Java-programming

Working with Java SE

Language and Classlibrary basics

Today

- We'll start with a recap of pre-studies

But as the main topic

- You will learn to use date- and time information and present and query data for different locales

Java Environment

JRE, JDK

Versions

Java Standard Edition

- Originally developed by Sun Microsystems
  - Currently ”owned” by Oracle
- Standard Edition defines the basic operating environment for Java
  - Runtime Environment, JRE, provides the virtual machine and basic class libraries needed for running the Java applications
  - Development environment, JDK or Java SDK, provides basic development tools
- Features
  - Platform independent, WORA
  - Desktop applications
  - Database connectivity
  - Remote calls, RMI
  - I/O-operations
  - Network communications
  - XML-processing

Versions

- JDK Alpha and Beta (1995)
- JDK 1.0 (January 23, 1996)
- JDK 1.1 (February 19, 1997)
- J2SE 1.2 (December 8, 1998)
  - Java2
- J2SE 1.3 (May 8, 2000)
- J2SE 1.4 (February 6, 2002)

. . .

- **Java SE 8 (March 18, 2014)**

LTS

. . .

- **Java SE 11 (September, 2018)**

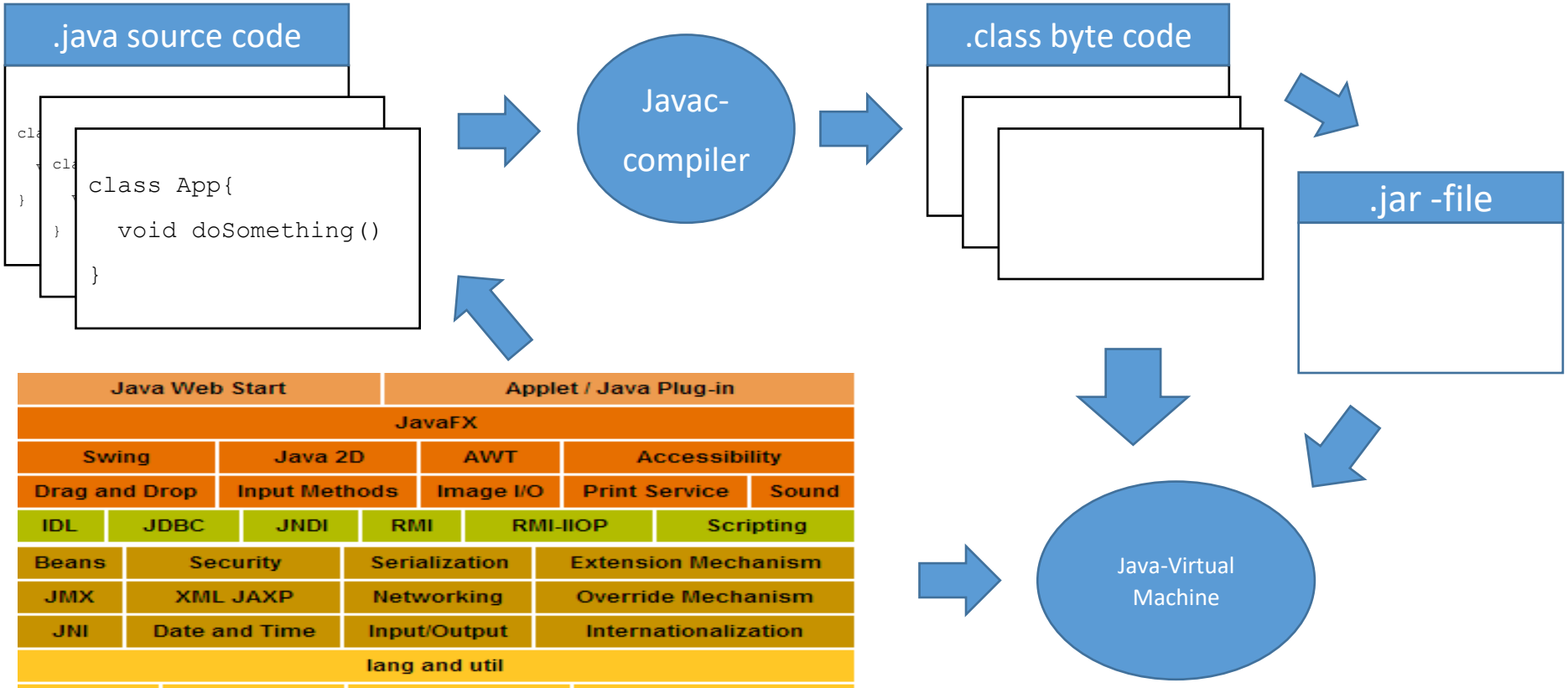
. . .

- **Java SE 17 (September 2021)**

. . .

- Java SE 19, (September 2022)

Simple build process



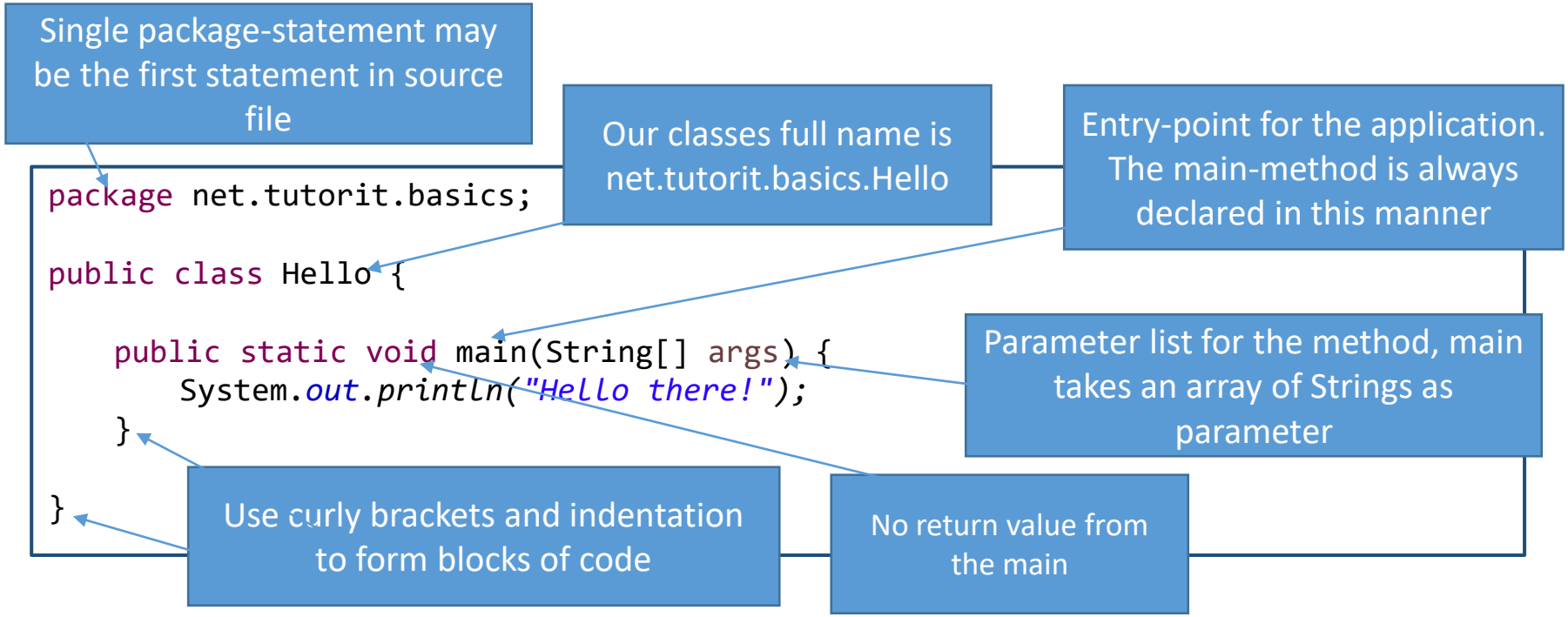
Java Web Start	Applet / Java Plug-in				
JavaFX					
Swing	Java 2D	AWT	Accessibility		
Drag and Drop	Input Methods	Image I/O	Print Service	Sound	
IDL	JDBC	JNDI	RMI	RMI-IIOP	Scripting
Beans	Security	Serialization	Extension Mechanism		
JMX	XML JAXP	Networking	Override Mechanism		
JNI	Date and Time	Input/Output	Internationalization		
lang and util					
Math	Collections	Ref Objects	Regular Expressions		
Logging	Management	Instrumentation	Concurrency Utilities		
Reflection	Versioning	Preferences API	JAR	Zip	

Class libraries of JRE are needed at development and runtime

.class -files may be distributed and run by themselves. Often a single zipped jar-file containing all .class files is created.

Hello world!

- The source code is written to .java –files
- Source files contain one or more classess and interfaces
- If class or interface is declared public then the name of the file must match the name of the class/interface
  - So only one public class in a file but perhaps several non public classes



Package

- Package defines a namespace
  - Classes defined within one package must have unique names
- In file system packages are seen as subdirectories
  - Classes declared to net.tutorit.app –package are stored into/net/tutorit/app -directory
- Packages help you organize your source code
  - net.tutorit.ui, ui-related classes
  - net.tutorit.bl, business logic classes
  - net.tutorit.da, data-access classes
  - net.tutorit.util, general utility classes
- Class libraries are also organized into packages
  - java.lang –package doesn't require import
  - java.util contains many utility classess and collections
  - Java.time contains the currently preferred classes for data/time -manipulation
- When you use classes from other packages you need an import statement
  - Or you must use fully qualified class name, for example java.util.Date
- Only public classes may be referred outside the package

import

- Import –statement tells the compiler to use classes from other packages
  - Use all classes from a package : import java.time.*;
  - Use specified class from a package: import java.time.LocalDate;
- In most cases the latter version is preferred
  - There is no difference in execution speed nor application size
  - Compilation might be slightly faster
  - There is no possibility of name conflict
  - Foreign classes that are used, are neatly documented at the beginning of the source code file
- Special case: import static
  - Static members of imported class can be used without being prefixed by the class name

```
package net.tutorit.basics;

import java.time.LocalDate;

public class Hello {


    public static void main(String[] args) {
        System.out.println("Hello there!");
        System.out.println("Today is:"+LocalDate.now());
    }

}
```

This file uses LocalDate-class from java.time -package

Returns the current date

Concatenate current date to a string

Import static

- java.lang.Math-class provides methods for typical numeric operations
  - Logarithms, trigonometric-functions etc
  - They are all declared with static modifier

Sample of import static

```
package net.tutorit;

import java.util.Scanner;
import static java.lang.Math.*;

public class Application {

    public static void main(String[] args) {
        Scanner scanner=new Scanner(System.in);
        System.out.print("Enter x:");
        int x=scanner.nextInt();
        System.out.print("Enter y:");
        int y=scanner.nextInt();
        System.out.println("Distance from origin="+sqrt(x*x+y*y));
        // Without static import you would need to;
        System.out.println("Distance from origin="+Math.sqrt(x*x+y*y));
    }

}
```

All methods of Math-class can be called without class name

Class-definition

All implementations go to a class

- Public classes may be used outside of their package
- Unless imported the fully qualified class name must be used when referring to “foreign” classes
  - Fully qualified class name consists of package name and the class name
  - net.tutorit.basics.Hello or java.time.LocalDate
- Class may also be declared final
  - Prohibits extending, inheritance
- Class members surrounded by {….}
  - Fields, member variables
  - Methods, member functions

Datatypes and variables

- Java has primitive types and object types
- Variables of object type are references to the object of that type or null if the reference is not set
  - Currently actual addresses of objects
- Variables must always be initialized before use
  - Member variables (fields) have default values
- Null-value for object references should always be tested before use
- Parameters are always passed by value
- Type cast must always be used when loss of precision is possible

Data Type	Size	Default Value	Object Wrapper
byte	8 bits	0	Byte
short	2 bytes	0	Short
int	4 bytes	0	Integer
long	8 bytes	0L	Long
float	4 bytes	0.0f	Float
double	8 bytes	0.0d	Double
char	16 bits	'\u0000'	Char
boolean	1 byte	false	Boolean
Object-types	32 or 64 bits	null	

```
int i=32;
long l=32L;
double d=3.14;
float f=3.14f;
String s="Hello";
System.out.println(String.format(
        "%d, %d, %f, %f, %s", i,l,d,f,s));
```

String is an object type though it in many occasions behaves like a primitive type.

```
// Type casts:
double d=4.23;
int i=(int)d;
// Strings and wrappers
int i=Integer.parseInt("123");
String s="Hello "+i;
```

Arithmetic operators

- Basic arithmetic operators +,-,* and / are available
  - Precedence is typical, multiplications and divisions are evaluated first
  - In complex calculations use brackets

```
int a=1,b=2;
int c=a+b;

a=a+b;

a+=b;

a++;

a=9%2;

c=(a+b)*c;
```

Declare variables a and b with single statement.
Declare variable c having initial value of 3.

Variable a gets a new value which is 3

Variable a is incremented by the value of b, a becomes 5

Similar notations are available for -,* and /

Variable a is incremented by 1

%-sign is the “remainder”-operator, variable a gets value of 1

a+b is evaluated first the sum is multiplied by c

Conditionals and logical operators

- If – else
  - Condition must always evaluate to bool –value
- Switch – case
  - Primitive types, string or enums
- Also ? : operator is available

```
int a=3;
String s="Hello";

if (!((a!=3) && (s.equals("Hello")))){
    System.out.println("Yes, it is true");
}
else System.out.println("Not this");

s = (a==3 ? "Hi" : "Hello");

switch(s)
{
    case "Hi":
        System.out.println("Hi, how are you");
        break;
    case "Hello":
        System.out.println("Hello there");
        break;
    default:
        System.out.println("Value not found");
        break;
}

if (!s.equals("Hello World")) System.out.println("No it is not");
```

Operator	Meaning
==	Equals
!=	Not equals
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
&&	Logical and
||	Logical or
!	Logical not

Equality of object contents must be tested with equals-method. The == operator tests whether the references refer to the same object.

Strings and wrappers

- String is designed to be used for constant strings (immutable class)
  - Instantiation happens rarely
  - Same constant string is kept in memory and referred to from many places
  - Avoid doing String manipulation with Strings in a loop
- StringBuffer class is meant for variable-length strings
  - When string is changed in a loop StringBuffer offers MAJOR performance increase
- StringBuilder is even faster if you don't need synchronization
- And both of them will work better if you set initial size for them

- In fact the wrapper classes for numeric values have similar implementation to String

```
Integer i=3;
Integer j=3;
System.out.println(i==j); // true

j=new Integer(3);
System.out.println(i==j); // false
```

Compiler sees the single Integer constant 3 and both i and j refer to object that holds that value

j refers to an object that is explicitly instantiated, therefore I and j no longer refer to the same object

Arrays and ArrayList

- Arrays are constant size collections of variables
  - For object types the reference to the object is stored into the array
  - Arrays them selves are object types
  - Each array has length-member
- ArrayList is a dynamic collection class from from java.util -package
  - Items may be added, inserted, deleted, sorted etc

```
int[] monthLengths={31,28,31,30,31,30,31,31,30,31,30,31};

String[] monthNames=new String[12];
monthNames[0]="January";
monthNames[1]="Fegruary";
monthNames[2]="March";
// ETC
monthNames[11]="December";

ArrayList<String> weekDays=new ArrayList();
weekDays.add("Monday");
weekDays.add("Tuesday");
weekDays.add("Thursday");
weekDays.add(2,"Wednesday");
weekDays.add("Friday");
weekDays.add("Saturday");
weekDays.add("Sunday");

System.out.println(monthNames[3]);
System.out.println(monthNames[monthNames.length-1]);
System.out.println(weekDays.get(2));
System.out.println(weekDays.get(weekDays.size()-1));
```

Zero-based indexes!

Initialized array of 12 integers

Uninitialized array of 12 strings, values are populated later

Import java.util.ArrayList to create a new ArrayList

Add items

Insert item into index 2

```
April
December
Wednesday
Sunday
```

Loops

- For-loop
  - Traditional and ”for each” –notation
- While
- Do-while
- Break and continue

```
// These are identical
for (int i=0;i<5;i++) {
     System.out.println("Val "+i);
}

int i=0;
while(i<5) {
     System.out.println("Val "+i);
     i++;
}
```

Any expression evaluating into boolean may be used

```
int counter=0;
while(true){
     if (monthNames[counter].equals("March")) break;
    counter++;
}
System.out.println("March has the index of "+counter);

System.out.println("Months that have 31 days: ");
for(int i=0;i<monthLengths.length;i++){
    if (monthLengths[i]!=31) continue;
    System.out.println(monthNames[i]);
}

for (String wd:weekDays) System.out.println(wd);

for(String mn : monthNames){
    System.out.println(mn);
}
```

Interrupt looping

Ignore rest of iteration

Looping collections and arrays

Exercise

Create a number guessing game as a console application

- Generate a random number between 1-100

```
int value=(int)(Math.random()*100+1);
```

- Loop until the user guesses the number
  - You may read user input for example with a Scanner object (java.util-package)

```
Scanner sc=new Scanner(System.in);
String input=sc.next();
int i=Integer.parseInt(input);
```

  - After each guess show whether the guess was too big or small
- Finally display how many guesses it took

No, let's go through the implementation for last week's checkpoint instead.

Catching exceptions

- If something goes wrong in the execution of your code an Exception is thrown
  - Exceptions may (and should) be caught in your code
- Exceptions are identified by their type
  - Class library provides you with huge number of exceptions
  - Documentations states which may be caused by different methods (IDE knows this also)
  - You may declare your own Exception types also (a class that extends Exception)
- You try to do something that may cause exceptions
  - Try block is followed by one or more catch-blocks, each processing specific type of exceptions
  - Last catch may just catch Exception, all exceptions listed before are caught
  - Optionally you may add finally block which is executed whether or not an exception is thrown in try block

```
String userInput="Tom,18"; // Name and age, comma separated
String[] arr=userInput.split(","); // Split the input into an array
try {
    int age=Integer.parseInt(arr[1]);  // Convert second item into int
}
catch(NumberFormatException ex) {
    System.out.println("Second item was not a number");
}
catch(Exception ex) {
    System.out.println("Something went wrong");
}
finally {
    System.out.println("Ready to continue");
}
```

Dates

Getting started

Dates

- Originally we just had java.util.Date
  - Now mostly deprecated
  - Doesn't support leap-second
- Then we had java.util.Calendar(s)
  - Especially GregorianCalendar
  - Somewhat clumsy to use
- Since Java 8 we have DateTime API in java.time-package
  - Immutable value clasesses (as Strings and wrappers)
  - LocalDate, LocalTime and LocalDateTime
  - But also zoned- and offset-versions

Of course every programmer needs to know that Gregorian calendar is the most widely used calendar today

Sadly we currently live in a mess of all these concepts and when we move to databases we will also encounter java.sql.Date

Some LocalDate(Time) samples

- All LocalXXX-classes
  - Hold quite a few static methods for creating objects
  - Provide methods for calculations
  - Hold constants that help forming objects
- Remember that objects are immutable
  - Each calculation always creates a new object

```
Period per=Period.ofDays(8);
LocalDateTime dt2=dt.plus(per).minusMonths(3)
    .withHour(12).withMinute(0).withSecond(0).withNano(0);

LocalTime tm=LocalTime.NOON;
LocalTime tm2=LocalTime.of(8, 25);

LocalDate dt3=LocalDateTime
    .ofInstant(new Date().toInstant(),ZoneId.of("Europe/Helsinki"))
    .plusHours(8)
    .toLocalDate();
```

Some classes of java.time-package

Class	Description
LocalDate	a date, without time of day, offset or zone
LocalTime	the time of day, without date, offset or zone
LocalDateTime	the date and time, without offset or zone
OffsetDate	a date with an offset such as +02:00, without time of day or zone
OffsetTime	the time of day with an offset such as +02:00, without date or zone
OffsetDateTime	the date and time with an offset such as +02:00, without a zone
ZonedDateTime	the date and time with a time zone and offset
YearMonth	a year and month
MonthDay	month and day
Year/MonthOfDay/DayOfWeek/...	classes for the important fields
DateTimeFields	stores a map of field-value pairs which may be invalid
Calendrical	access to the low-level API
Period	a descriptive amount of time, such as "2 months and 3 days"

Timezones

- A timezone is a set of rules, corresponding to a region where the standard time is the same, there are about 40 of them.
  - Each timezone has an offset from UTC, so it's time moves in sync with UTC but by some difference.
  - A time-zone offset is the period of time that a time-zone differs from Greenwich/UTC.
  - Unfortunately the defined offset from UTC can vary due to changes in the rules.
- Timezones have two identifiers: abbreviated, eg "PLT", and longer, eg: "Asia/Karachi".
- Offsets can be resolved from a timezone at a particular point in time.
  - For example in summer Paris is Greenwich/UTC + one hour.
  - In winter it's Greenwich/UTC + two hours.

```
ZonedDateTime zdt=ZonedDateTime.now();
ZonedDateTime zdt2=ZonedDateTime.now(ZoneId.of("Europe/Paris"));
ZonedDateTime zdt3=ZonedDateTime.now(ZoneId.of(ZoneId.SHORT_IDS.get("PST")));
System.out.println(zdt);
System.out.println(zdt2);
System.out.println(zdt3);
------------------------------------------------------------------------
2018-02-03T18:51:46.789+02:00[Europe/Helsinki]
2018-02-03T17:51:46.790+01:00[Europe/Paris]
2018-02-03T08:51:46.790-08:00[America/Los_Angeles]
```

Exercise

- Study the api documentation different classes for using dates & time
- Create Techniques-project (we will use this for other techniques too)
  - With dates-package that holds DateTests-class
  - Implement the following to separate functions withing DateTests-class
  - UseDateTests –class from main
- Use date & time apis to instantiate and print out current date, then current time.
- Create a Duration object to hold duration of 5 weeks (35 days) and print it.
- Grab your current date, convert it to LocalDateTime (with time part set to 9am), then add previous duration of 5 weeks to it. What day is it?
- Use the api to figure out how many days there is to Christmas this year?
- How many days until the next Friday the 13th?

Exercise

- Create a new project BookApp
- Add class Book
  - String title
  - String author
  - double price
  - int numCopies
  - YearMonth published

We'll continue with this later on.

We will implement BookStore that holds array Books.

Books can be purchased from the BookStore and listings of Books may be queried based on different criterias.

When-ever you have extra time you can come back to this project and try to work with it perhaps peeking forward in the material

- Ensure that published is in past, if not throw an exception

- Test your class in the main

Localization

… or internationalization, what ever …

Localization

- Java.util.Locale represents a language region requiring specific formattings
  - Three versions of constructors taking
    - Language ISO-code: ”fi”, ”en”
    - Country ISO-code: ”FI”, ”US”, ”GB”
    - Possibly variant of the language
  - Also holds predifined constants for some locales
    - Locale.ENGLISH

```
Locale fi=new Locale("fi");
Locale en=Locale.ENGLISH;
// Many localization features take locale
// as parameter, but you may also
Locale.setDefault(fi);
```

Numbers and Currency

- java.text-package is a good starting point for localization
  - NumberFormat
  - Collator
  - MessageFormat

```
Locale lf = new Locale("fi","FI");
NumberFormat nf = NumberFormat.getNumberInstance(lf);
System.out.println(nf.format(-234.567));

// Numbers in US locale
nf = NumberFormat.getNumberInstance(Locale.US);
System.out.println(nf.format(-234.567));

// Similarly with FI currency
nf = NumberFormat.getCurrencyInstance(lf);
System.out.println(nf.format(-234.567));

// Similarly with US currency
nf = NumberFormat.getCurrencyInstance(Locale.US);
System.out.println(nf.format(-234.567));
```

Date Formatting

- Java.text –package also holds DateFormat-interface and SimpleDateFormat-class for formatting and parsing java.util.Dates
- Of course java.time.LocalXXX-classes require use of package java.time.format

```
String s1 = DateFormat.getDateInstance(DateFormat.SHORT,Locale.US).format(d1);
DateFormat df=new SimpleDateFormat("dd.MM.yyyy");
System.out.println(df.format(new Date()));
try {
    Date dt=df.parse("24.12.2017");
}
catch(ParseException ex) {}


LocalDate currentDate=LocalDate.now();
DateTimeFormatter formatter=DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
String tx=currentDate.format(formatter);
LocalDate parsed=LocalDate.parse("24.12.2017",formatter);
```

Localization: Resources

- All string-constants that are displayed to the user should be placed into properties files
  - One default-file, for example “Translations.properties”
  - One file for each locale “Translations_fi.properties” and “Translations_en.properties”
- Properties-files are text-files containing key-value-pairs
  - Key is the name of the variable, value the actual string presented to the user
  - All properties files must declare same keys in format:
    variable_name=The actual value for the parameter

Together these files form a “Resource bundle”.

Place the files into a “package” in your source code

No spaces or dots allowed

Basically any string

key	Translations.properties	Translations_fi.properties	Translations_en.properties
Book_title	BookTitle	Nimi	Title
Book_author	BookAuthor	Kirjailija	Author

Using resource bundle

- Basically you just
  - Load the bundle for current or specific locale
  - Load strings from the bundle
- The loaded string may contain parameter insertion slots
  - MessageFormat-class provides utilities to work with parameters

```
ResourceBundle b=ResourceBundle.getBundle("Messages");
String s=b.getString("message");

// s="{0}, today is {1}"
Object[] pars={"Hello", new Date()};
MessageFormat msg=new MessageFormat(s);
msg.setFormatByArgumentIndex(1, new SimpleDateFormat("dd.MM.yyyy"));
String formatted=msg.format(pars);
```

Exercise

- Continue with Techniques-project
- Work with Locales
  - Format numbers, currencies and dates to different locales
  - Again separate function(s) do DateTests-class

- Let’s assume our company has offices at Tokio, Helsinki and New York
  - Ask user for his/her city
  - Ask for date and time
  - Display that information for each for each office in localized manner

- Create a ResourceBundle ”Messages” with english and finnish translations
- Make one of the translations hold parameter slots
  - Work with MessageFormat

So today

- You should have gained familiarity working with different classes to manipulate date and time information and also learned to understand the complexities related to different time-zones

Generic programming

Generics

Reflection

Annotations

Something extra if we have time, if not we'll come back to this later

Java Generics

- With generics you may create constructs that use abstract datatypes on definition
  - The type is set when the construct is used
  - The abstract type must be object-type
- You may implement
  - Generic methods
  - Generic classes
- When you use generics you gain
  - Type-safety, compiler can check that you use correct consistently
  - Avoid using type casting extensively

```
class TradClass{
    Object item;

    TradClass(Object o){
        item=o;
    }

    public Object getItem() {
        return item;
    }
}
```

```
TradClass tc1=new TradClass("Hello");
TradClass tc2=new TradClass(new Car("V70"));
String s1=(String)tc1.getItem();
Car c1=(Car)tc2.getItem();

GenClass<String> gc1=new GenClass<>("Hello");
GenClass<Car> gc2=new GenClass<>(new Car("V70"));
String s2=gc1.getItem();
Car c2=gc2.getItem();
```

```
class GenClass<T>{
    T item;

    GenClass(T o){
        item=o;
    }

    public T getItem() {
        return item;
    }
}
```

Reflection

- With reflection-technique you may study the type-information of class or object at runtime
- java.lang.Class is the starting point
  - Rest of the classes are defined in java.lang.reflect-package
- With class-object you may
  - Query the name of the class - String getName()
  - Instantiate class - Object newInstance()
  - Query available constructors - Constructor[] getConstructors()
  - Query the fields - Field[] getFields()
  - Query the methods - Method[] getMethods()
  - Query annotations attached to the class
    - Annotation[] getAnnotations()
    - Annotation getAnnotation(Class annotationType)

```
Class c1=Car.class;
Car volvo=new Car("V70",5,80);
Class c2=volvo.getClass();
```

C1 and c2 refer to the same object, the type information is in memory only once and it is created when the class-loader first loads the class

Annotations

- Annotations add metadata to an element
  - Class
  - Field
  - Method
  - And even parameters
- Annotations can be accessed
  - Through reflection at runtime
  - At compile time
- Annotations are basically just @interfaces
  - Only methods returning a value, taking no parameters
  - Method name is considered to be the metadata-item, return value type specifies the type for the item

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE,ElementType.METHOD})
public @interface Coder {
    String email();
}

@Coder(email="tom@tutorit.net")
class Car {

    @Coder(email="tim@tutorit.net")
    public void accelerate(int ds) {
        . . .
    }
}

void Test(Object o) {
    Class c=o.getClass();
    Coder ca=(Coder)c.getAnnotation(Coder.class);
    if (ca==null)
        System.out.println(o+" is not coded");
    sendEmailTo(ca.email());
}
```

Demo

- Declaring a generic method
- Declaring a generic class
- Declaring an annotation
- Using annotations and reflection to create a report of books