(1) Java Abstractions: User defined Types

Nico Ludwig (@ersatzteilchen)

TOC

- (1) Java Abstractions: User defined Types
  - Record oriented Programming
  - Abstracted Types
  - Reference Types in Memory
  - Concepts of Object Orientation
  - Constructors
  - Encapsulation
  - Unified Modeling Language
  - Packages – A first Glimpse

- Cited Literature:
  - Just Java, Peter van der Linden
  - Thinking in Java, Bruce Eckel
  - Growing Object-Oriented Software, Guided by Tests, Steve Freeman, Nat Pryce

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!

Separated Data that is not independent

- Let's assume we have some methods dealing with day, month and year, obviously reflecting a calendar date:

```
public class Program {
        static void printDate(int day, int month, int year) {
             System.out.println(day+"."+month+"."+year);
        }
        static int readDay() {
             Scanner inputScanner = new Scanner(System.in);
             return inputScanner.nextInt();
        }
        static int readMonth() {
             Scanner inputScanner = new Scanner(System.in);
             return inputScanner.nextInt();
        }
        static int readYear() {
             Scanner inputScanner = new Scanner(System.in);
             return inputScanner.nextInt();
        }
}
```

- We can then use these methods like so:

```
// Three independent ints representing a single date:
int day = 17, month = 10, year = 2012;
printDate(day, month, year);
// >17.10.2012
// Read a day from the console.
day = readDay();
// <18
printDate(day, month, year); // The day-"part" has been modified.
// >18.10.2012
```

There are Problems with this Approach

- Yes! The presented solution works!
  - We end in a set of static methods (and later also static types).
  - Such a set of methods and types to help implementing software is called Application Programming Interface (API).
  - An API is a kind of collection of building blocks to create applications.
  - Here we have an API to read a day value from console and print date infos to console.

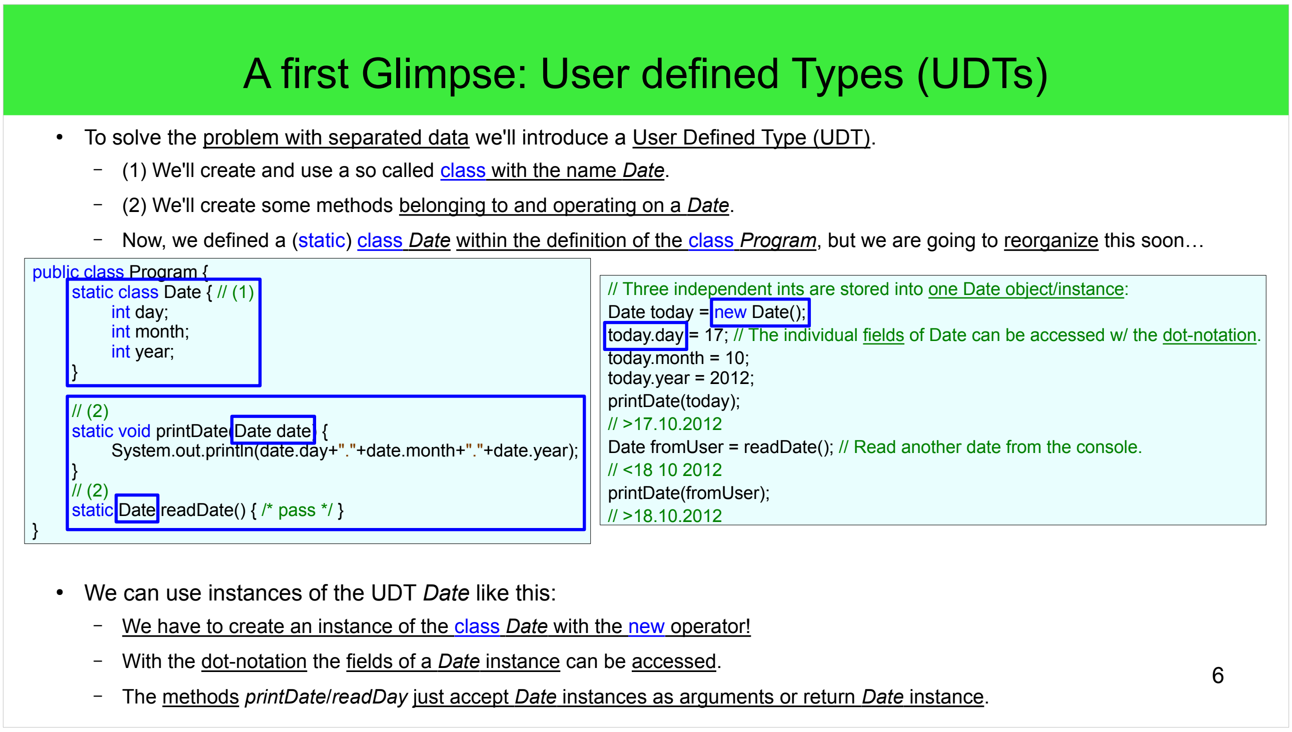
Definition
An Application Programming Interface (API) is a standardized collection of methods and interfaces to program specific applications.

- But there are serious problems with our way of dealing with day, month and year:
  - We have always to pass three separate ints to *printDate()*.
  - We have to know that these separate ints belong together, they make up a (calendar) date!
  - The "concept" of a date is completely hidden! – We have "just three ints".
  - So, after some time of developing we have to remember the concept once again!

```
// How do these ints belong together?
int day = 17;
int month = 10;
int year = 2012;
printDate(day, month, year);
// >17.10.2012
```

- => We have serious sources of difficult-to-track-down programming errors!
  - E.g. we can change some variable names, easily obscuring the meaning of the code!
- The problem: we have to handle pieces of data that somehow belong together!
  - The "belonging together" defines the concept that we have to find.

```
// Oups!
int df = 10;
int rg = 2012;
int kl = 17;
printDate(df, rg, kl);
// >10.2012.17
```



A first Glimpse: User defined Types (UDTs)

- To solve the problem with separated data we'll introduce a User Defined Type (UDT).
  - (1) We'll create and use a so called class with the name *Date*.
  - (2) We'll create some methods belonging to and operating on a *Date*.
  - Now, we defined a (static) class *Date* within the definition of the class *Program*, but we are going to reorganize this soon…

```
public class Program {
    static class Date { // (1)
        int day;
        int month;
        int year;
    }
    // (2)
    static void printDate(Date date) {
        System.out.println(date.day+"."+date.month+"."+date.year);
    }
    // (2)
    static Date readDate() { /* pass */ }
}
```

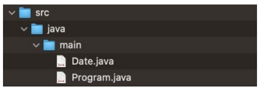
```
// Three independent ints are stored into one Date object/instance:
Date today = new Date();
today.day = 17; // The individual fields of Date can be accessed w/ the dot-notation.
today.month = 10;
today.year = 2012;
printDate(today);
// >17.10.2012
Date fromUser = readDate(); // Read another date from the console.
// <18 10 2012
printDate(fromUser);
// >18.10.2012
```

- We can use instances of the UDT *Date* like this:
  - We have to create an instance of the class *Date* with the new operator!
  - With the dot-notation the fields of a *Date* instance can be accessed.
  - The methods *printDate*/*readDay* just accept *Date* instances as arguments or return *Date* instance.

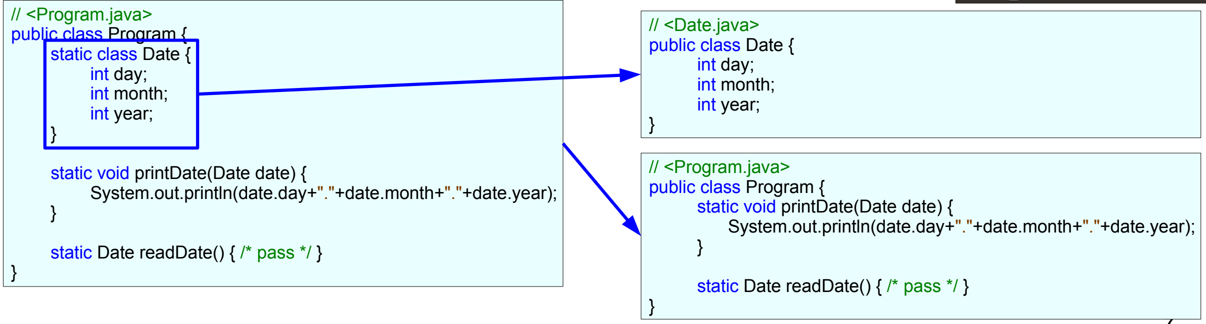
- Arrays are also UDTs!
- The phrase "belonging to methods" can be clearly explained now, e.g. the method *Date.printDate* depends on the bare presence of the definition of the UDT *Date*!

Basic Features of UDTs – Part 1 – Organization in separate Source Files

- Now it's time to discuss how UDTs are organized in Java.
  - Each individual/top level definition of a public class has to reside in its own file with the suffix .java.
    - Making a class public guarantees, that we can use it from "everywhere" in our program.
  - The file containing the public class needs to have exactly the same name of the class. – Also the casing must match exactly!
  - => Hence we will define new classes in own files each.
  - => Hence we assume, that all the java-files of our programs reside in the same directory.
  - Eventually we got rid of static classes within the class *Program*!



```
src
  java
    main
      Date.java
      Program.java
```

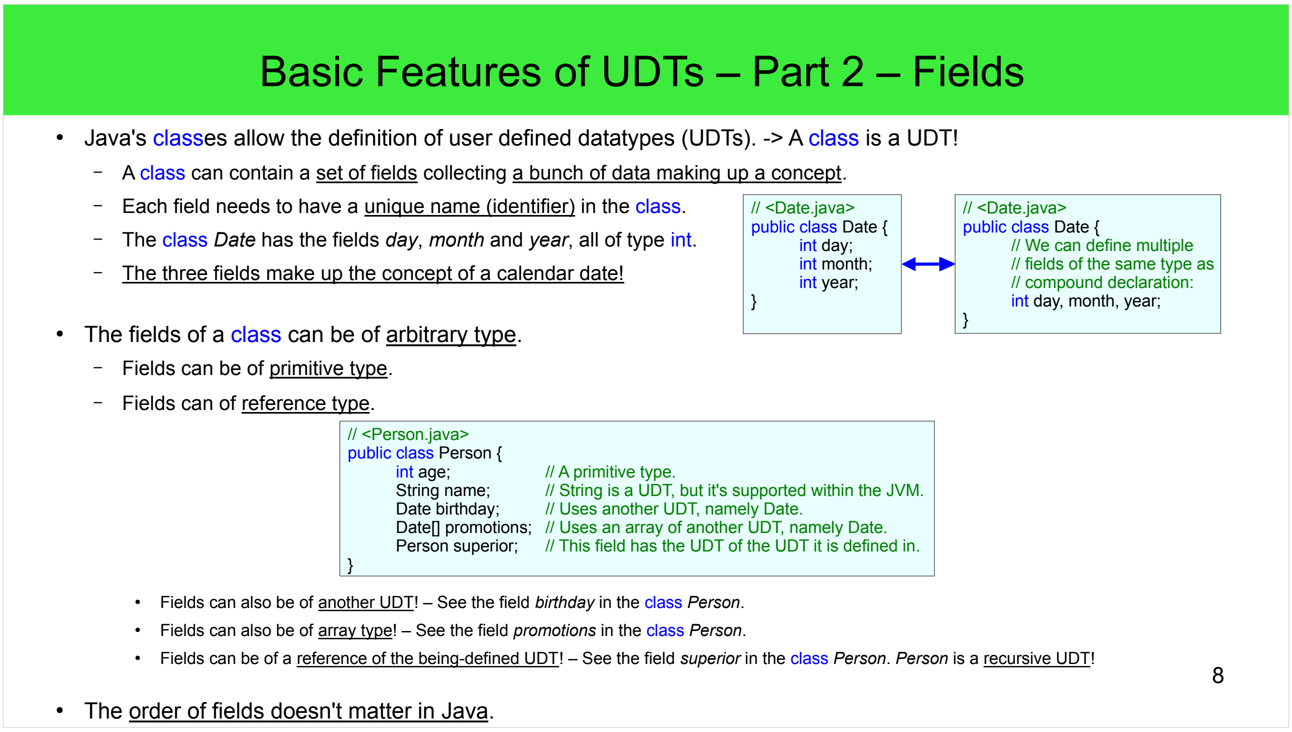


```
// <Program.java>
public class Program {
	static class Date {
		int day;
		int month;
		int year;
	}

	static void printDate(Date date) {
		System.out.println(date.day+"."+date.month+"."+date.year);
	}

	static Date readDate() { /* pass */ }
}
```

```
// <Date.java>
public class Date {
	int day;
	int month;
	int year;
}
```

```
// <Program.java>
public class Program {
	static void printDate(Date date) {
		System.out.println(date.day+"."+date.month+"."+date.year);
	}

	static Date readDate() { /* pass */ }
}
```

- User defined enums and interfaces are usually also organized in separate java-files.



Basic Features of UDTs – Part 2 – Fields

- Java's classes allow the definition of user defined datatypes (UDTs). -> A class is a UDT!
  - A class can contain a set of fields collecting a bunch of data making up a concept.
  - Each field needs to have a unique name (identifier) in the class.
  - The class *Date* has the fields *day*, *month* and *year*, all of type int.
  - The three fields make up the concept of a calendar date!

```
// <Date.java>
public class Date {
        int day;
        int month;
        int year;
}
```

```
// <Date.java>
public class Date {
        // We can define multiple
        // fields of the same type as
        // compound declaration:
        int day, month, year;
}
```

- The fields of a class can be of arbitrary type.
  - Fields can be of primitive type.
  - Fields can of reference type.

```
// <Person.java>
public class Person {
        int age;                // A primitive type.
        String name;            // String is a UDT, but it's supported within the JVM.
        Date birthday;          // Uses another UDT, namely Date.
        Date[] promotions;      // Uses an array of another UDT, namely Date.
        Person superior;        // This field has the UDT of the UDT it is defined in.
}
```

- Fields can also be of another UDT! – See the field *birthday* in the class *Person*.
- Fields can also be of array type! – See the field *promotions* in the class *Person*.
- Fields can be of a reference of the being-defined UDT! – See the field *superior* in the class *Person*. *Person* is a recursive UDT!

- The order of fields doesn't matter in Java.

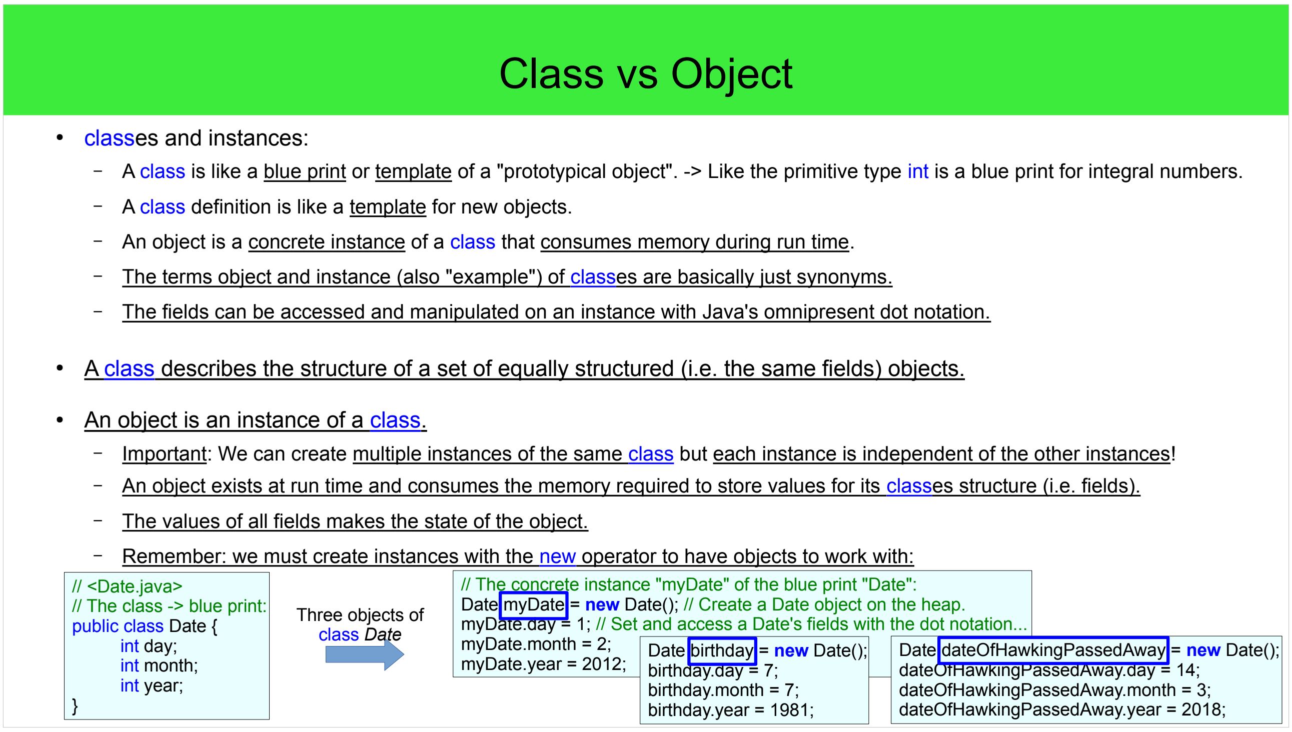
- In C# and Java, the syntactic definitions of UDTs are not terminated by semicolons!

Basic Features of UDTs – Part 3 – UDTs, Record-Types

- The idea of a UDT is the invention of a new type, composed of other types.
  - UDTs can be composed of primitive types and/or composed of other UDTs.
  - => UDTs make APIs really powerful, simple to use and simple to document.

- In general programming terms, UDTs as we defined it just now, are often called record-types.
  - In Java, record-types can be defined with classes obviously.
  - An API consisting of static methods and record-types is a record-oriented API.

- Sometimes, record-types are also called complex types, whereas primitive types are also called scalar types.
  - The terms complex and scalar types stem from the mathematical theory behind linear algebra.
  - The theories behind linear algebra and record-oriented programming have a common ground!
  - Scalar instances consist of one elementary value (e.g. a real number), complex instances consist of a set of elementary values (e.g. a vector).



Class vs Object

- classes and instances:
  - A class is like a blue print or template of a "prototypical object". -> Like the primitive type int is a blue print for integral numbers.
  - A class definition is like a template for new objects.
  - An object is a concrete instance of a class that consumes memory during run time.
  - The terms object and instance (also "example") of classes are basically just synonyms.
  - The fields can be accessed and manipulated on an instance with Java's omnipresent dot notation.
- A class describes the structure of a set of equally structured (i.e. the same fields) objects.
- An object is an instance of a class.
  - Important: We can create multiple instances of the same class but each instance is independent of the other instances!
  - An object exists at run time and consumes the memory required to store values for its classes structure (i.e. fields).
  - The values of all fields makes the state of the object.
  - Remember: we must create instances with the new operator to have objects to work with:

```
// <Date.java>
// The class -> blue print:
public class Date {
        int day;
        int month;
        int year;
}
```

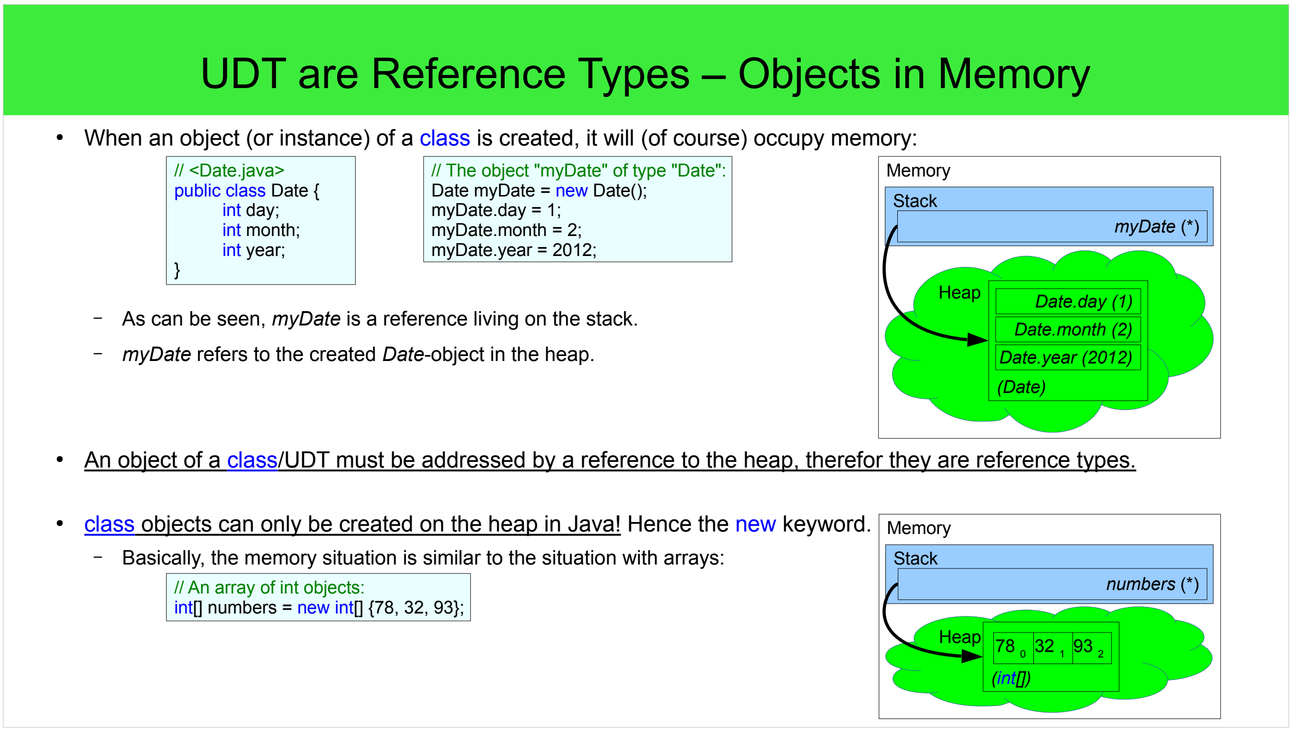
Three objects of class *Date*

```
// The concrete instance "myDate" of the blue print "Date":
Date myDate = new Date(); // Create a Date object on the heap.
myDate.day = 1; // Set and access a Date's fields with the dot notation...
myDate.month = 2;
myDate.year = 2012;
```

```
Date birthday = new Date();
birthday.day = 7;
birthday.month = 7;
birthday.year = 1981;
```

```
Date dateOfHawkingPassedAway = new Date();
dateOfHawkingPassedAway.day = 14;
dateOfHawkingPassedAway.month = 3;
dateOfHawkingPassedAway.year = 2018;
```

- In languages like Java, C++ and C# classes act as a template for objects. Languages like JavaScript offer another approach. In JavaScript classes are not required, instead a prototypical object can be created to which fields and methods can be added. So fields and methods exist on an individual object not on a class, which is the common type of multiple objects. This kind of object-orientation is sometimes called prototype-based object-orientation (in opposite to class-based object-orientation as offered in Java).



UDT are Reference Types – Objects in Memory

- When an object (or instance) of a class is created, it will (of course) occupy memory:

```
// <Date.java>
public class Date {
        int day;
        int month;
        int year;
}
```

```
// The object "myDate" of type "Date":
Date myDate = new Date();
myDate.day = 1;
myDate.month = 2;
myDate.year = 2012;
```

  - As can be seen, *myDate* is a reference living on the stack.
  - *myDate* refers to the created *Date*-object in the heap.

- An object of a class/UDT must be addressed by a reference to the heap, therefor they are reference types.

- class objects can only be created on the heap in Java! Hence the new keyword.
  - Basically, the memory situation is similar to the situation with arrays:

```
// An array of int objects:
int[] numbers = new int[] {78, 32, 93};
```

- The size of UDT-objects is not clearly defined in Java. However, there is a way to get the length of the serialized byte[] of an object:

```
public class Date implements Serializable {
        int day;
        int month;
        int year;
}
```

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
try (ObjectOutputStream oos = new ObjectOutputStream(baos)) {
        oos.writeObject(new Date());
}
System.out.println("Date's serialized length: "+baos.toByteArray().length);
// >Date's serialized length: 60
```

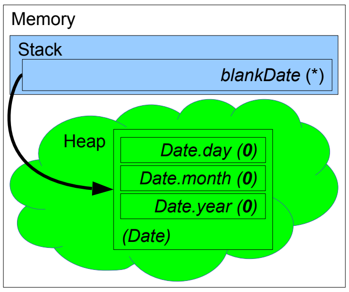
- The requirement is, that the UDT is question is serializable (for UDTs: implement *Serializable*).
- The resulting length could be different on different platforms.

UDT are Reference Types – Uninitialized Fields

- When the fields of a class are uninitialized in the object, they have guaranteed default-values in Java:

```
// <Date.java>
public class Date {
        int day;
        int month;
        int year;
}
```

```
// The object "blankDate" of type "Date":
Date blankDate = new Date();
// We don't assign values to the fields!
```



  - Fields of integral (byte, char, short, int and long) type will default to 0.
  - Fields of floaty type (float, double) will default to 0.0f/0.0.
  - Fields of boolean type will default to false.
  - Fields of reference type will default to null.

- Mind, that there is a difference between local variables and fields: fields need not to be effectively initialized before usage!

- It must be said, that this "0/false/null-defaulting" is good, and esp. safer than dealing with arbitrary default-values.
  - Arbitrary values, i.e. completely undefined values in uninitialized fields, is the way C++ works which is a source of dangerous bugs!

Arrays of UDTs

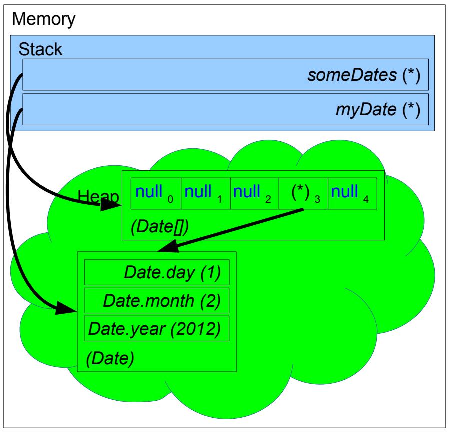
- We can create arrays of UDTs:

```
Date[] someDates = new Date[5]; // Create an array of five (uninitialized) Dates.
```

- A very important point is, that all five elements of *someDates* are null references!
  - Therefore we have to set the elements of the array to (non-null) *Date* objects explicitly.
  - E.g. let's set the fourth element of *someDates*:

```
Date myDate = new Date(); // Create a Date object on the heap.
myDate.day = 1; // Set and access a Date's fields with the dot notation...
myDate.month = 2;
myDate.year = 2012;
someDates[3] = myDate; // Copy a reference to myDate.
```

  - Mind, that all other elements of *myDate* are still null-references!

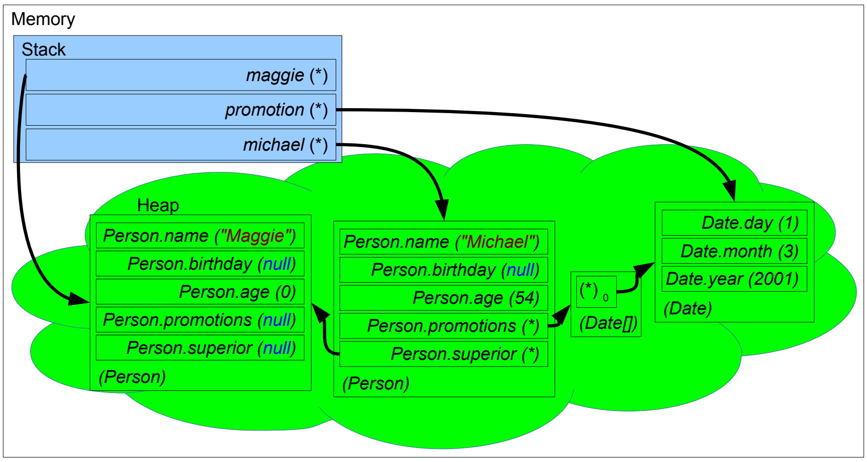


More complex Objects in Memory – Part 1

- Now we'll inspect the situation with the UDT *Person* that has fields of primitive type and UDTs:

```
// <Person.java>
public class Person {
        int age;
        String name;
        Date birthday;
        Date[] promotions;
        Person superior;
}
```

```
Person maggie = new Person();
maggie.name = "Maggie";

Date promotion = new Date();
promotion.day = 1;
promotion.month = 3;
promotion.year = 2001;

Person michael = new Person();
michael.age = 54;
michael.name = "Michael";
michael.promotions = new Date[] {promotion};
michael.superior = maggie;
```



Memory

Stack

maggie (*)

promotion (*)

michael (*)

Heap

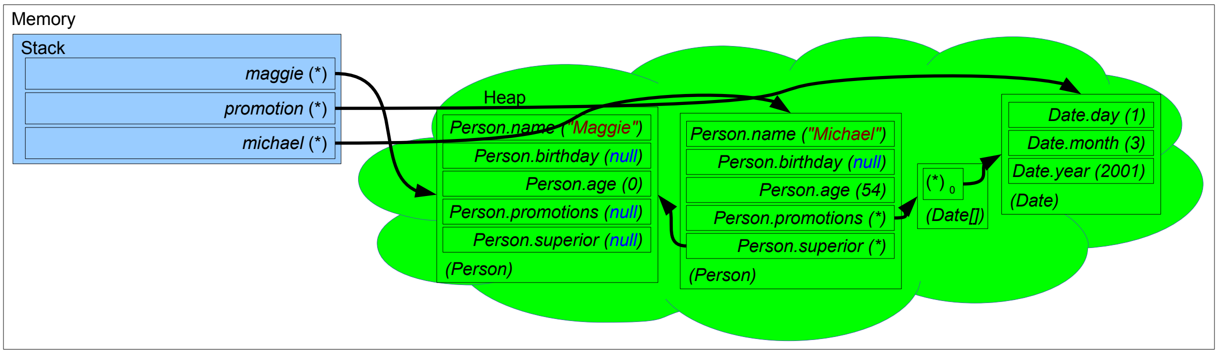
Person.name ("Maggie")
Person.birthday (null)
Person.age (0)
Person.promotions (null)
Person.superior (null)
(Person)

Person.name ("Michael")
Person.birthday (null)
Person.age (54)
Person.promotions (*)
Person.superior (*)
(Person)

(*) $_0$
(Date[])

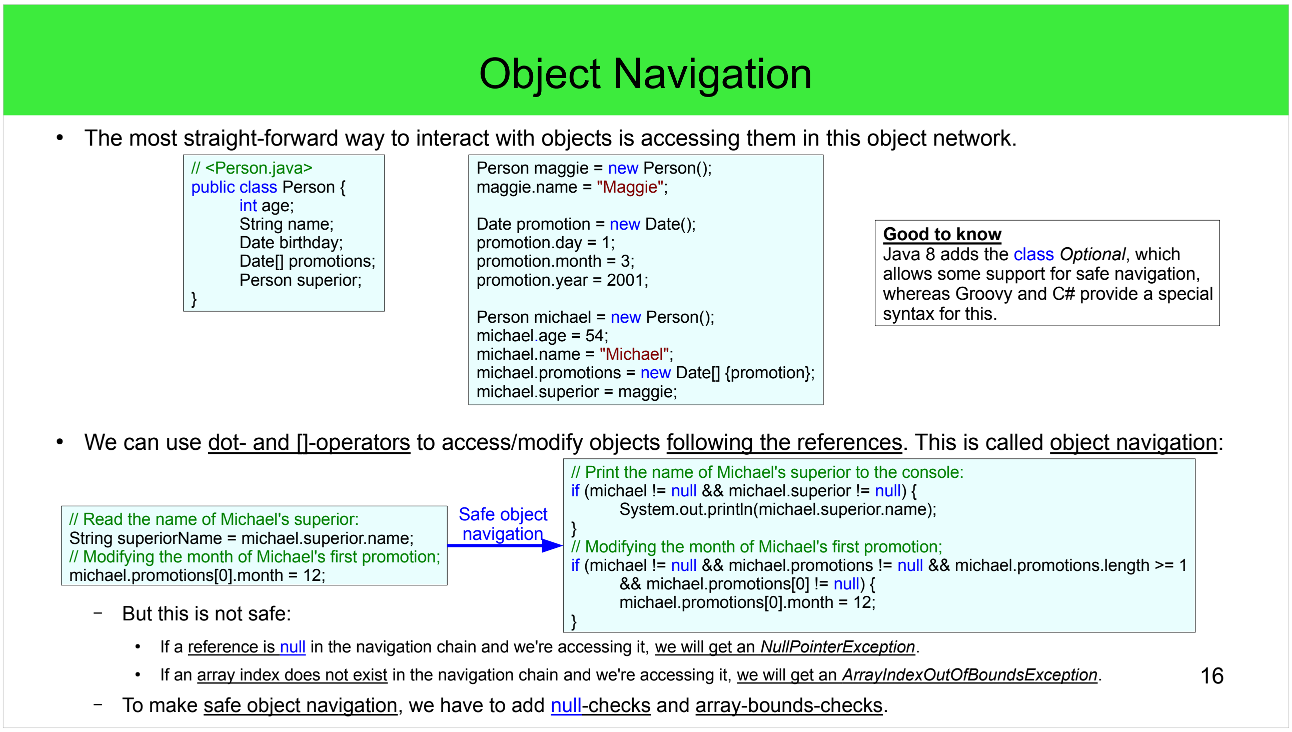
Date.day (1)
Date.month (3)
Date.year (2001)
(Date)

More complex Objects in Memory – Part 2



- Notice:
  - UDTs are always reference types!
  - Objects can refer to each other and reference variables on the stack refer to objects in the heap.
  - Fields of reference type, we haven't set, like *birthday* refer to no object and have the default value null.

- A instances having other UDT-fields, which are references to other instance build up a network of objects in memory.



Object Navigation

- The most straight-forward way to interact with objects is accessing them in this object network.

```
// <Person.java>
public class Person {
        int age;
        String name;
        Date birthday;
        Date[] promotions;
        Person superior;
}
```

```
Person maggie = new Person();
maggie.name = "Maggie";

Date promotion = new Date();
promotion.day = 1;
promotion.month = 3;
promotion.year = 2001;

Person michael = new Person();
michael.age = 54;
michael.name = "Michael";
michael.promotions = new Date[] {promotion};
michael.superior = maggie;
```

Good to know
Java 8 adds the class *Optional*, which allows some support for safe navigation, whereas Groovy and C# provide a special syntax for this.

- We can use dot- and []-operators to access/modify objects following the references. This is called object navigation:

```
// Read the name of Michael's superior:
String superiorName = michael.superior.name;
// Modifying the month of Michael's first promotion;
michael.promotions[0].month = 12;
```

```
// Print the name of Michael's superior to the console:
if (michael != null && michael.superior != null) {
        System.out.println(michael.superior.name);
}
// Modifying the month of Michael's first promotion;
if (michael != null && michael.promotions != null && michael.promotions.length >= 1
        && michael.promotions[0] != null) {
        michael.promotions[0].month = 12;
}
```

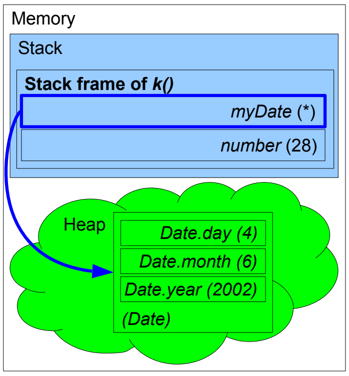
  - But this is not safe:
    - If a reference is null in the navigation chain and we're accessing it, we will get an *NullPointerException*.
    - If an array index does not exist in the navigation chain and we're accessing it, we will get an *ArrayIndexOutOfBoundsException*.
  - To make safe object navigation, we have to add null-checks and array-bounds-checks.

- There are at least two other important differences between C/C++' pointers and Java's references:
  - The state of references is clearly defined in Java: it either refers to an object or is a null-reference. – A reference cannot be in an undefined state.
  - Calling methods on a null-reference in Java (i.e. dereferencing a pointer in C/C++) will throw an NPE. In C/C++ dereferencing an uninitialized pointer or null-pointer leads to undefined behavior.

UDT Objects and Local Variables in Memory

- Let's have another look at how *Date* objects reside in memory:

```
// <Program.java>
public class Program {
        static void k() {
              Date myDate = new Date();
              myDate.day = 4;
              myDate.month = 6;
              myDate.year = 2002;
              int number = 28;
        }
}
```

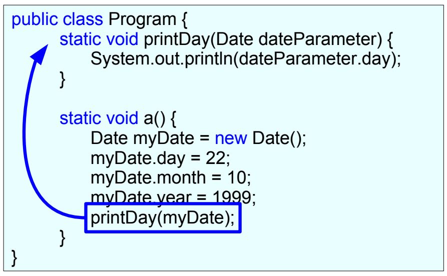


- The memory diagram shows an important truth about memory in Java:
  - Local variables are always created on the stack, instance variables are always created on the heap.
  - The locals *myDate* and *number* live on the stack, the instance variables' values of the *Date*-object "behind" *myDate* live on the heap.
    - I.e. *myDate.day*, *myDate.month* and *myDate.year* are all living on the heap. Whereas *numbers* live on the stack
  - Bottom line: values of value type (e.g. int) reside on the heap, if they are used for fields of a UDT.

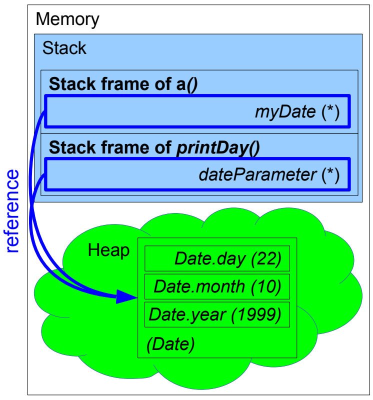
Passing UDT Objects to Methods – Part 1

- We've noticed, that arguments are passed to methods by value: a parameter's value is a copy of the argument's value.

- This is also true for class objects; classes are reference types: when we pass a reference to a method, the reference will be copied:



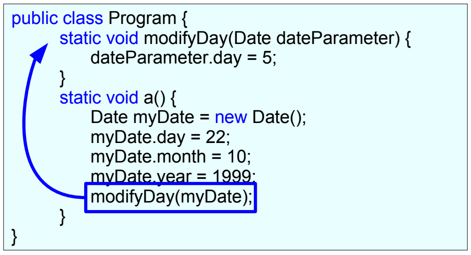
```
public class Program {
    static void printDay(Date dateParameter) {
        System.out.println(dateParameter.day);
    }

    static void a() {
        Date myDate = new Date();
        myDate.day = 22;
        myDate.month = 10;
        myDate.year = 1999;
        printDay(myDate);
    }
}
```



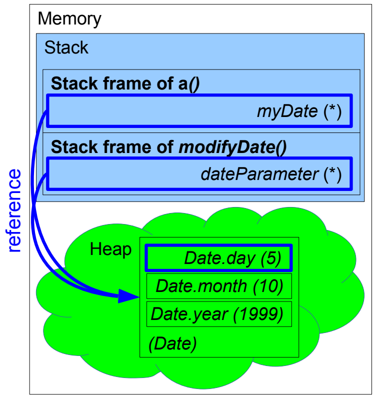
- "class objects" are always reference types!
  - Practically, it means that *myDate* and *dateParameter* are different references, which refer to the same object.
  - Remember this is called aliasing.

Passing UDT Objects to Methods – Part 2

- Because only references to UDT objects are passed around, we have read and write access to the same, shared object:



```
public class Program {
    static void modifyDay(Date dateParameter) {
        dateParameter.day = 5;
    }
    static void a() {
        Date myDate = new Date();
        myDate.day = 22;
        myDate.month = 10;
        myDate.year = 1999;
        modifyDay(myDate);
    }
}
```



- In *modifyDate()* we can modify the *day* field, which was created and passed from *a()* to *modifyDate()* as part of *myDate*!
  - We have to keep this in mind during programming, because it can be source of bugs!

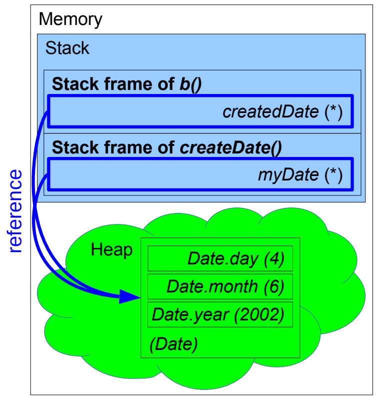
- If it is a source of bugs, why is aliasing supported? – Incessant (deep) copying of full UDT objects would be very costly!

Returning UDT Objects from Methods

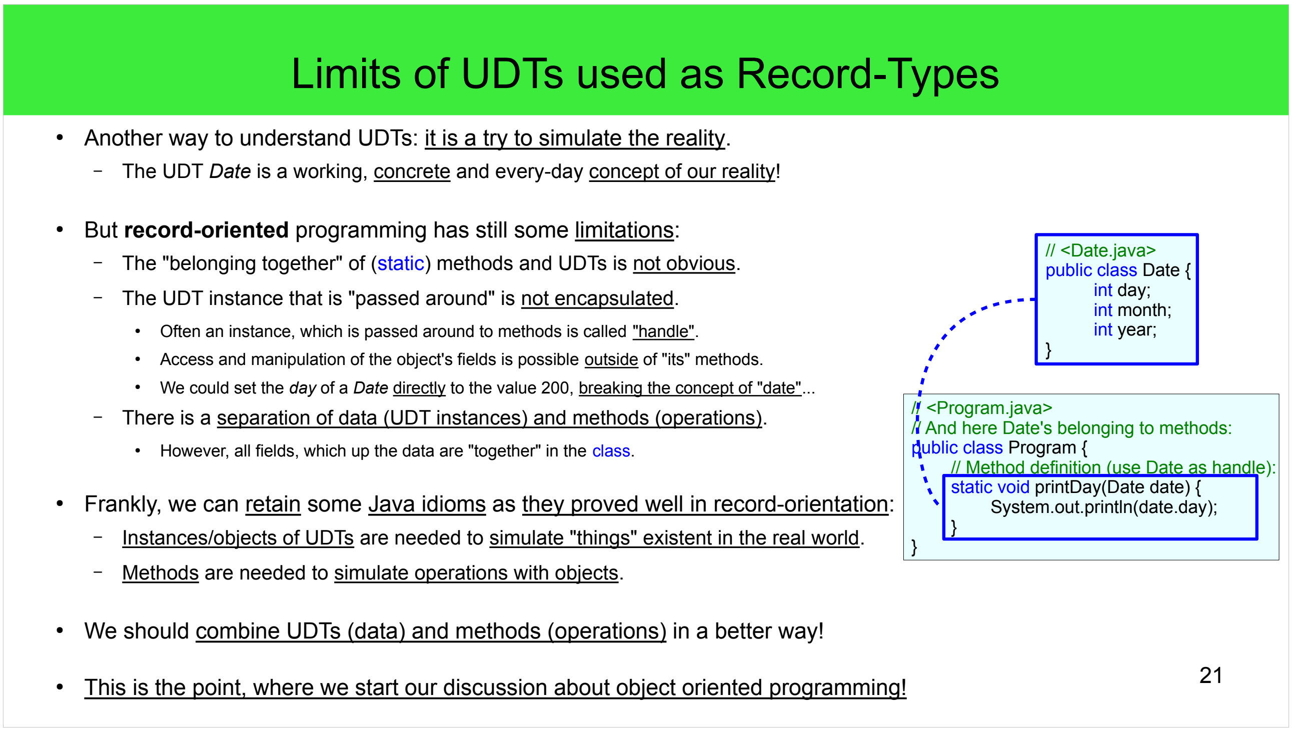
- Also values, which are returned from methods are passed by value: an "accepted" value is a copy of the returned value.

- This is also true for "class objects", mind, that classes are reference types: if we return a reference from a method, the reference will be copied:

```
public class Program {
        static Date createDate() {
                Date myDate = new Date();
                myDate.day = 4;
                myDate.month = 6;
                myDate.year = 2002;
                return myDate; // returned value
        }
        static void b() {
                // createdDate contains the accepted value
                Date createdDate = createDate();
        }
}
```



- "class objects" are always reference types!
  - It means that the value returned from *createDate()* and the accepted value in *a()* are different references, which refer to the same object.
  - This is also an aliasing effect.



Limits of UDTs used as Record-Types

- Another way to understand UDTs: it is a try to simulate the reality.
  - The UDT *Date* is a working, concrete and every-day concept of our reality!
- But **record-oriented** programming has still some limitations:
  - The "belonging together" of (static) methods and UDTs is not obvious.
  - The UDT instance that is "passed around" is not encapsulated.
    - Often an instance, which is passed around to methods is called "handle".
    - Access and manipulation of the object's fields is possible outside of "its" methods.
    - We could set the *day* of a *Date* directly to the value 200, breaking the concept of "date"...
  - There is a separation of data (UDT instances) and methods (operations).
    - However, all fields, which up the data are "together" in the class.
- Frankly, we can retain some Java idioms as they proved well in record-orientation:
  - Instances/objects of UDTs are needed to simulate "things" existent in the real world.
  - Methods are needed to simulate operations with objects.
- We should combine UDTs (data) and methods (operations) in a better way!
- This is the point, where we start our discussion about object oriented programming!

```
// <Date.java>
public class Date {
        int day;
        int month;
        int year;
}
```

```
// <Program.java>
// And here Date's belonging to methods:
public class Program {
    // Method definition (use Date as handle):
    static void printDay(Date date) {
        System.out.println(date.day);
    }
}
```

- Mind that we meaningful inventions were also based on the simulation of technology, e.g. the wings of a plane were inspired by birds' bird wings.

Concepts of Object Orientation

- **Abstracted types** are not only concerning a set of data (record)! – Its aim is combining self contained data and behavior!

- **Abstracted types** require two concepts:
  - (1) Abstraction by combining data and methods into a UDT to define a concept.
  - (2) Encapsulation to protect data from unwanted access and modification to keep an object in a valid state:
    - The *day*-part of a *Date* instance should not be modifiable from "outside".

- **Object orientation (oo)** is not only combining behavior and data! – Its aim is simulation of reality in a computer program!
  - To simulate reality, oo requires two more concepts:
  - (3) The whole – part (aggregation or composition) association:
    - We say "A car object has an engine object.".
  - (4) The specialization – generalization association:
    - We say "three cars drive in front of me", rather than there "drives a van, a bus and a sedan in front of me". We can generalize, as, e.g., a van is a car.

- "Object-orientation" is only the umbrella term for these four concepts.
  - Oo languages provide idioms that allow expressing these concepts.
  - In the rest of this lecture we're going to understand abstracted types, i.e. **(1) abstraction** and **(2) encapsulation**.

Abstraction of Data and Behavior in UDTs – Part 1

- Let's assume following class *Date* and its belonging to method *printDay()* in the class *Program*:

```
// <Date.java>
public class Date {
        int day;
        int month;
        int year;
}
```

```
// <Program.java>
// The definition of Date's belonging to methods:
public class Program {
        // Method definition.
        static void printDay(Date date) {
                System.out.println(date.day);
        }
}
```

- In Java we can put the belonging to methods into the definition of the class in question:
  - *Date.printDay()* is now a non-static method of *Date*.
  - *Date.printDay()* can directly access a *Date*-object's data/fields, e.g. *day*.
    - So the formally awaited parameter "*Date date*" is no longer required.
    - *Date*'s data and the (formerly) static method *printDay()* are now combined into one UDT.

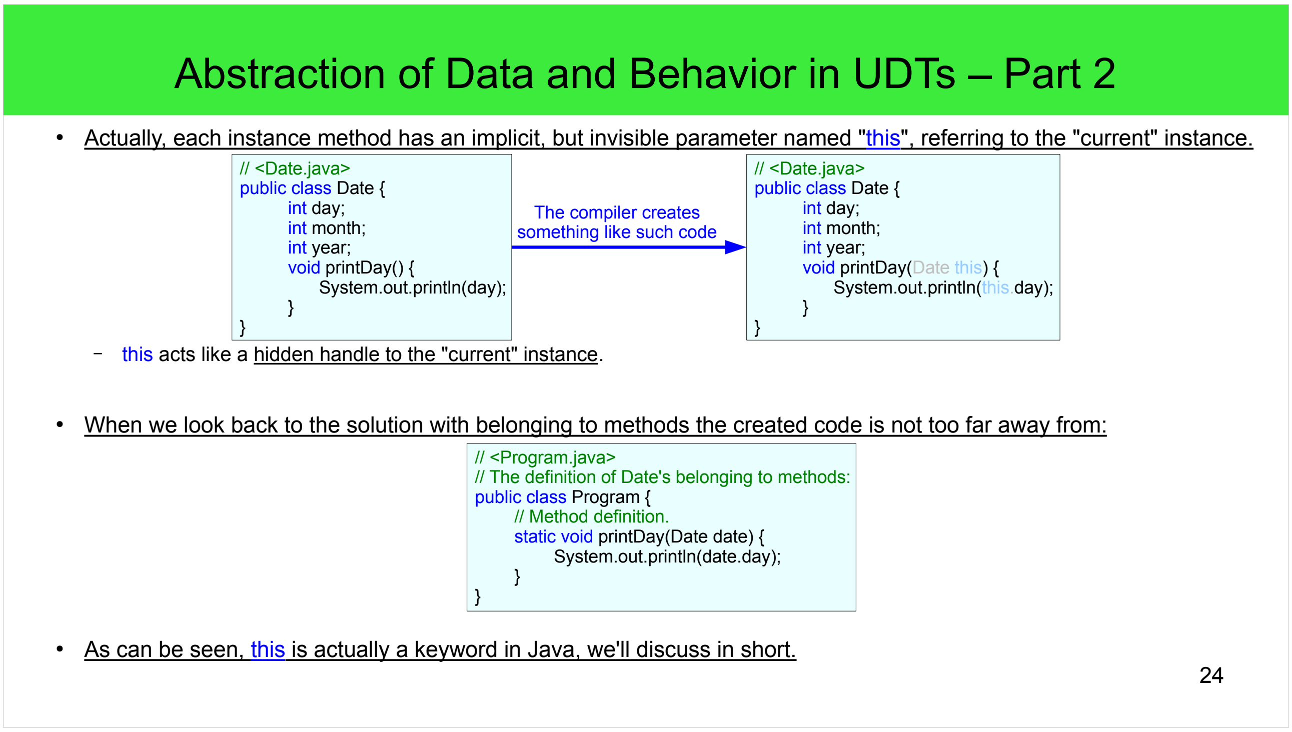
```
// <Date.java>
public class Date {     // Definition of the UDT "Date" composed of
        int day;        // data (i.e. fields) and methods.
        int month;
        int year;
        void printDay() { // Instance method definition.
                System.out.println(day);
        }
}
```

- After we created an instance of *Date*, we can call *printDay()*:

```
Date date = new Date();
date.day = 24;
date.printDay();
// >24
```

```
Date.printDay(); // Invalid! java: non-static method printDay() cannot be referenced from a static context
```

  - A non-static method can only be called on an instance of the defining UDT, therefor we call them instance methods.
  - In opposite static methods are also called class methods.



Abstraction of Data and Behavior in UDTs – Part 2

- Actually, each instance method has an implicit, but invisible parameter named "this", referring to the "current" instance.

```
// <Date.java>
public class Date {
        int day;
        int month;
        int year;
        void printDay() {
              System.out.println(day);
        }
}
```

The compiler creates something like such code

```
// <Date.java>
public class Date {
        int day;
        int month;
        int year;
        void printDay(Date this) {
              System.out.println(this.day);
        }
}
```

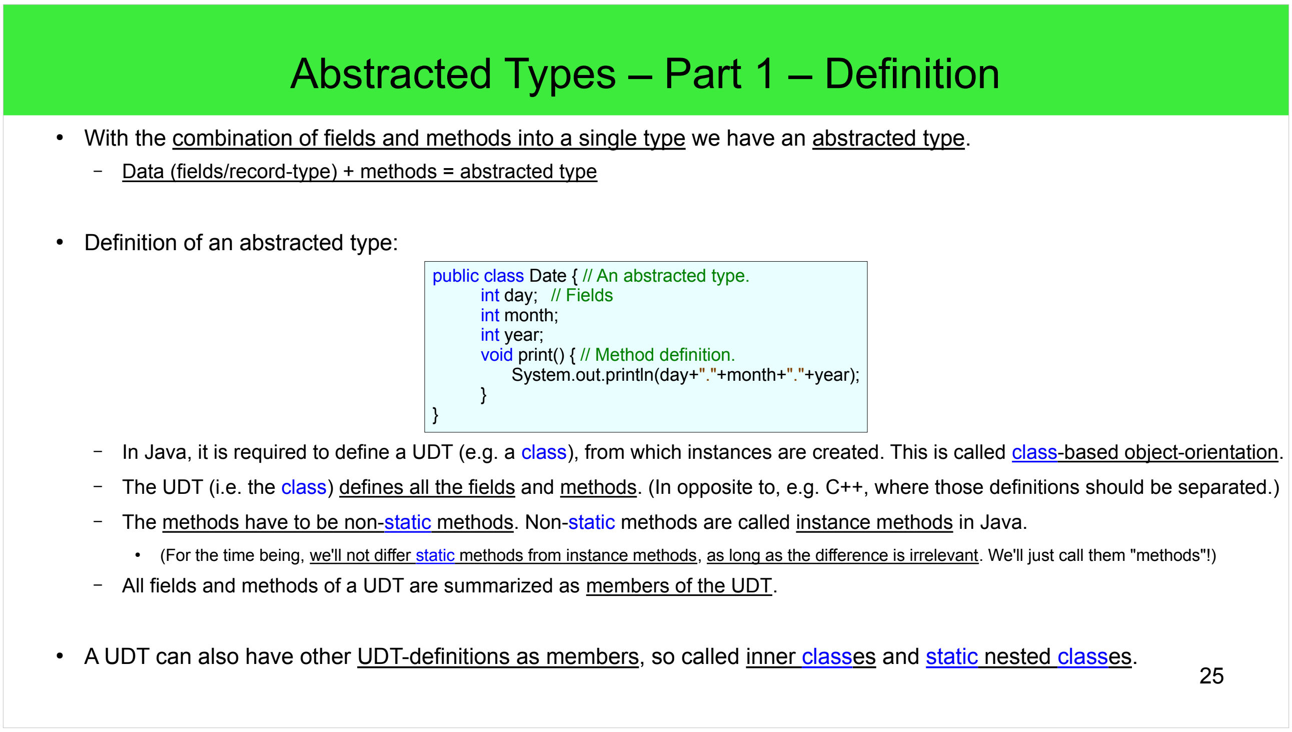
  - this acts like a hidden handle to the "current" instance.

- When we look back to the solution with belonging to methods the created code is not too far away from:

```
// <Program.java>
// The definition of Date's belonging to methods:
public class Program {
        // Method definition.
        static void printDay(Date date) {
              System.out.println(date.day);
        }
}
```

- As can be seen, this is actually a keyword in Java, we'll discuss in short.

- When we discussed Java's memory concepts we learned that arguments and locals of a method are kept in the section "local variables and parameters". The section is organized like an array and the arguments are stored at the first indexes. However, if a method is an instance method, the element at the zeroth index will be the this-reference.



Abstracted Types – Part 1 – Definition

- With the combination of fields and methods into a single type we have an abstracted type.
  - Data (fields/record-type) + methods = abstracted type

- Definition of an abstracted type:

```
public class Date { // An abstracted type.
        int day;   // Fields
        int month;
        int year;
        void print() { // Method definition.
             System.out.println(day+"."+month+"."+year);
        }
}
```

  - In Java, it is required to define a UDT (e.g. a class), from which instances are created. This is called class-based object-orientation.
  - The UDT (i.e. the class) defines all the fields and methods. (In opposite to, e.g. C++, where those definitions should be separated.)
  - The methods have to be non-static methods. Non-static methods are called instance methods in Java.
    - (For the time being, we'll not differ static methods from instance methods, as long as the difference is irrelevant. We'll just call them "methods"!)
  - All fields and methods of a UDT are summarized as members of the UDT.

- A UDT can also have other UDT-definitions as members, so called inner classes and static nested classes.

- Class-based object-orientation means, that UDTs like Java's classes act like a template for objects.
  - In languages like JavaScript, objects can be created without having a "solid" UDT. JavaScript applies so called prototype-based object-orientation.
- We can also define inner interfaces and enums.

Abstracted Types – Part 2 – Definition of Instance Methods

- Generally we already discussed (static) methods. Instance methods are very similar concerning definition:

```
public class Date { // (members hidden)
        void print() { // Method definition.
             System.out.println(day+"."+month+"."+year);
        }
}
```

- Instance methods can return objects or not and they can also have parameters.

- Instance methods can also have multiple overloads.

```
public class Date { // (members hidden)
        void print() { // Method definition.
             System.out.println(day+"."+month+"."+year);
        }
        void print(String text) { // Overloads the method print().
             System.out.println(text+": "+day+"."+month+"."+year);
        }
}
```

```
myDate.print();  // Calls the parameter-less overload of print().
// >17.10.2012
myDate.print("The date is"); // Calls print()'s overload with one String parameter.
// >The date is: 17.10.2012
```

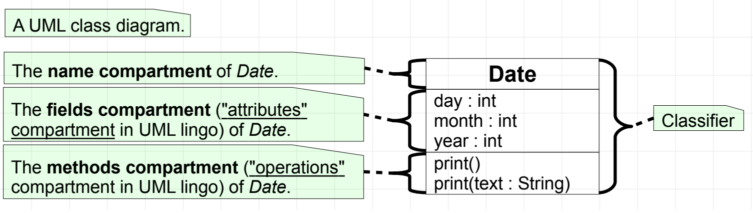
- Methods with the same signature only differing in the static keyword do not overload and lead to a compile time error.

Abstracted Types – Part 3 – Graphical Notation via UML

- The Unified Modeling Language (UML) is a graphical notation to express abstracted types.
  - The UML uses a set of diagram types to show different aspects of a software design.
  - The diagrams can be used to develop and document oo dependencies/structures of a program/system/reality graphically.

- What we see here and in upcoming lectures is a so called class diagram:

```java
// <Date.java>
public class Date {
        int day;
        int month;
        int year;
        void print() {
                // pass
        }
        void print(String text) {
                // pass
        }
}
```



A UML class diagram.

The **name compartment** of *Date*.

The **fields compartment** ("attributes" compartment in UML lingo) of *Date*.

The **methods compartment** ("operations" compartment in UML lingo) of *Date*.

Classifier

  - Classes, called classifiers in UML, are drawn as rectangular boxes, carrying the name of the classifier in **bold** font.
  - Classifiers can also show compartments, that enumerate fields and methods.
  - Boxes with a "dog ear", so called notes, can be used to place comments into the diagram.

Abstracted Types – Part 4 – Calling Instance Methods

- We can use instances of the abstracted type *Date* like this:
  - We already know, that we can access the fields of a *Date* instance with the dot-notation.
  - New to us is, that we can also call instance methods like *Date.print()* with the dot-notation.

```
Date myDate = new Date();
myDate.day = 17; // The individual fields can be accessed with the dot-notation.
myDate.month = 10;
myDate.year = 2012;
myDate.print(); // The methods can be called with the dot-notation as well.
// >17.10.2012
```

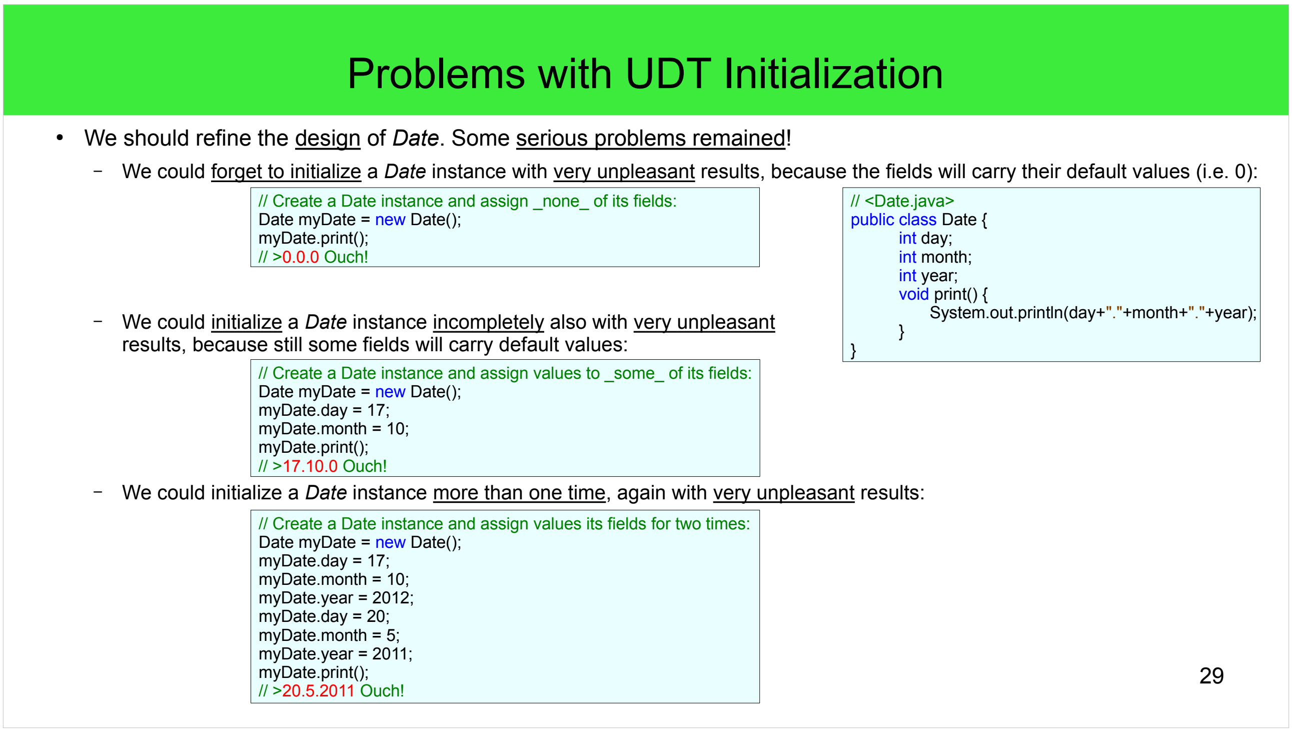
- Keep in mind, that instance methods can not be called on type names!

```
Date.print();  // Invalid! Will result in a compile time error: non-static method print() cannot be referenced from a static context
```

- If we try to call an instance method on a null-reference, a *NullPointerException* (NPE) will be thrown:

```
// Defined a Date reference and initialize it to null:
Date myDate = null;
myDate.print(); // Calling print() on a null reference will throw a NullPointerException NPE.
```

Good to know
NullPointerException is a bad name! A more correct name would be "*NullReferenceException*", because *myDate* is not a pointer – in Java it is a reference. Maybe the naming came from C++, which was the "model" language for Java, where pointers are a concept similar to Java's references.



Problems with UDT Initialization

- We should refine the design of *Date*. Some serious problems remained!
  - We could forget to initialize a *Date* instance with very unpleasant results, because the fields will carry their default values (i.e. 0):

```
// Create a Date instance and assign _none_ of its fields:
Date myDate = new Date();
myDate.print();
// >0.0.0 Ouch!
```

```
// <Date.java>
public class Date {
        int day;
        int month;
        int year;
        void print() {
             System.out.println(day+"."+month+"."+year);
        }
}
```

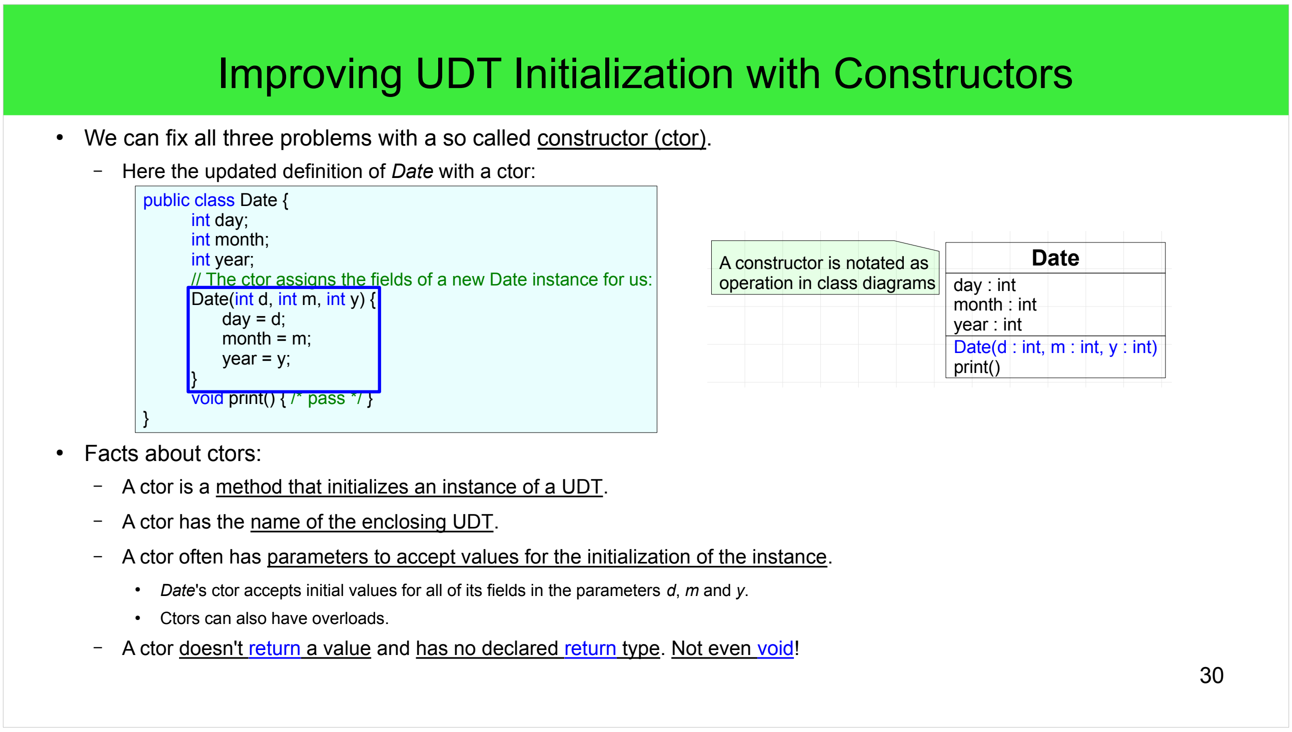
  - We could initialize a *Date* instance incompletely also with very unpleasant results, because still some fields will carry default values:

```
// Create a Date instance and assign values to _some_ of its fields:
Date myDate = new Date();
myDate.day = 17;
myDate.month = 10;
myDate.print();
// >17.10.0 Ouch!
```

  - We could initialize a *Date* instance more than one time, again with very unpleasant results:

```
// Create a Date instance and assign values its fields for two times:
Date myDate = new Date();
myDate.day = 17;
myDate.month = 10;
myDate.year = 2012;
myDate.day = 20;
myDate.month = 5;
myDate.year = 2011;
myDate.print();
// >20.5.2011 Ouch!
```

- Notice that the word "design" was used. Mind that we try to simulate the reality, and "simulating the reality" or "simulating the nature" is another definition of the term "art". Oo programming has many parallels to art, esp. do oo-programmers have to work creatively.



Improving UDT Initialization with Constructors

- We can fix all three problems with a so called constructor (ctor).
  - Here the updated definition of *Date* with a ctor:

```
public class Date {
        int day;
        int month;
        int year;
        // The ctor assigns the fields of a new Date instance for us:
        Date(int d, int m, int y) {
             day = d;
             month = m;
             year = y;
        }
        void print() { /* pass */ }
}
```

A constructor is notated as operation in class diagrams

Date
day : int month : int year : int

Date(d : int, m : int, y : int)
print()

- Facts about ctors:
  - A ctor is a method that initializes an instance of a UDT.
  - A ctor has the name of the enclosing UDT.
  - A ctor often has parameters to accept values for the initialization of the instance.
    - *Date*'s ctor accepts initial values for all of its fields in the parameters *d*, *m* and *y*.
    - Ctors can also have overloads.
  - A ctor doesn't return a value and has no declared return type. Not even void!

- Why were the parameters named *d*, *m* and *y* and not *day*, *month* and *year*?

Calling Constructors – Part 1

- The definition of ctors is one thing, but their usage is far more interesting!

```
// Create a Date instance with the ctor and pass values to initialize its fields:
Date myDate = new Date(17, 10, 2012); // The ctor performs the assignment of the fields!
myDate.print();
// >17.10.2012
```

- The syntax of calling a ctor is like calling a method while creating an instance.
  - Indeed ctors are methods. – Only the definition and usage is somewhat special.

- Due to the bare syntax of the ctor call:
  - 1. There is no way to forget to call the ctor!

```
Date myDate = new Date(); // Invalid! Doesn't call the ctor we've defined!
```

  - 2. There is no way to call a ctor and miss any of the initialization values!
    - You have to pass arguments to satisfy all of the ctor's parameters!
  - 3. There is no way to call a ctor more than once on the same instance!
    - Multiple initialization is not possible.

Calling Constructors – Part 2

- Ctors make it easy to create an instance of a class while passing it to or returning it from a method:

```
public class Program {
        static void printDay(Date dateParameter) {
                System.out.println(dateParameter.day);
        }
        static void a() {
                Date myDate = new Date();
                myDate.day = 22;
                myDate.month = 10;
                myDate.year = 1999;
                printDay(myDate);
        }
}
```

```
public class Program {
        static void modifyDay(Date dateParameter) {
                dateParameter.day = 5;
        }
        static void a() {
                printDay(new Date(22, 10, 1999));
        }
}
```

```
public class Program {
        static Date createDate() {
                Date myDate = new Date();
                myDate.day = 4;
                myDate.month = 6;
                myDate.year = 2002;
                return myDate;
        }
        static void b() {
                Date createdDate = createDate();
        }
}
```

```
public class Program {
        static Date createDate() {
                return new Date(4, 6, 2002);
        }
        static void b() {
                Date createdDate = createDate();
        }
}
```

- This way of calling a ctor creates a kind of anonymous instance, i.e. an anonymous argument or return value.
  - Of course the instance is then held in the parameter named *dateParameter* or assigned to the reference named *createdDate*.
  - From a high-level perspective, it makes ctors kind of enable literals of UDTs: the ctor-call expression represents an instance.

The default Constructor – Definition and Usage

- After we have defined our handy ctor, we have to use it for initialization always:

```
Date myDate = new Date(); // Invalid! We have to call a ctor we've defined!
```

```
Date anotherDate = new Date(17, 10, 2012); // Ok! Calls the ctor.
```

- Additionally, we should also define a default constructor (dctor).

```
public class Date { // (members hidden)
        // The ctor assigns the fields of a new Date instance for us:
        Date() {                // This dctor assigns the
            day = 1;            // fields of a new Date
            month = 1;          // instance to meaningful
            year = 1970;        // default values.
        }
        Date(int d, int m, int y) { /* other ctor, pass */ }
}
```

Good to know
The special term "default constructor" is common sense in many languages and frameworks, sometimes it is called also "parameterless constructor". Other terms like "standard constructor" or "common constructor" (German: "allgemeiner Konstruktor") do simply not exist officially. The leading sources for technical terms are specs and compiler messages, but neither professors nor teachers nor books.

  - A dctor initializes a UDT with a default state, i.e. with default values for a UDT's fields.
  - Usually this means that all fields are initialized with values that are meaningful defaults.

- So as ctors are methods, a dctor is the parameterless overload of the ctor.
  - Let's use both *Date* ctors to create two instances of *Date*:

```
Date myDate = new Date(); // Now, that's Ok! Calls the dctor.
myDate.print();
// >1.1.1970
```

```
Date anotherDate = new Date(17, 10, 2012); // Call the other overloaded ctor.
anotherDate.print();
// >17.10.2012
```

The default Constructor – Consequences

- If we don't provide any ctor, a dctor will be implicitly created by the compiler.
  - This created dctor default-initializes the fields to the default-values we already know (0/false/null).
  - So, this dctor is implicitly created if not provided, therefor it is called default ctor!

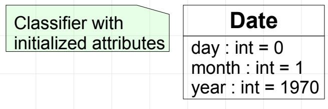
- Java allows to initialize all or only some fields within the class definition by simple assignment.
  - Then these initializations will be simply put into the generated dctor.

```
public class Date { // (members hidden)
        int day;            // The field day defaults to 0.
        int month = 1;      // The fields month and year get
        int year = 1970;    // explicit default values.
}
```

```
public class Date { // (members hidden)
        // Something like this dctor is generated by the compiler:
        Date() {
                day = 0;
                month = 1;
                year = 1970;
        }
}
```

Hint
It can make sense to write an explicit dctor, even if it just does an assignment also an inline assignment to fields could do as shown above to place javadoc comments.

- We can also initialize fields (attributes) of a class (classifier) in a UML class diagram:

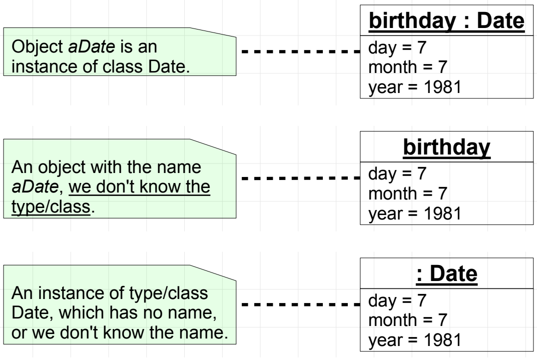


UML Notation of Date Instances

- Let's once again discuss an object of type *Date* (*birthday*):

```
Date birthday = new Date();
birthday.day = 7;
birthday.month = 7;
birthday.year = 1981;
```

- The UML represents objects as boxes with underscored "title" and its assigned to fields:



Object *aDate* is an instance of class Date.

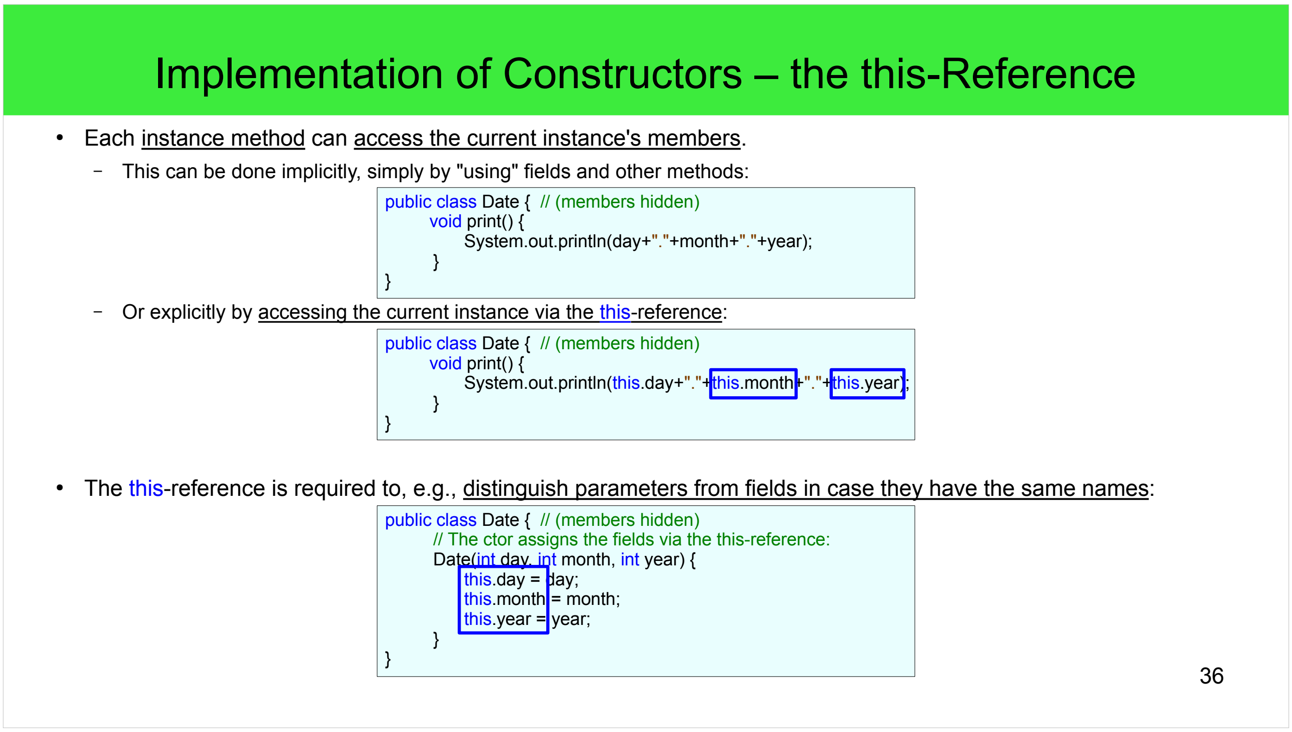
birthday : Date
day = 7
month = 7
year = 1981

An object with the name *aDate*, we don't know the type/class.

birthday
day = 7
month = 7
year = 1981

An instance of type/class Date, which has no name, or we don't know the name.

: Date
day = 7
month = 7
year = 1981



Implementation of Constructors – the this-Reference

- Each instance method can access the current instance's members.
  - This can be done implicitly, simply by "using" fields and other methods:

```
public class Date {  // (members hidden)
        void print() {
              System.out.println(day+"."+month+"."+year);
        }
}
```

  - Or explicitly by accessing the current instance via the this-reference:

```
public class Date {  // (members hidden)
        void print() {
              System.out.println(this.day+"."+this.month+"."+this.year);
        }
}
```

- The this-reference is required to, e.g., distinguish parameters from fields in case they have the same names:

```
public class Date {  // (members hidden)
         // The ctor assigns the fields via the this-reference:
         Date(int day, int month, int year) {
               this.day = day;
               this.month = month;
               this.year = year;
         }
}
```

- The this reference is also useful to trigger code insight/completion on the members of the current instance in the IDE.

Excursus: Variable Shadowing

- In the Java introduction, we learned, that, e.g., two variables of the same name cannot share the same scope:

```
if (x == 78) {
        int y = 23;
        int y = 50; // Invalid! variable y is already defined …
}
```

```
if (x == 78) {
        int y = 23;
        if (z == 8) {
                int y = 50; // Invalid! variable y is already defined …
        }
}
```

- The scoping rules between methods and local variables in methods are more relaxed:

```
public class Date {  // (members hidden)
        int day, month, year;
        Date(int day, int month, int year) { // Ok!
              // pass
        }
}
```

  - Local variables and parameters can have the same names as fields, although they share the same scope.
  - The variables in the methods are said to shadow the equally named fields. There exists local variable and parameter shadowing.

- To distinguish fields from showed variables (parameter shadowing in our case) we have to prefix the field names with this:

```
public class Date {  // (members hidden)
        int day, month, year;
        Date(int day, int month, int year) {
              day = day; // Oups! Assigns parameter day to itself!
              month = month; // Oups! Assigns parameter month to itself!
              year = year; // Oups! Assigns parameter year to itself!
        }
}
```

```
public class Date {  // (members hidden)
        int day, month, year;
        Date(int day, int month, int year) {
              this.day = day; // Ok!
              this.month = month; // Ok!
              this.year = year; // Ok!
        }
}
```

Implementation of Constructors – calling another Constructor

- We already discussed the DRY principle, in order to reuse code by the application of methods.
  - In Java, we can reuse, i.e. call, a classes ctor from within another ctor of the same class like so:

```
public class Date { // (members hidden)
    Date(int day, int month, int year) {
        // pass
    }

    // This ctor delegates its work to another ctor:
    Date() {
        this(1, 1, 1970);
        System.out.println("Just delegating...");
    }
}
```

```
Date date = new Date();
// >Just delegating...
date.print();
// >1.1.1970
```

- The syntax should be self-explanatory, but there are some peculiarities:
  - this() can only be used in ctors!
  - We can have any code in a delegating ctor, but calling another ctor, i.e. this(), must be the first statement in that ctor!

- Ctor overloading is a very important concept for users of our UDTs and also for delegation!

Unrestricted Access – A Problem rises!

- Let's assume the already defined UDT *Date* will be used like this:

```
Date myDate new Date(17, 10, 2012); // Ok, construct the Date.
myDate.month = 14; // Oups! Quattrodecember??
myDate.print();
// >17.14.2012
```

Definition
Encapsulation is the idea to have objects that allow interaction w/o detailed knowledge of their implementation details. The concept is expressed by restricted access to an object's data via a strictly defined interface, e.g. via getters/setters to enforce its invariant.

- What have we done?
  - We can freely access and modify all fields of the class *Date*! So far so good...
  - We can also set all the fields to invalid values as far as *Date*'s concept is concerned.
    - Following the concept of a date, which is abstracted by the UDT *Date*, there doesn't exist a month with the value 14!

- How can we fix that?
  - We have to restrict access to all fields, e.g. *month*, of the UDT in question. This is called encapsulation.
  - Encapsulation means, that the fields of an object can neither be written, nor read from "outside".
  - => We implement encapsulation by marking fields as private fields.

Hint
We've already encountered encapsulation when we discussed procedural programming. *printPrompt()* hides the implementation detail "*Scanner* as local variable" from its callers. -> Locals represent also a kind of encapsulation!

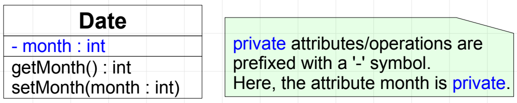
```
static int printPrompt(String promptText) {
        System.out.println("Please enter a number:");
        System.out.println(promptText);
        Scanner inputScanner = new Scanner(System.in);
        return inputScanner.nextInt();
}
```

- But, how can we then get or set the values of private fields?
  - We implement public get and set methods for each private field we want to access or manipulate from "outside".

Unrestricted Access – A Solution is in Sight!

- Now we'll encapsulate the field *month* of *Date* will be used like this:

```
public class Date { // (members hidden)
        private int month; // The field month is now private.
        /**
         * An accessible getter for the field month.
         */
        int getMonth() {
             return this.month;
        }
        /**
         * A accessible setter for the field month.
         */
        void setMonth(int month) {
             this.month = month;
        }
}
```


Date
- month : int
getMonth() : int
setMonth(month : int)

private attributes/operations are prefixed with a '-' symbol.
Here, the attribute month is private.

```
Date myDate new Date(17, 10, 2012); // Ok, construct the Date.
```

```
myDate.month = 14; // Invalid, won't compile! 'month' has only private access in Date
```

```
myDate.setMonth(14); // Oups! Again "quattrodecember"??
myDate.print();
// >17.14.2012
```

- What have we done?
  - We made *month* a private field of *Date*.
  - We added get and set methods to access and manipulate the field *month*.
  - Such methods are often called getter and setter or accessor and manipulator.
  - => With these changes, we can no longer directly access or manipulate *month*. It results in a compile time error!

- Hm, wait! – We can still set the field *month* to an invalid value! The field could still have the invalid value 14!
  - Ok, we missed something! – We have to add code to *setMonth()* in order to check the values to be set!

Unrestricted Access – A clever Setter!

- The idea to make our setter *setMonth()*, and thus *Date*, more stable, we've to add some code to our setter:

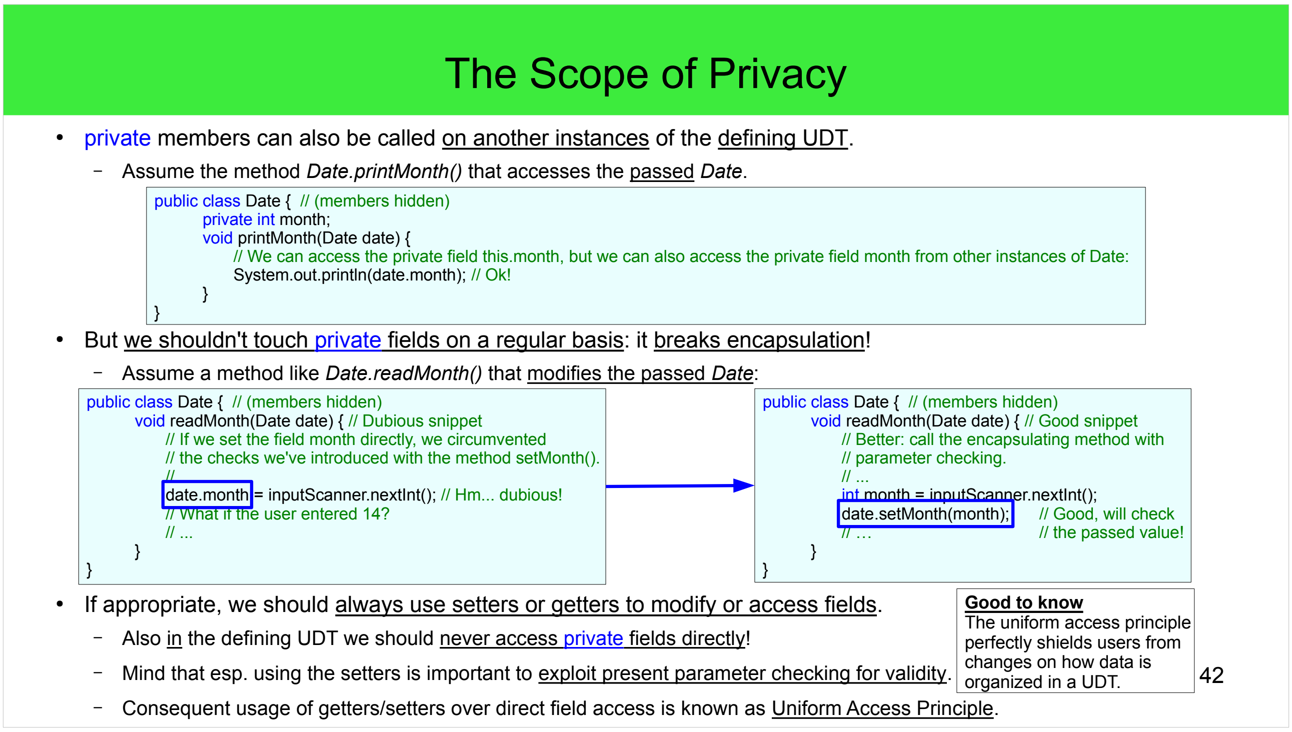
```
public class Date { // (members hidden)
        void setMonth(int month) {              // The setter checks the
             if (1 <= month && month <= 12) {   // validity of the value to be
                    this.month = month;         // set for month.
             }
        }
}
```

  - With that more clever setter, we'll be able to only set valid values for the month:

```
Date myDate = new Date(17, 10, 2012);       // Construct the Date.
myDate.setMonth(14);                        // Try to set quattrodecember.
myDate.print();                             // myDate remains 17.10.2012!
// >17.10.2012
```

- What have we reached effectively?
  - The field *month* can't be directly accessed/modified, but only via *getMonth()* and *setMonth()*.
  - Esp. *setMonth()* checks the validity of the *month*(-parameter) to be set. – In this implementation it ignores an invalid value to be set.
    - I.e. with this implementation of *setMonth()* the old, still valid value of *month* remains set.

- Mind, that we could add any other code we want to have into getters/setters!



The Scope of Privacy

- private members can also be called on another instances of the defining UDT.
  - Assume the method *Date.printMonth()* that accesses the passed *Date*.

```
public class Date { // (members hidden)
        private int month;
        void printMonth(Date date) {
              // We can access the private field this.month, but we can also access the private field month from other instances of Date:
              System.out.println(date.month); // Ok!
        }
}
```

- But we shouldn't touch private fields on a regular basis: it breaks encapsulation!
  - Assume a method like *Date.readMonth()* that modifies the passed *Date*:

```
public class Date { // (members hidden)
        void readMonth(Date date) { // Dubious snippet
              // If we set the field month directly, we circumvented
              // the checks we've introduced with the method setMonth().
              //
              date.month = inputScanner.nextInt(); // Hm... dubious!
              // What if the user entered 14?
              // ...
        }
}
```

```
public class Date { // (members hidden)
        void readMonth(Date date) { // Good snippet
              // Better: call the encapsulating method with
              // parameter checking.
              // ...
              int month = inputScanner.nextInt();
              date.setMonth(month);      // Good, will check
              // …                       // the passed value!
        }
}
```

- If appropriate, we should always use setters or getters to modify or access fields.
  - Also in the defining UDT we should never access private fields directly!
  - Mind that esp. using the setters is important to exploit present parameter checking for validity.
  - Consequent usage of getters/setters over direct field access is known as Uniform Access Principle.

Good to know
The uniform access principle perfectly shields users from changes on how data is organized in a UDT.

- A static method of the defining UDT can also 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!

Access Specifiers in Java

- Up to now, we've discussed two types of accessibility for members of a class: private and "non-private"

```
// <Date.java>
public class Date { // (members hidden)
        private int month; // The private field month.
        /**
         * An non-private getter for the field month.
         */
        int getMonth() {
             return this.month;
        }
        /**
         * A non-private setter for the field month.
         */
        void setMonth(int month) {
             this.month = month;
        }
}
```

```
// <Program.java>
public class Program {
        public static void main(String[] args) {
             Date myDate = new Date(17, 10, 2012); // Construct the Date.
             myDate.month = 4; // Invalid, private fields cannot be accessed.
             myDate.setMonth(14); // OK, we can access the non-private method.
        }
}
```

- All right, so we can access only non-private members from outside the class. But what means outside?

- A UDT's non-private members are accessible to other UDTs living in the same package.
  - Packages are Java's way to group multiple UDTs together, e.g. when they cover the same "topic" for programming.

- Now, we'll just have a glimpse over packages, we'll discuss them in depth in a future lecture.

Packages

- In short, packages are used to group UDTs logically and "shield" them from each other.

- We've already used the classes *Scanner* and *Arrays*, which reside in the package *java.util* (among a lot of other UDTs).
  - To make use of *Scanner* and *Arrays*, e.g. in our class *Program*, we had to explicitly import the package *java.util*:

```
// <Program.java>
import java.util.*; // Import the package java.util.

public class Program {
	// … use the types defined in java.util, e.g. Scanner
}
```

Good to know
Java's standard packages, i.e. those coming with JDK, carry the *java* prefix in their names.

- … but, why do have to import *java.util.* explicitly?
  - => *Scanner* and *Arrays* reside in another package than *Program*! The import makes the types in *java.util* "known" to *Program*.

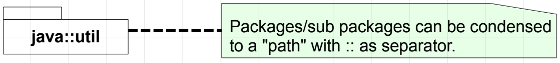
- … but, this yields another question: In which package does our UDT *Program* reside?

- We are going to answer this question along with a discussion of the consequences.

- We start discussing packages with the introduction of its UML notation to simplify further explanations.

Packages – UML Notations

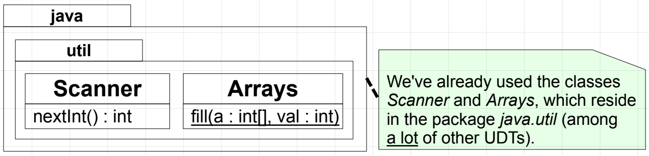
- A package can be represented as folder symbol, that shows the package name or fully qualified path within the box.



java::util

Packages/sub packages can be condensed to a "path" with :: as separator.

- We can also draw packages and sub packages as cascades symbols:
  - If a package symbol contains other symbols (classes or sub packages), the UML standard suggests to write package names into the label of the package folder symbol.



java

util

Scanner

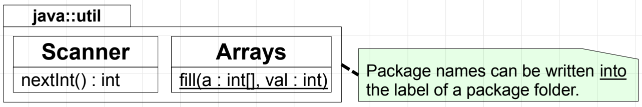
nextInt() : int

Arrays

fill(a : int[], val : int)

We've already used the classes *Scanner* and *Arrays*, which reside in the package *java.util* (among a lot of other UDTs).

- Alternatively, the cascade can be condensed showing the package path into the label of an "aggregating package":



java::util

Scanner

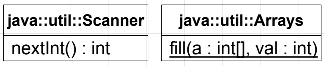
nextInt() : int

Arrays

fill(a : int[], val : int)

Package names can be written into the label of a package folder.

- And we can just use fully qualified classifier names as alternative to surrounding package boxes:



java::util::Scanner

nextInt() : int

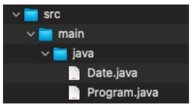
java::util::Arrays

fill(a : int[], val : int)

Packages – The default Package

- All right, but what is the situation of our UDTs *Program* and *Date*? In which package do they reside, if any?

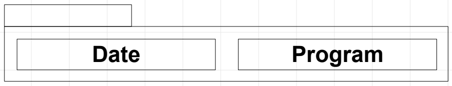
- The package structure must be aligned to the directory structure of the project in Java.
  - In our case, *Program* and *Date* reside in the same directory, which makes the top of our project.
  - Remember that <projectDirectory>/src/main/java contains the sources according Gradle standards:



```
src
  main
    java
      Date.java
      Program.java
```

- All types whose definition-files reside in the same directory are put into the same package.
  - Actually, a package-declaration must also be put into the java-files, but we'll discuss peculiarities about packages in a future lecture.
  - Packages must be reflected by equally named sub-directory-structures of the project's top-directory.

- So, *Program* and *Date* reside in the same directory, thus in the same package, but this package has no name.
  - Additionally this common directory is the top-directory of our project.
  - When UDTs reside in the top-directory which implies having no package name the types are said to reside in the default package.



Date **Program**

Packages – package-private Access

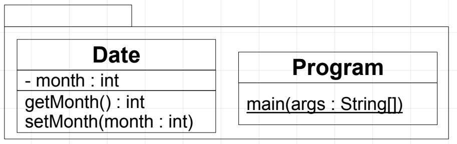
- So, *Program* and *Date* reside in the same package, in the default package!

- The important point is that all non-private members in the enclosing package can access and modify each other.
  - Because *Program* and *Date* reside in the same package, these UDTs can access each-others non-private members accordingly:

```
// <Date.java>
public class Date { // (members hidden)
        /**
         * A non-private setter for the field month.
         */
        void setMonth(int month) {
              this.month = month;
        }
}
```

```
// <Program.java>
public class Program {
        public static void main(String[] args) {
              Date myDate = new Date(17, 10, 2012); // Construct the Date.
              myDate.setMonth(14); // OK, we can access the non-private method.
        }
}
```

- When we just leave away any access modifier on a member definition it is said to have package-private access.
  - I.e. *Date.setMonth()* is a package-private method.
  - We can still not access private members within the package, they are only accessible within the same class.



Date

- month : int

getMonth() : int

setMonth(month : int)

Program

main(args : String[])

Importing Packages

- To use UDTs, which are defined in other packages, we can just fully qualify the name of the UDT:

```
java.util.Scanner inputScanner = new java.util.Scanner(System.in);
int number = inputScanner.nextInt();
```

  - Fully qualifying a name just means, that we write a name and prefix it with the package name, in which it is residing.
  - *java.util* is the package name and *Scanner* is the name of the UDT we want to use.

- Alternatively, we can import the complete package into *Program*, this makes all UDTs in a package directly visible:

```
// <Program.java>
import java.util.*;

public class Program {
        public static void main(String[] args) {
              Scanner inputScanner = new Scanner(System.in);
              int number = inputScanner.nextInt();
        }
}
```

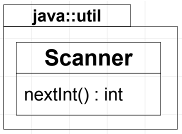
  - This "*-import" would allow us to use *Arrays*, which is also defined in *java.util*, without any qualification.

- Alternatively, we can import only a specific UDT, by just importing a fully qualified UDT name:

```
// <Program.java>
import java.util.Scanner;

// pass
```

The "public" Access Specifier – Part 1

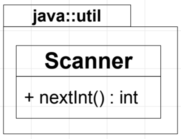
- If non-private members of a UDT are only visible in its enclosing package, how can we call methods from other packages?



```
Scanner inputScanner = new Scanner(System.in);
// Why can we access Scanner.nextInt()?
int number = inputScanner.nextInt();
```

- The truth is, that Java provides another access modifier for members to be accessible to other packages: public access.
  - So, the method *Scanner.nextInt()* is a public method:

```
package java.util;

public class Scanner {
        public int nextInt() {
        // pass
        }
}
```



public attributes/operations are prefixed with a '+' symbol. Here, the operation *nextInt()* is public.

  - As can be seen, the access modifier public is just put in front of a member's name, as we did with the private access modifier.

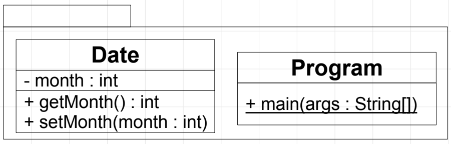
- What we can take away: public members are accessible from inside the enclosing package and other packages.
  - … and public members are also accessible from inside the defining class.

- Oh, and of course, the reason why we declare public classes all the time is to make them useable in other packages!

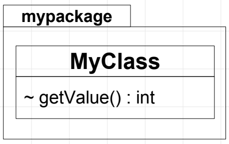
The "public" Access Specifier – Part 2

- Put simple, we can improve *Date* a lot, i.e. make it reusable from outside its package, by making its methods public.

```
// <Date.java>
public class Date { // (members hidden)
        private int month; // The methods are now public.
        /**
         * An public getter for the field month.
         */
        public int getMonth() {
            return this.month;
        }
        /**
         * A public setter for the field month.
         */
        public void setMonth(int month) {
            this.month = month;
        }
}
```



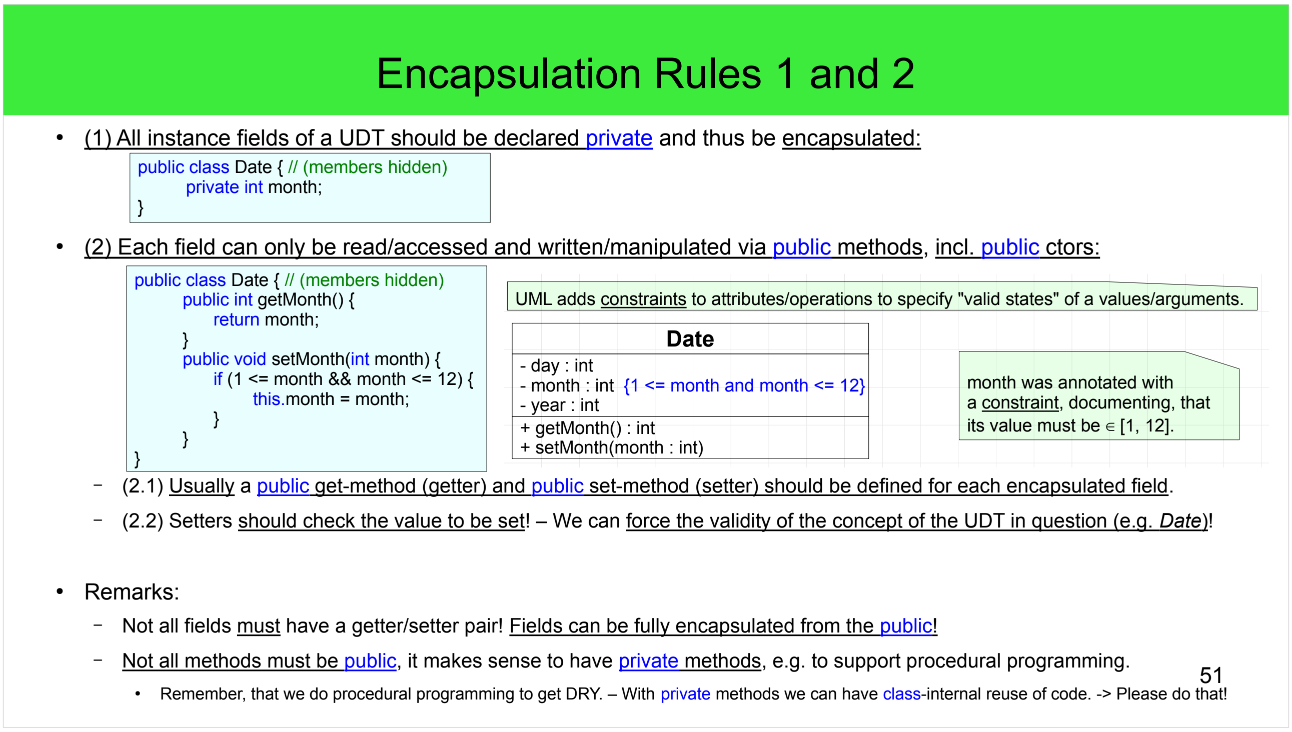
Good to know



The UML also allows to notate methods having package private access. Therefor the '~' symbol is used.

- However, after changing *Date*'s methods to be public, still the same accessibility rules hold true for *Program*:

```
// <Program.java>
public class Program {
        public static void main(String[] args) {
            Date myDate = new Date(17, 10, 2012); // Construct the Date.
            myDate.month = 4; // Invalid, private fields can still not be accessed.
            myDate.setMonth(14); // OK, we can access the public method.
        }
}
```



Encapsulation Rules 1 and 2

- (1) All instance fields of a UDT should be declared private and thus be encapsulated:

```
public class Date { // (members hidden)
        private int month;
}
```

- (2) Each field can only be read/accessed and written/manipulated via public methods, incl. public ctors:

```
public class Date { // (members hidden)
        public int getMonth() {
              return month;
        }
        public void setMonth(int month) {
              if (1 <= month && month <= 12) {
                     this.month = month;
              }
        }
}
```

UML adds constraints to attributes/operations to specify "valid states" of a values/arguments.

```
Date
- day : int
- month : int  {1 <= month and month <= 12}
- year : int
+ getMonth() : int
+ setMonth(month : int)
```

month was annotated with a constraint, documenting, that its value must be ∊ [1, 12].

  - (2.1) Usually a public get-method (getter) and public set-method (setter) should be defined for each encapsulated field.
  - (2.2) Setters should check the value to be set! – We can force the validity of the concept of the UDT in question (e.g. *Date*)!

- Remarks:
  - Not all fields must have a getter/setter pair! Fields can be fully encapsulated from the public!
  - Not all methods must be public, it makes sense to have private methods, e.g. to support procedural programming.
    - Remember, that we do procedural programming to get DRY. – With private methods we can have class-internal reuse of code. -> Please do that!

- The definition of private ctors is also possible!

Encapsulation Rule 3

- (3) The validity of values for a field should be documented on the belonging to setter and getter!
  - Documentation is required, so that *Date*'s methods can be used by the public/3rd party developers!

```
public class Date { // (members hidden)
        /**
         * Sets the month of this date. The argument must be within
         * [1, 12]. If not, the old, but still valid value will be kept.
         *
         * @param month the new month of this date
         */
        public void setMonth(int month) { /* pass */ }
}
```

  - (3.1) Mind, that a setter shows a certain behavior, in case of trying to set invalid values! – Which behavior does it show?
  - (3.2) For completion, also the getter should be documented:

```
public class Date { // (members hidden)
        /**
         * Retrieves the month of this Date instance. The month is a value within [1, 12].
         *
         * @return the month
         */
        public int getMonth() { /* pass */ }
}
```

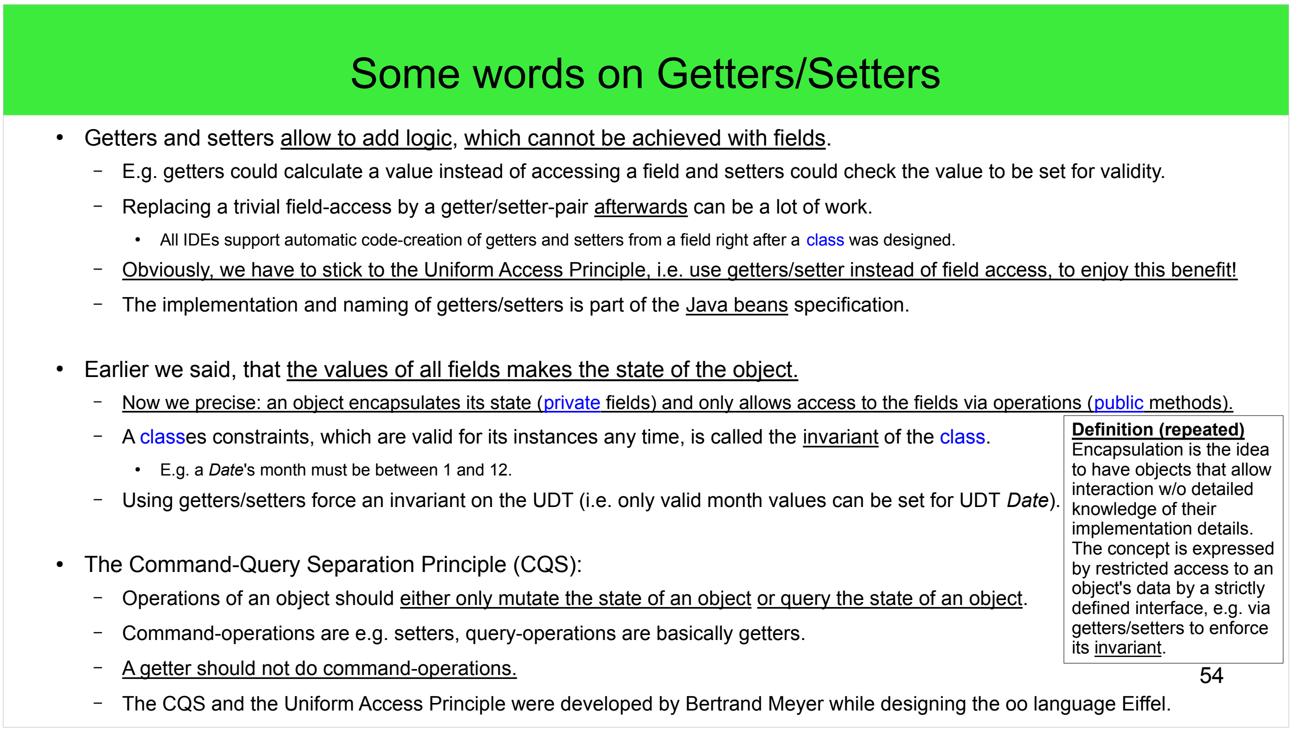
  - (3.3) Actually, all non-private methods, i.e. package private access and public, should be documented.

Excursus: Getters/Setters for boolean Fields

- If appropriate, getters encapsulating boolean fields should be named specially for better readability.
  - They should have method names starting with *is* or *has*.
  - If appropriate also the field in question can be prefixed with *is* or *has*.

```
public class Examples {
        private boolean isValid
        private boolean hasLicense;

        public boolean hasLicense() {
            return this.hasLicense;
        }
        public boolean setLicense(boolean hasLicense) {
            this.hasLicense = hasLicense;
        }
        public boolean isValid() {
            return this.isValid;
        }
        public boolean setValid(boolean isValid) {
            this.isValid = isValid;
        }
}
```

- In most cases boolean setters can be named like other setters.



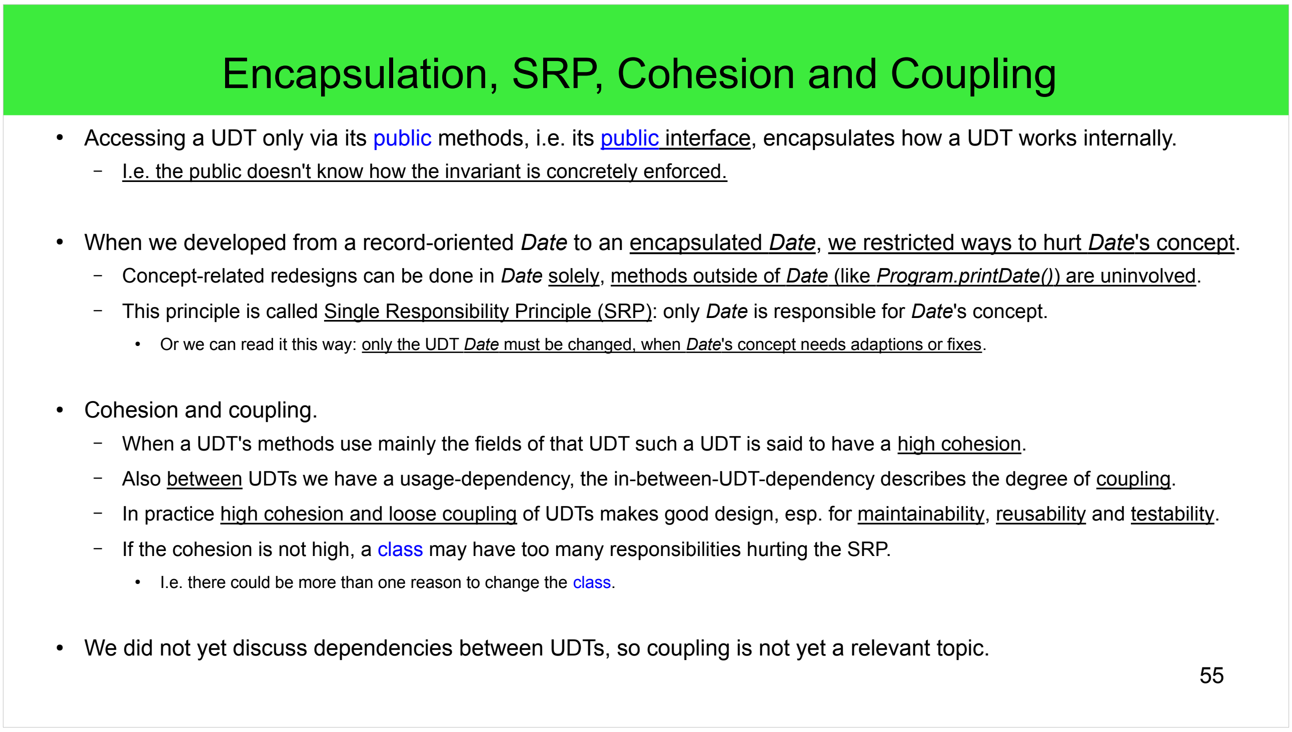
Some words on Getters/Setters

- Getters and setters allow to add logic, which cannot be achieved with fields.
  - E.g. getters could calculate a value instead of accessing a field and setters could check the value to be set for validity.
  - Replacing a trivial field-access by a getter/setter-pair afterwards can be a lot of work.
    - All IDEs support automatic code-creation of getters and setters from a field right after a class was designed.
  - Obviously, we have to stick to the Uniform Access Principle, i.e. use getters/setter instead of field access, to enjoy this benefit!
  - The implementation and naming of getters/setters is part of the Java beans specification.
- Earlier we said, that the values of all fields makes the state of the object.
  - Now we precise: an object encapsulates its state (private fields) and only allows access to the fields via operations (public methods).
  - A classes constraints, which are valid for its instances any time, is called the invariant of the class.
    - E.g. a *Date*'s month must be between 1 and 12.
  - Using getters/setters force an invariant on the UDT (i.e. only valid month values can be set for UDT *Date*).
- The Command-Query Separation Principle (CQS):
  - Operations of an object should either only mutate the state of an object or query the state of an object.
  - Command-operations are e.g. setters, query-operations are basically getters.
  - A getter should not do command-operations.
  - The CQS and the Uniform Access Principle were developed by Bertrand Meyer while designing the oo language Eiffel.

Definition (repeated)
Encapsulation is the idea to have objects that allow interaction w/o detailed knowledge of their implementation details. The concept is expressed by restricted access to an object's data by a strictly defined interface, e.g. via getters/setters to enforce its invariant.

- Alternatively, there exists the Java library lombok, which enables automatic synthesis of getters/setters.

```
// <Date.java>
import lombok.Getter;

public class Date {
        @Getter
        int month;
}
```



Encapsulation, SRP, Cohesion and Coupling

- Accessing a UDT only via its public methods, i.e. its public interface, encapsulates how a UDT works internally.
  - I.e. the public doesn't know how the invariant is concretely enforced.
- When we developed from a record-oriented *Date* to an encapsulated *Date*, we restricted ways to hurt *Date*'s concept.
  - Concept-related redesigns can be done in *Date* solely, methods outside of *Date* (like *Program.printDate()*) are uninvolved.
  - This principle is called Single Responsibility Principle (SRP): only *Date* is responsible for *Date*'s concept.
    - Or we can read it this way: only the UDT *Date* must be changed, when *Date*'s concept needs adaptions or fixes.
- Cohesion and coupling.
  - When a UDT's methods use mainly the fields of that UDT such a UDT is said to have a high cohesion.
  - Also between UDTs we have a usage-dependency, the in-between-UDT-dependency describes the degree of coupling.
  - In practice high cohesion and loose coupling of UDTs makes good design, esp. for maintainability, reusability and testability.
  - If the cohesion is not high, a class may have too many responsibilities hurting the SRP.
    - I.e. there could be more than one reason to change the class.
- We did not yet discuss dependencies between UDTs, so coupling is not yet a relevant topic.

- An extreme case of low cohesion can be a UDT that parses dates and URLs. It has two responsibilities for which developers have to care for (maintenance). – Usually such a UDT will end up fulfilling only one responsibility "good enough".
- An extreme case of high cohesion can be a UDT, which only does one very primitive thing, e.g. parse the punctuation of a URL. In this case one could say that such a UDT is so primitive that it doesn't represent a meaningful concept.
- From a practical standpoint, a high coupling means that a change in one UDT also needs changes in another UDT.

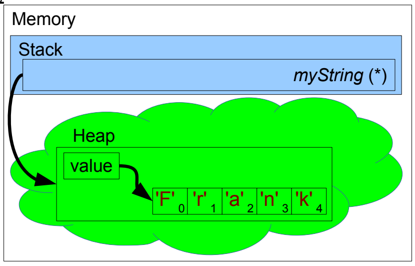
Example: String encapsulates char Array and allows public Access

- In a past lecture we've discussed, that in Java a *String* encapsulates a char[]. *String* is a UDT, i.e. a class.

- When we inspect *String*'s implementation, we'll see code like this (simplified):

```
public class String { // Simplified implementation of String
        // The value is used for character storage:
        private char value[];
        /**
         * Returns the char value at the
         * specified index.
         */
        public char charAt(int index) {
             return value[index];
        }
        // pass
}
```

```
String myString = "Frank";
```



  - The most important aspect in *String*'s definition is the private field *value*, which encapsulates *String*'s internal representation as char[].
  - This array is the central data, the essence of a *String* instance.

- The power of the UDT *String* is to put a set of useful public methods "around" the encapsulated char[], e.g. *String.charAt()*:

```
char firstCharacter = myString.charAt(0);
// firstCharacter = 'F'
```

  - There is one important point regarding *String*: *String* has no public setters! – I.e. a *String* object cannot be (publicly) modified

Naming Conventions for abstracted Types

- Following naming conventions should be used in future.

- Names for UDTs:
  - PascalCase
  - Don't use prefixes! (Like the prefixes 'C' for classes or 'E' for enums.)

- Names for methods:
  - camelCase

- Names for fields:
  - camelCase, no prefixes like '_'.
  - Don't use prefixes to denote a field's type! (Like the prefix 'i' for an int field.)

Thank you!