MAY,
        JUNE,
        JULY,
        AUGUST,
        SEPTEMBER,
        OCTOBER,
        NOVEMBER,
        DECEMBER
}
```

- Usage of the enum *Month* is also very simple:

```
// myBirthMonth is of type Month:
Month myBirthMonth = Month.OCTOBER;
// Invalid! Can't assign to a enum constant:
Month.JANUARY = Month.DECEMBER;
```

  - The fields of an enum of often just called enum constants, they are guaranteed to be different from each other and unique.

- From our current understanding an enum is like a class, but it has only public static final fields.
  - And each field is just an instance of the enclosing enum, i.e. *Month.JANUARY* is an instance of the enum *Month*.
  - In the enum, we just leave away the definition of public static final *Month*, that we would need for fields otherwise.

Revisiting Enumerations – Part 3

```
// <Month.java>
public enum Month { // Month's enum constants:
        JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY,
        AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER
}
```

- enums are closed, i.e. we cannot create new *Month*-instances "outside" from *Month*:

```
Month anotherMonth = new Month(); // Invalid! enum types may not be instantiated
```

- enums can also be used in switch statements and switch expressions!
  - This is an excellent usage of switch, because a compiler could warn, if we missed any constant in switch of the actually closed enum type!
    - Using switch with "free constants" is much more error prone!
  - A syntactic peculiarity is, that constants must be unqualified in switch (i.e. we cannot write *Month.MAY*, but instead must write *MAY*).
  - The closeness of *Month* makes the only possible default case the null case!
    - Sure, this is only true, if we handled all enum constants of an enum in switch.
    - (If enum defined value types, handling null would not be required!)

```
static String monthString(Month month) {
        switch (month) {
                case JANUARY: return "January";
                case FEBRUARY: return "February";
                case MARCH: return "March";
                case APRIL: return "April";
                case MAY: return "May";
                case JUNE: return "June";
                case JULY: return "July";
                case AUGUST: return "August";
                case SEPTEMBER: return "September";
                case OCTOBER: return "October";
                case NOVEMBER: return "November";
                case DECEMBER: return "December";
                default: return "null";
        }
}
```

Revisiting Enumerations – Part 4

```
// <Month.java>
public enum Month { // Month's enum constants:
        JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY,
        AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER
}
```

- enums are a simple way to put an idea of powerful type systems into effect: enums make illegal states unrepresentable.
  - A *Month* representing a value beyond 1, 12 cannot be created at all and *Month* is closed in its type.

- But some useful features are not provided by the enum *Month*, for example:
  - A numeric value of a certain enum constant would be good: *Month.JANUARY* to 1, *Month.FEBRUARY* to 2 etc.
  - A nice textual representation of a certain enum constant: *Month.JANUARY* to "January", *Month.FEBRUARY* to "February" etc.
    - The method *monthString(Month month)* introduced on the last slide implemented this with switch.

- An interesting point about enums is, that we can add methods and even ctors to them, which makes them very mighty!
  - I.e. we can add the missing features ourselves!

- Let's see, how we can improve *Month*!

Revisiting Enumerations – Part 5

- Well, we've quite a lot to do, to make *Month* more useful!
  - (1) Put the definition into a separate java-file and make it public.
  - (2) Add a ctor accepting a numeric value as well as a more useful name as *String*. The ctor is implicitly public.
  - (3) Add two private fields to store the numeric value as well as the name of a specific *Month*.
  - (4) Add public getters to retrieve the numeric value as well as the useful name of a specific *Month*.

```
// <Month.java>
public enum Month {
        JANUARY(1, "January"), FEBRUARY(2, "February")
          // pass
        ; // The semicolon is required!

        private int numericValue;
        private String name;

        Month(int numericValue, String name) {
            this.numericValue = numericValue;
            this.name = name;
        }

        public int getNumericValue() {
            return numericValue;
        }

        public String getName() {
            return name;
        }
}
```

- Esp. with the new ctor, which is the only ctor now, we have to create the enum constants calling this ctor!
  - The syntax looks weird, esp. it is required to list the constants as first members in the enum. And it is also required to terminate this list with a semicolon!

- Using these new features should be self explanatory:

```
Month theMonth = Month.JANUARY;
System.out.println("Numeric month value: "+theMonth.getNumericValue());
// >Numeric month value: 1
System.out.println("Handsome month name: "+theMonth.getName());
// >Handsome month name: January
```

```
// We can now reformulate monthString() w/o using switch:
static String monthString(Month month) {
        return null != month ? month.getName() : "null";
}
```

Revisiting Enumerations – Part 6

- enums can be compared for relative order with the method *compareTo()*, e.g. we can use *Month.compareTo()*:

```
// <Month.java>
public enum Month { // Month's enum constants:
        JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY,
        AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER
}
```

Good to know
Fair enough, Java already provides the enum *Month* in the package *java.time*. *java.time.Month* has more features than our implementation.

  - Relative order means the order in which the enum constants are defined in the enum definition:

```
boolean marchBeforeMay = Month.MARCH.compareTo(Month.MAY) < 0;
// marchBeforeMay = true
```

  - This implicitly defined relative order in enums makes enums sortable!

- enums also provide a *toString()* method, which just returns the name of the referenced enum constant as *String*:

```
String octoberString = Month.OCTOBER.toString();
// octoberString = "OCTOBER"
```

- All enums implicitly define the static method *values()*. We can use for each to write all *Month*-value-*String*s to the console:

```
for (Month month : Month.values()) {
        System.out.print(month.toString() + ", ");
}
```

Terminal

```
NicosMBP:src nico$ java Program
JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER,
NicosMBP:src nico$
```

Deprecation – Part 1

- After a while we notice that almost nobody uses *Date.setDay()*, *Date.setMonth()* and *Date.setYear()*. – Instead the ctor is used.
  - So we decide to comment these methods, then we do not need to maintain them in future. Hence, *Date* is an unmodifiable type!

```
public class Date { // (other members omitted)
        public Date(int day, int month, int year) {
            this.day = day;
            this.month = month;
            this.year = year;
        }
        public void setDay(int day) {
            this.day = day;
        }
        public void setMonth(int month) {
            this.month = month;
        }
        public void setYear(int year) {
            this.year = year;
        }
}
```



```
public class Date { // (other members omitted)
        public Date(int day, int month, int year) {
            this.day = day;
            this.month = month;
            this.year = year;
        }
        //public void setDay(int day) {
        //    this.day = day;
        //}
        //public void setMonth(int month) {
        //    this.month = month;
        //}
        //public void setYear(int year) {
        //    this.year = year;
        //}
}
```

  - In future, only the ctor of *Date* should be used to rather create a new *Date* object instead of modifying an existing *Date* object!

- But there is a problem: there is still code written by other people, which makes use of any of these methods!
  - We have broken their code!

```
Date myDate = new Date(17, 10, 2012); // Construct the Date.
myDate.setMonth(12); // Invalid! Cannot resolve method 'setMonth(int)'
```

- Java provides a better mechanism to deal with such situations: deprecation.

Deprecation – Part 2

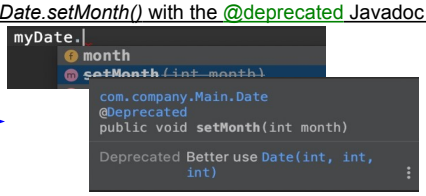
- A method, which should no longer be used by other programmers can be marked as being deprecated.
  - This is done by writing the *@Deprecated* annotation in front of the definition of the method in question.

```
public class Date { // (members hidden)
        @Deprecated
        public void setMonth(int month) {
            this.month = month;
        }
}
```

```
Date myDate = new Date(17, 10, 2012); // Construct the Date.
myDate.setMonth(12); // Ok, but issues a warning! 'setMonth(int)' is deprecated
```

  - We will discuss annotations in a future lecture.
  - We can also deprecate fields as well as complete UDTs with the *@Deprecated* annotation!
  - If the compiler finds code, which uses a deprecated method, it will issue a warning.
  - But even if deprecated, it will still be called at run time! -> It is just a compiler warning!
- But, in order to be a "fair UDT provider", we should tell other programmers an alternative to using *Date.setMonth()*!
  - Therefor we add an extra documentation in the Javadoc comment of *Date.setMonth()* with the @deprecated Javadoc tag:

```
/**
 * @deprecated Better use {@link Date#Date(int, int, int)}
 */
@Deprecated
public void setMonth(int month) {
        this.month = month;
}
```



  - Notice, that we have embedded the @link Javadoc tag to link to another documentation, here that of the preferred ctor.

- There is a vivid discussion, whether methods or UDTs (in sum APIs) being marked *@Deprecated* should be removed in a future version of an API. The problem is, that APIs were often used many times before those were *@Deprecated*, and it can be downright too costly to refactor that code to using alternatives. For that reason the designers of the JDK are very reluctant to remove *@Deprecated* APIs in future versions of the JDK. Example: several methods in the UDT *java.util.Date*. Since Java 9, deprecation can be specified more accurately, using *@Deprecated(forRemoval=[true|false], since="<version>")*. In newer Java versions (~ Java 10), some *@Deprecated* APIs were actually removed!
- It should also be said, that Java's way avoiding to remove deprecated APIs is also a part its success as "stable platform".

Blank finals Fields

- On the last slides, we discussed how to make *Date* an unmodifiable type.
  - It basically means, that *Date* instances don't allow callers to modify their state, i.e. there are no setters.
  - We already marked the setters as *@Deprecated*, but *Date*'s fields could still be modified from within *Date*'s code!

- We'll now "stabilize" *Date* being unmodifiable, by making its fields unmodifiable as well.
  - We can make *Date*'s fields *day*, *month* and *year* final. Then they become run time constants!
  - But Java forces the initialization of all final fields in all ctors of the UDT. I.e. all ctors of *Date* must initialize *day*, *month* and *year*!
    - The initialization of final fields is just performed via assignment in the ctors.
    - Mind, that these are final non-static fields!

```
public class Date {  // (other members omitted)
    private final int day;
    private final int month;
    private final int year;

    // All ctors need to initialize all final fields:
    public Date(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }
    @Deprecated
    public void setMonth(int month) {
        this.month = month;
    }
}
```

will no longer compile:
"Cannot assign value to final variable 'month'"

- In Java, final fields, whose initialization is deferred into the ctors are called blank finals.

- After we made the fields final, the next step will be removing the modifying methods, because they do not longer compile!

Example: Blank Finals in the String Class

- The *String* class uses a blank final char[] to store the *String* value, with which it was initialized.

```
public class String { // Simplified implementation of String
        // The blank final value is used for character storage:
        private final char value[];

        // Initializes the blank final value
        public String(String original) {
            this.value = original.value;
        }
        // ...
}
```

  - As can be seen, the ctor initializes *value* from the original *String* argument.
  - This makes totally sense for *String*, because *String* objects are guaranteed to be immutable in Java, so must its fields.
    - *String* doesn't provide any methods to modify its content.

- Each ctor must directly or indirectly (by calling other ctors via this()) initialize all blank finals of the enclosing class.
  - When we inspect the implementation of *String* we'll also see that all ctors initialize the field *value*.

- Using blank finals is a good style in general.

Nested Classes

- Java allows to define classes, which are defined in a definition of an other class, those are called nested classes.

- Nested classes fall in two categories:
  - (1) static nested classes and
  - (2) non-static nested classes (also called inner classes).
  - => Now, we will discuss static nested classes.

- E.g. lets make *Date* a static nested class of the class *Program* in file Program.java:

```
// <Program.java>
public class Program {
        static class Date { // (members hidden)
                public static Date today() { /*pass*/ }
                public void print() { /*pass*/ }
        }

        public static void main(String … args) {
                Date myDate = Date.today();
                myDate.print();
        }
}
```

Good to know
Sometimes, (public) static nested classes are called top-level nested classes. Why this alternative term makes sense, will be clarified in another lecture when we discuss inner classes.

  - The syntax is not too spectacular. A static nested class is basically put into "its" outer class as a static member.
  - However, the important fact is, that *Date* is now fully nested into *Program* and the file Date.java is not longer existent.

Nested Classes – Tidying up

- Next, we'll move *Date* out of *Program* into another "Utility" class, which we can use to collect different classes.
  - We'll call this class just *Utils*:

```
// <Utils.java>
public class Utils {
    static class Date { // (members hidden)
        public static Date today() { /*pass*/ }
        public void print() { /*pass*/ }
    }
}
```

- In this case we have to qualify the class name of *Date* as *Utils.Date* to access it from another class like *Program*:

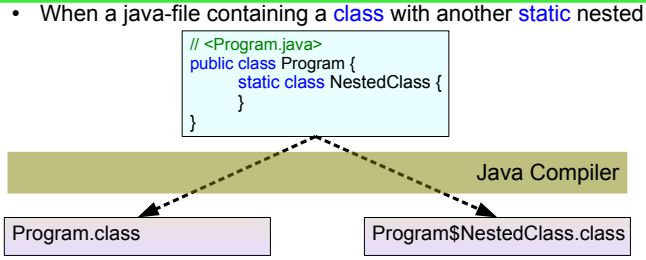
```
// <Program.java>
public class Program {
    public static void main(String ... args) {
        Utils.Date myDate = Utils.Date.today();
        myDate.print();
    }
}
```

- More information:
  - We can also define private nested classes.
    - We could make *Date* private in *Utils* (which doesn't make a lot of sense here), to make it unaccessible from outside of *Utils*.
  - We can also define static nested enums and interfaces (which we discuss in a future lecture).
  - We can nest static nested classes into other nested classes cascadingly, then we need all levels in the qualification.
  - The outer class can access the private members of its nested classes and vice versa.

- Nested static classes shouldn't be used too often.
  - Their benefit is, that we can avoid having a lot of java-files.
  - On the other hand, if all classes are concentrated into one class, the structure of a program is hard to understand from outside.
  - However, the idea to make a static inner class private, to hide its usage, can be a great tool to build complex oo designs.

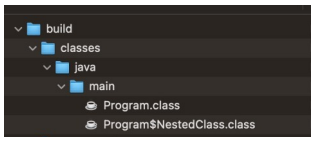
static nested Classes and the Java Compiler

- When a java-file containing a class with another static nested class is compiled...




```
// <build.gradle>
apply plugin: 'java'
apply plugin: 'application'
```

```
Terminal
NicosMBP:MyProject nico$ ./gradlew compileJava
NicosMBP:MyProject nico$
```



  - ...the compiler will generate Program.class for the enclosing class *Program*
  - and the compiler will generate Program$NestedClass.class for the static nested class *Program.NestedClass*.

- The names of the class-files for static nested classes follow a scheme:
  - The file names are prefixed with the name of the enclosing class. **Program**$NestedClass.class
  - Then, following a '$', the name of the static nested class follows. Program$**NestedClass**.class
  - In case, there are more static nested classes cascaded, we'll get more class-files with more '$'-separated class names.

Critique on static nested Classes

- Nested classes are required in Java to put certain functionality into effect, but there is also some critique on the concept.

```
// <OuterClass.java>
public class OuterClass {
        static class NestedClass {
        }
}
```

- What we see here is a simplified UML class diagram, which concentrates on showing connections between classes.
  - (We have just all collapsed the attribute- and operations-compartments, leaving only the name-compartment.)
  - The connector between *OuterClass* and *NestedClass* is a so called association.
  - Actually we see the problem at the association: the association is mutual!
  - It means *OuterClass* depends on *NestedClass* and vice versa! – Both classes know each other! This is a tight coupling!

- The mutual association of nested classes to their outer classes makes understandability and maintenance problematic.
  - It can be difficult to change or refactor such classes, if they have many interdependencies, which is usually true for nested classes.

Thank you!