

**Интернализация:
классы Locale,
NumberFormat,
DateFormat,
Currency, Date,
Calendar**

Интернализация

- ❑ Класс **java.util.Locale** позволяет создать объект, описывающий географический или культурный регион, обеспечивая возможность создания многонациональных программ с учётом региональных настроек дат, времён, чисел, валюты и т.д. *Например,*

```
Locale locale = new Locale("fr", "FR");
```

- ❑ Класс **Locale** предназначен только для идентификации локали, никаких данных для локализации он не содержит.
- ❑ Виртуальная машина использует текущие региональные установки операционной системы:

```
Locale locale = Locale.getDefault();
```

- ❑ Метод *setDefault()* устанавливает используемые по умолчанию региональные данные.

<https://docs.oracle.com/javase/7/docs/api/java/util/Locale.html>

Интернализация

- ❑ Создание экземпляра **java.util.Locale** может быть сделано четырьмя различными способами:
 - используя константы **Locale**;
 - используя конструкторы **Locale**;
 - используя класс **Locale.Builder** (с Java 7);
 - используя метод **Locale.forLanguageTag()** (с Java 7).
- ❑ Для популярных стран - готовые константы:
 - **Locale.CANADA**
 - ...
 - **Locale.FRANCE**
 - **Locale.GERMAN**
 - **Locale.ITALY**
 - **Locale.JAPAN**

<https://docs.oracle.com/javase/8/docs/api/java/util/Locale.Builder.html>

Интернализация

- ❑ Конструкторы *Класс `java.util.Locale`* :
 - `Locale(String language);`
 - `Locale(String language, String country);`
 - `Locale(String language, String country, String variant);`
- ❑ Язык описывается двухбуквенным кодом ISO 639, код записывается в нижнем регистре;
- ❑ Страна обозначается двухбуквенным кодом ISO-3166, код записывается в верхнем регистре;
- ❑ Получить список языков и стран можно с помощью методов `Locale.getISOLanguages()` и `Locale.getISOCountries()` соответственно.

Интернализация

Например,

1. **long** number = 25_000_000L;
2. NumberFormat def = NumberFormat.getInstance();
3. System.out.println("Default: " + def.format(number));
4. NumberFormat cur =
NumberFormat.getInstance(Locale.GERMAN);
1. System.out.println("German: " + cur.format(number));
2. Locale loc = new Locale("ja", "JP");
3. NumberFormat curJp = NumberFormat.getInstance(loc);
4. System.out.println("Japan: " + curJp.format(number));

Получить правило форматирования для текущей локали

Получить *number* в виде строки с учетом локали

Получить правило форматирования для указанной локали

Вывод в консоли:

Default: 25 000 000
German: 25.000.000
JAPAN: 25,000,000

Интернализация

- ❑ Стандарты представления чисел и дат в различных странах могут существенно различаться;
- ❑ Для конвертации данных в различные региональные стандарты применяются возможности классов чувствительных к локали:
 - ✓ `java.text.NumberFormat`
 - ✓ `java.text.DateFormat`
 - ✓ `java.util.Calendar`
- ❑ Для преобразования числа в строку и обратно используются методы:
 - `String format(double number);`
 - `Number parse(String source);`

<https://docs.oracle.com/javase/7/docs/api/java/text/NumberFormat.html>

Интернализация

- ❑ Класс **java.util.Currency** представляет собой валюту.
- ❑ Валюта идентифицируются по их коду ISO 4217.
- ❑ Приложение может содержать только один экземпляр **Currency** для любой валюты, и причина - отсутствие открытого конструктора.
- ❑ Для получения экземпляра **Currency** используется перегруженный метод `getCurrencyInstance()`. Например,
 1. `BigDecimal currencyAmount = new BigDecimal("10.55");`
 2. `Currency cur = Currency.getInstance(Locale.US);`
 3. `NumberFormat curFmt =`
`NumberFormat.getCurrencyInstance(Locale.US);`
 4. `System.out.println(cur.getDisplayName() + ": " +`
`curFmt.format(currencyAmount));`

Вывод в консоли:
US Dollar: \$10.55

Интернализация

Пример,

1. **long** number = 25_000_000L;
2. NumberFormat curDef =
 NumberFormat.getCurrencyInstance();
3. System.out.println("Default: "+curDef.format(number));
4. NumberFormat curIt =
 NumberFormat.getCurrencyInstance(Locale.ITALY);
5. System.out.println("ITALY: "+curIt.format(number));
6. NumberFormat curCh =
 NumberFormat.getCurrencyInstance(Locale.CHINA);
6. System.out.println("CHINA: "+curCh.format(number));

Получить правило
форматирования
валюты для текущей
локали

Вывод в консоли:

Default: 25 000 000 грн.
ITALY: € 25.000.000,00
CHINA: ¥ 25,000,000.00

Интернализация

- ❑ Для получения текущей даты, не вдаваясь в подробности календаря, используйте класс **java.util.Date** для создания объекта **Date**, который будет содержать текущую системную дату и время.
- ❑ Время - это число в миллисекундах, прошедших с 1 января 1970 года, 00:00:00 GMT.

Например,

```
Date date = new Date();  
System.out.println(date);  
System.out.println(date.getTime());
```

Вывод в консоли:

```
Sun Sep 27 09:45:35 EEST 2015  
1443336335584
```

Интернализация

- ❑ Для отображения даты и времени в различных региональных стандартах можно использовать класс **java.text.DateFormat**;
- ❑ Получение объекта, отвечающего за обработку регионального стандарта даты и времени, похож на создание объекта, отвечающего за представление чисел:

`DateFormat.getDateInstance(<вид>, <локаль>);`

- ❑ Первый параметр это константы класса **DateFormat**:
 - **SHORT** – сокращенная дата, где месяц указывается в виде числа;
 - **MEDIUM** – месяц указывается сокращенно словесно, часовой пояс не указывается;
 - **LONG** – месяц указывается словесно полностью;
 - **FULL** – с указанием дня недели.

<https://docs.oracle.com/javase/7/docs/api/java/text/DateFormat.html>

Интернализация

Например,

```
Date date = new Date();  
System.out.println("Current date: " + date);  
DateFormat df =  
    DateFormat.getDateInstance(DateFormat.SHORT, Locale.UK);  
System.out.println("SHORT: " + df.format(date));  
  
df = DateFormat.getDateInstance(DateFormat.MEDIUM,  
                                Locale.UK);  
System.out.println("MEDIUM: " + df.format(date));  
  
df = DateFormat.getDateInstance(DateFormat.LONG, Locale.UK);  
System.out.println("LONG: " + df.format(date));  
  
df = DateFormat.getDateInstance(DateFormat.FULL, Locale.UK);  
System.out.println("FULL: " + df.format(date));
```

Интернализация

Вывод в консоли:

Current date: Wed Aug 16 16:50:59 EEST 2017

SHORT: 16/08/17

MEDIUM: 16-Aug-2017

LONG: 16 August 2017

FULL: Wednesday, 16 August 2017

Интернализация

- ❑ Если нужно быть точным относительно календаря, используйте класс **java.util.Calendar**.
- ❑ Описание полей:
 - DATE – число, обозначающее день месяца;
 - DAY_OF_MONTH – число, обозначающее день месяца;
 - DAY_OF_WEEK_IN_MONTH – число, указывающее день недели в текущем месяце;
 - DAY_OF_YEAR – число, обозначающее день года;
 - WEEK_OF_YEAR – число, указывающее неделю года;
 - HOUR – число, указывающее час;
 - HOUR_OF_DAY – число, указывающее час дня;
 - MILLISECOND – число, указывающее миллисекунды;
 - MINUTE – число, указывающее минуты;
 - MONTH – число, указывающее месяц года;
 - SECOND – число, указывающее секунды;
 - YEAR – число, указывающее год.

<https://docs.oracle.com/javase/7/docs/api/java/util/Calendar.html>

Интернализация

Например,

1. `Calendar mcl = Calendar.getInstance();`
2. `int day = mcl.get(Calendar.DATE);`
3. `int month = mcl.get(Calendar.MONTH) + 1;`
4. `int yr = mcl.get(Calendar.YEAR);`
5. `String dateStr = day + "." + month + "." + yr;`
6. `System.out.println(dateStr);`
7. `int hour = mcl.get(Calendar.HOUR);`
8. `int min = mcl.get(Calendar.MINUTE);`
9. `int sec = mcl.get(Calendar.SECOND);`
10. `System.out.println(hour + ":" + min + ":" + sec);`

Нумерация месяцев начинается с 0, для правильного отображения нужно добавить 1

Вывод в консоли:

16.8.2017
2:17:25

Локализация приложений

- ❑ Для создания приложений поддерживающих несколько языков рекомендуется собрать все локализуемые данные в отдельном месте - в *пакетах ресурсов* (resource bundle):
 - в этом случае достаточно отредактировать ресурс, не затрагивая исходный код программы.
- ❑ Каждый пакет представляет собой файл свойств или класс, который описывает элементы, специфические для конкретного регионального стандарта (например, сообщения, надписи и т.д.).
- ❑ В каждый пакет помещаются ресурсы для всех региональных стандартов, поддержка которых предполагается в программе.

Интернализация

- ❑ *Файлы свойств* используются для определения строковых ресурсов, а для ресурсов других типов создаются классы ресурсов.
- ❑ Для именования пакетов ресурсов используются специальные соглашения:
 - ресурсы для конкретной страны именуются так:
имяПакета_язык_СТРАНА
 - ресурс для конкретного языка так:
имяПакета_язык
 - ресурс, применяемый по умолчанию, так:
имяПакета
т.е. имя пакета не содержит суффикса.

Интернализация

Например, для поддержки английского, русского и украинских языков имена файлов ресурсов (свойств) будут выглядеть так:

- ✓ text_en_GB.properties
- ✓ text_ru_RU.properties
- ✓ text_uk_UA.properties
- ✓ text.properties

Обязательно наличие файла ресурсов с базовым именем - *пакет ресурсов по умолчанию*

- ❑ Файл свойств - это обычный текстовый файл, где каждая строка содержит ключ и значение. *Например*,

```
# This is the text_en_GB.properties file  
label1 = login  
label2 = password  
message1 = Hello
```

Значение – строка символов

Ключ – строка символов

Комментарий начинается с символа #

Интернализация

- ❑ Класс **java.util.PropertyResourceBundle** позволяет загружать ресурсы из файла свойств и является подклассом абстрактного класса **java.util.ResourceBundle**.
- ❑ Для загрузки ресурсов используется перегруженный статический метод:

```
getBundle(String baseName, Locale loc);  
getBundle(String baseName);
```

- Загружает класс пакета ресурсов с заданным именем, а также его родительские классы для указанного регионального стандарта;
- Если классы пакетов расположены в Java-пакете, то должно быть указано полное имя, (*например, intl.ProgramResources*).
- Классы пакетов ресурсов должны быть объявлены открытыми (*public*), чтобы метод мог обращаться к ним.

<https://docs.oracle.com/javase/8/docs/api/java/util/PropertyResourceBundle.html>

Интернализация

❑ Порядок поиска ресурсов:

А. Сначала формируется список кандидатов в имена пакетов:

имяПакета_трс-язык_трс-СТРАНА_трс-вариант

имяПакета_трс-язык_трс-СТРАНА

имяПакета_трс-язык

имяПакета_рсу-язык_рсу-СТРАНА_рсу-вариант

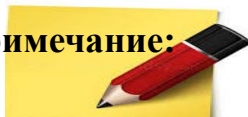
имяПакета_рсу-язык_рсу-СТРАНА

имяПакета_рсу-язык

имяПакета

трс - текущий региональный стандарт
рсу - региональный стандарт по умолчанию

Примечание:



Кандидаты формируются так: сначала тот, который соответствует языку, стране и варианту **трс**. Затем последовательно отбрасываются вариант, страна и язык. Затем ресурс, соответствующий **рсу** с последовательным отбрасыванием для него варианта, страны и языка. В конце пакет ресурсов по умолчанию.

Интернализация

В. Для каждого кандидата:

- ✓ попытка загрузить класс с таким именем (если такой класс может быть загружен, то *getBundle()* создает его и возвращает как результирующий пакет ресурсов);
- ✓ если загрузка не удалась, то *getBundle()* начинает поиск файла свойств, попытка его загрузить и создать на его основе экземпляр класса **PropertyResourceBundle** (из имени кандидата он генерирует путь, заменяя все "." на "/", добавляя "/" впереди и .properties в конце);
- ✓ если попытка успешна, то полученный объект возвращается как результирующий пакет ресурсов;
- ✓ если не удалось загрузить файл ресурсов – берется следующий кандидат в имя пакета;
- ✓ если кандидатов больше нет – выбрасывается исключение **java.util.MissingResourceException**.

Интернализация

С. Для загруженного пакета ресурсов создается родительская цепочка:

- ✓ родительские имена определяются путем последовательного исключения из имени варианта, страны и языка;
- ✓ родительские элементы нужны в тех случаях, когда необходимый ресурс не найден в загруженном пакете (т.е. поиск ресурса проверяет последовательно все пакеты до первого вхождения).

Например, если необходимый ресурс не найден в пакете *имяПакета_de_DE*, то выполняется поиск ресурса в пакетах *имяПакета_de*, *имяПакета*.

- ✓ если ресурс найден в пакете – он возвращается;
- ✓ если нет, то вызов делегируется родителю; если вызов дошел до последнего родителя, а ресурс не найден – выбрасывается исключение **java.util.MissingResourceException**.

Интернализация

- ❑ Для получения ресурса используются методы:
 - `String getString (String name)`
Извлекает значение по ключу из пакета ресурсов или его родительских пакетов.
 - `Object getObject (String name)`
Извлекает объект по ключу из пакета ресурсов или его родительских пакетов.
- ❑ Для выборки всех значений ключей в **ResourceBundle** используется метод:
 - `Enumeration getKeys ()`
Возвращает объект *Enumeration*, содержащий ключи текущего пакета ресурсов (при этом в объект *Enumeration* также помещаются ключи из родительских пакетов ресурсов).

Интернализация

Рекомендации по практическому применению

- ❑ Все файлы ресурсов следует разместить в директории **property** к корню проекта (на уровне с пакетами приложения);
- ❑ Для взаимодействия с файлами свойств лучше создать отдельный специальный класс, экземпляр которого будет только один и позволит:
 - извлекать информацию по ключу;
 - изменять значение локали.

Примечание:



Загрузка ресурсов из файлов свойств производится с помощью класса `java.util.Properties`, поэтому предполагается, что файл записан в кодировке *iso-8859-1*. Для всех символов, не входящих в эту кодировку, необходимо использовать escape-последовательность вида `\uXXXX`, где XXXX – код символа в *UTF-16*.

Интернализация

Например, создадим три следующих файла:

✓ Файл *text_en_GB.properties*

```
# This is the text_en_GB.properties file  
label1 = login  
label2 = password  
message = Hello
```

Английский

✓ Файл *text_uk_UA.properties*

```
# This is the text_uk_UA.properties file  
label1 = логін  
label2 = пароль  
message = Вітаю
```

Украинский

✓ Файл *text.properties*

```
# This is the text.properties file  
label1 = логин  
label2 = пароль  
message = Приветствую
```

Русский
(по умолчанию)

Интернализация

Пример, описание класса для работы с файлами ресурсов:

```
public enum ResourceManager {  
    INSTANCE;  
    private ResourceBundle resourceBundle;  
    private final String resourceName = "property.text";  
    private ResourceManager() {  
        resourceBundle = ResourceBundle.getBundle(  
            resourceName, Locale.getDefault());  
    }  
    public void changeResource(Locale locale) {  
        resourceBundle = ResourceBundle.getBundle(  
            resourceName, locale);  
    }  
    public String getString(String key) {  
        return resourceBundle.getString(key);  
    }  
    public Enumeration getSetKey() {  
        return resourceBundle.getKeys();  
    }  
}
```

Экземпляр ресурса

Полное имя файла

При создании получает локаль по умолчанию

Метод изменения локали для ресурса

Метод получения значения по ключу

Метод получения всех ключей

Интернализация

Пример, использования класса ResourceManager:

```
public class ResourceRunner {  
    public static void main(String[] args) {  
        ResourceManager manager =  
            ResourceManager.INSTANCE;  
        System.out.println("Default welcome -> " +  
            manager.getString("message") + "\n");  
        for (int i=0; i<3; i++) {  
            printMenu();  
            char command = inputCommand();  
            Locale locale = getLocale(command);  
            manager.changeResource(locale);  
            showResource(manager);  
            System.out.println();  
        }  
    }  
}
```

The diagram consists of five rectangular boxes with Russian text, connected to the code by blue arrows. The boxes are: 'Печать меню' (Print menu) pointing to 'printMenu()'; 'Получение пункта меню' (Get menu item) pointing to 'inputCommand()'; 'Создание локали' (Create locale) pointing to 'getLocale(command)'; 'Установка нового пакета ресурсов с учетом новой локали' (Install new resource package taking into account the new locale) pointing to 'changeResource(locale)'; and 'Вывод всех ресурсов' (Output all resources) pointing to 'showResource(manager)'.

// ...

Интернализация

Продолжение примера,

```
public static void printMenu() {
    System.out.println("1 – english");
    System.out.println("2 – русский");
    System.out.println("3 – український");
    System.out.print("> ");
}

public static char inputCommand() {
    Scanner sc = new Scanner(System.in);
    while (true) {
        try {
            int com = sc.nextInt();
            if (com < 1 || com > 3)
                throw new IOException();
            return Character.forDigit(com, 10);
        } catch (IOException exp) {
            System.out.println("Error enter command!!! Repeat\n> ");
        }
    }
}
```

Интернализация

Продолжение примера,

```
public static Locale getLocale(char command) {  
    switch (command) {  
        case '1':  
            return new Locale("en", "GB");  
        case '3':  
            return new Locale("uk", "UA");  
        default:  
            return Locale.getDefault();  
    }  
}
```

```
public static void showResource(ResourceManager manager) {  
    Enumeration<String> setKey = manager.getSetKey();  
    while (setKey.hasMoreElements()) {  
        String key = setKey.nextElement();  
        System.out.println(key + " = " + manager.getString(key));  
    }  
}
```

Интернализация

Вывод в консоли:

Default welcome -> Приветствую

1 – english

2 – русский

3 – український

> 1

message = Hello

label1 = login

label2 = password

1 – english

2 – русский

3 – український

> 3

message = Â³òàŗ

label1 = ëîã³í

label2 = ïàďîëü

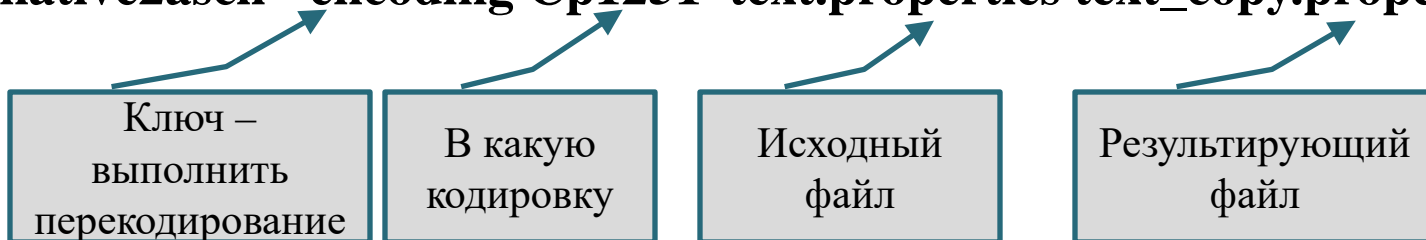
Не соответствие
кодировки

Интернализация

- ❑ Для корректной работы требуется воспользоваться утилитой `java native2ascii`, чтобы изменить кодировку файлов с русскими и украинскими словами.

Например,

`native2ascii -encoding Cp1251 text.properties text_copy.properties`



В результате получим файл:

This is the text.properties file

`label1 = \u043b\u043e\u0433\u0438\u043d`

`label2 = \u043f\u0430\u0440\u043e\u043b\u043e\u0432\u044c`

`message = \u041f\u0440\u0438\u0432\u0435\u0442\u0441\u044f \u0441\u0432\u0435\u0442\u0430 \u0441\u0432\u0435\u0442\u0430`

Интернализация

- ❑ После замены кодировки файлов ресурсов:

Вывод в консоли:

Default welcome -> Приветствую

1 – english

2 – русский

3 – український

> 3

message = Вітаю

label1 = логін

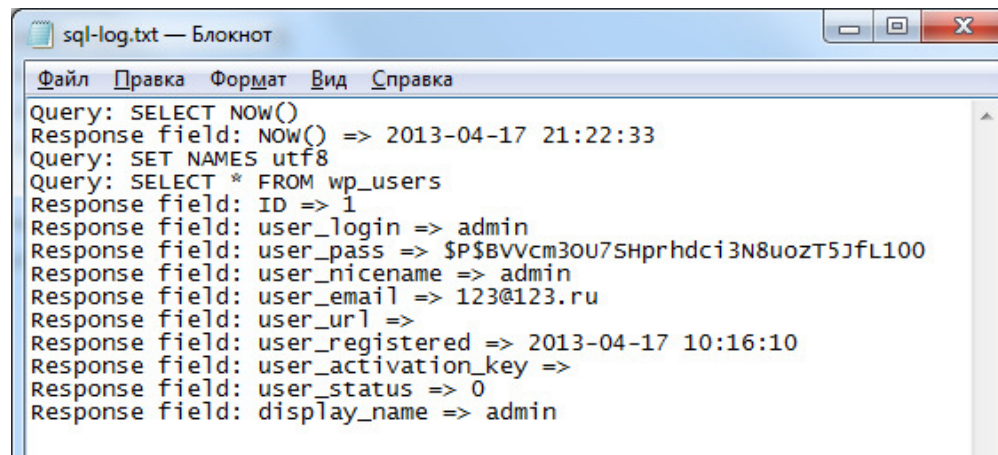
label2 = пароль



Логирование

Логирование

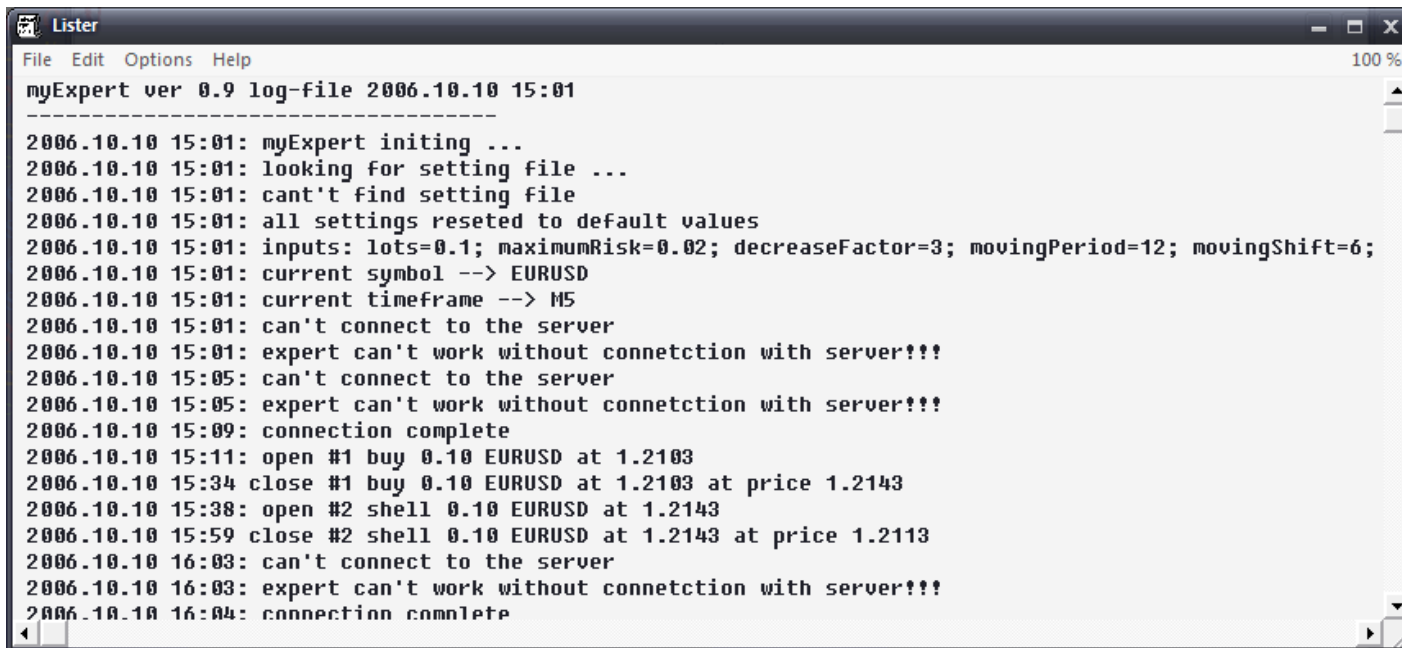
- ❑ **Логирование** – это возможность вести протокол (журнал, лог) работы самого приложения и внешних событий в хронологическом порядке.
- ❑ Обычно протоколирование ведется в файл.
 - *Лог-файлы сервера* — протоколируются определённые действия пользователя или программы на сервере. Например, информация, откуда пришёл пользователь, когда и сколько времени он провёл на сайте, что смотрел и скачивал, какой у него браузер и какой IP-адрес его компьютера;



```
sql-log.txt — Блокнот
Файл  Правка  Формат  Вид  Справка
Query: SELECT NOW()
Response field: NOW() => 2013-04-17 21:22:33
Query: SET NAMES utf8
Query: SELECT * FROM wp_users
Response field: ID => 1
Response field: user_login => admin
Response field: user_pass => $P$BVVcm3OU7SHprhdcI3N8uozT5JfL100
Response field: user_nicename => admin
Response field: user_email => 123@123.ru
Response field: user_url =>
Response field: user_registered => 2013-04-17 10:16:10
Response field: user_activation_key =>
Response field: user_status => 0
Response field: display_name => admin
```

Логирование

- *Файл-протокол* (обычно текстовый файл) — запись с различной (настраиваемой) степенью детализации сведений о происходящих в системе событиях (ошибки, предупреждения, сообщения):
 - ✓ в большинстве современных приложений одно событие — одна строка: так их легко генерировать программно и анализировать человеком.



```
Lister
File Edit Options Help
100 %
myExpert ver 0.9 log-file 2006.10.10 15:01
-----
2006.10.10 15:01: myExpert initing ...
2006.10.10 15:01: looking for setting file ...
2006.10.10 15:01: cant't find setting file
2006.10.10 15:01: all settings reseted to default values
2006.10.10 15:01: inputs: lots=0.1; maximumRisk=0.02; decreaseFactor=3; movingPeriod=12; movingShift=6;
2006.10.10 15:01: current symbol --> EURUSD
2006.10.10 15:01: current timeframe --> M5
2006.10.10 15:01: can't connect to the server
2006.10.10 15:01: expert can't work without connetction with server!!!
2006.10.10 15:05: can't connect to the server
2006.10.10 15:05: expert can't work without connetction with server!!!
2006.10.10 15:09: connection complete
2006.10.10 15:11: open #1 buy 0.10 EURUSD at 1.2103
2006.10.10 15:34 close #1 buy 0.10 EURUSD at 1.2103 at price 1.2143
2006.10.10 15:38: open #2 shell 0.10 EURUSD at 1.2143
2006.10.10 15:59 close #2 shell 0.10 EURUSD at 1.2143 at price 1.2113
2006.10.10 16:03: can't connect to the server
2006.10.10 16:03: expert can't work without connetction with server!!!
2006.10.10 16:04: connection complete
```

Логирование

- ❑ Существуют различные API регистрации сообщений и ошибок:
 - на практике применяется **Log4j**, обладающий качественной архитектурой, в следствие чего доминирует над другими.
- ❑ **Log4j** – это инструмент для формирования журнала сообщений (отладочных, информационных, системных, безопасности, сообщений об ошибках и т.д.).
- ❑ API Log4j2 можно загрузить по адресу:
<https://logging.apache.org/log4j/2.x/download.html>
- ❑ Перед использованием необходимо зарегистрировать загруженную библиотеку в приложении, *например*: файл **log4j-1.2.13.jar** (для первой версии). Для версии 2 требуется два файла: **log4j-api-2.11.0.jar** и **log4j-core-2.11.0.jar**



Логирование

- Для регистрации библиотеки версии 1 (например, файл **log4j-1.2.17.jar**) в проекте Maven достаточно в *pom.xml* добавить зависимость:

```
<dependency>  
    <groupId>log4j</groupId>  
    <artifactId>log4j</artifactId>  
    <version>1.2.17</version>  
</dependency>
```

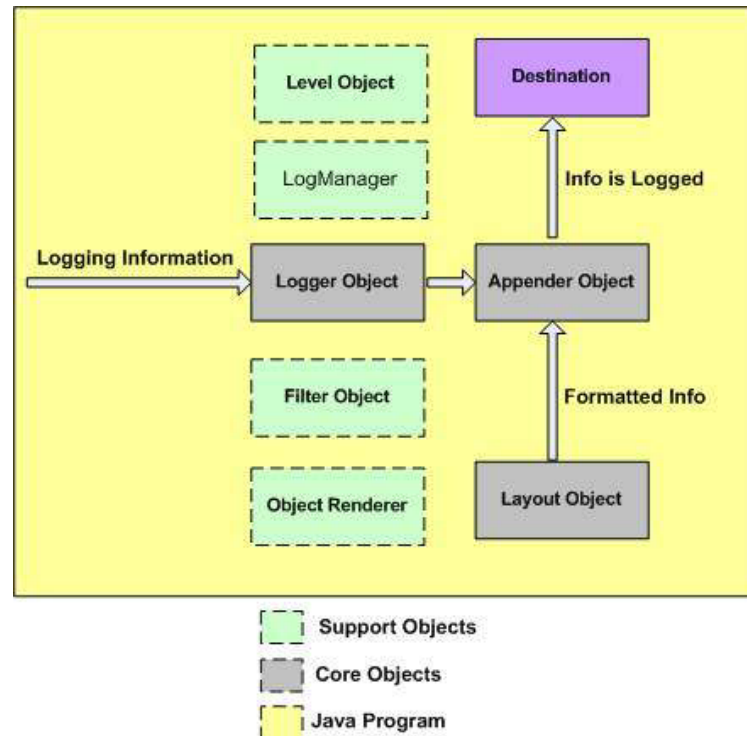
Логирование

- Для регистрации библиотеки версии 2 (например, файлы **log4j-api-2.11.0.jar** и **log4j-core-2.11.0.jar**) в проекте Maven достаточно в *pom.xml* добавить зависимости:

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.11.0</version>
  </dependency>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.11.0</version>
  </dependency>
</dependencies>
```

Логирование

- ❑ Log4j состоит из трех элементов:
 - регистрирующего (logger);
 - направляющего вывод (appender);
 - форматирующего (layout).



Таким образом **logger** регистрирует и направляет вывод события в пункт назначения, определяемый элементом **appender**, в формате, заданном элементом **layout**.

Регистраторы (Loggers)

- ❑ Основным элементом регистрации событий и ошибок является **org.apache.log4j.Logger**:
 - вывод регистратора может быть направлен на консоль, в файл, базу данных, GUI-компонент или сокет;
 - это компонент приложения, принимающий и выполняющий запросы на запись в регистрационный журнал.
- ❑ Каждый класс приложения может иметь свой собственный *logger* или быть прикреплен к общему для всего приложения:
 - регистраторы образуют иерархию, как и пакеты Java;
 - каждый регистратор имеет имя, описывающее иерархию, к которой он принадлежит;
 - разделителем в описании иерархия является точка.

Логирование

- ❑ Регистратор может быть создан/получен с помощью статического метода *getLogger(String name)/getLogger(Class name)*:
 - параметр *name* – уникальный;
 - вызов *getLogger()* с одним и тем же именем вернет один и тот же регистратор;
- ❑ Регистратор синхронизирован и его можно свободно использовать в многопоточной среде;
- ❑ На вершине иерархии находится корневой регистратор;
- ❑ Корневой регистратор всегда существует и у него нет имени;
- ❑ Корневой регистратор может быть получен статическим методом *getRootLogger()*.

Логирование

- ❑ Регистраторы могут конфигурироваться с использованием уровней, т.е. класса **org.apache.log4j.Level**, что означает сообщения какого уровня надо писать в лог;
- ❑ Уровни упорядочены по старшинству (в порядке возрастания):
 - ALL - все уровни включая пользовательские уровни;
 - TRACE - детальная отладочная информация;
 - DEBUG - отладочные сообщения;
 - INFO - информационные сообщения;
 - WARN - сообщения об потенциально опасной ситуации;
 - ERROR - сообщения об ошибках, которые позволят приложению работать дальше;
 - FATAL - сообщения об серьезных ошибках, которые скорее всего приведут к завершению работы приложения;
 - OFF - отключение записи всех сообщений.

Логирование

- ❑ Уровень регистратора можно указать с помощью метода *setLevel(Level level)*;
- ❑ Если уровень регистратора не указывается, то наследуется уровень родителя;
- ❑ Уровень корневого регистратора **DEBUG**.
- ❑ Для вывода сообщений конкретного уровня используются методы *debug()*, *info()*, *warn()*, *error()*, *fatal()*;
- ❑ Чтобы вывести информацию о возникшем исключении, в качестве второго параметра в вышеперечисленные методы нужно передать объект типа **Throwable**;
- ❑ Для вывода сообщения необходимо, чтобы уровень выводимого сообщения был не ниже, чем уровень регистратора:
 - Если уровень регистратора выше уровня сообщения, то оно отбрасывается и его запись не происходит. Например, уровень регистратора **INFO**, то вызов *logger.debug("message")* не даст никакого эффекта, т. к. **DEBUG < INFO**.

Логирование

❑ Общие методы регистрации сообщений:

- *log(Priority priority, Object message, Throwable t)* –выводит сообщения указанного уровня с информацией об исключительной ситуации **t**;
- *log(Priority priority, Object message)* – выводит сообщения указанного уровня.



- ✓ Класс **org.apache.log4j.Priority** является суперклассом для класса **org.apache.log4j.Level**;
- ✓ Если указанный уровень сообщения ниже уровня регистратора, то сообщения будет отброшено.

Направители вывода (Appenders)

- ❑ Объект типа **Appender** (реализует интерфейс **org.apache.log4j.Appender**) несет ответственность за вывод протоколируемой информации в различные пункты назначения, такие как:
 - *консоль* – класс **ConsoleAppender**;
 - *файл* – классы **FileAppender**, **RollingFileAppender**, **DailyRollingFileAppender**;
 - *база данных* – класс **JDBCAppender**;
 - *журнал событий ОС* – **NTEventLogAppender**;
 - *SMTP сервер* – **SMTPAppender**;
 - *удаленный сервер (сокеты)* – **SocketAppender** ;
 - *т.д.*

Особенности (1/2)

- ❑ Регистраторы (логгеры) связываются с направителями вывода (аппендерами) отношением «многие-ко-многим»:
 - у одного логгера может быть несколько аппендеров;
 - к одному аппендеру может быть привязано несколько логгеров.
- ❑ Аппендеры наследуют от логгеров (*уровень логирования*);
- ❑ Любой вывод, сделанный в логгере, будет направлен всем его предкам;
 - Например, корневой логгер сконфигурирован для вывода в консоль с уровнем ERROR, а дочерний – вывод в файл с уровнем INFO. Тогда запись через дочерний логгер произведется и в файл, и на консоль.
- ❑ Отказаться от наследования можно с помощью метода *setAdditivity(boolean additive)* класса **Logger**, передав ему значение *false*.

Особенности (2/2)

- ❑ Добавить аппендер к регистратору можно с помощью метода *addAppender(Appender newAppender)* класса **Logger**;
- ❑ Наиболее широко используются файловые аппендеры;
- ❑ Для установки файла нужно передать его имя в конструктор файлового аппендера или в метод *setFile(String file)* типа **Appender**;
- ❑ По умолчанию любые сообщения, записанные в файл, будут добавляться к уже имеющимся;
- ❑ Изменить это можно сбросив флаг *append* с помощью конструктора или метода *setAppend(boolean append)* типа **Appender**.

Форматеры (Layout)

- ❑ Вывод регистратора может иметь различный формат;
- ❑ Каждый формат представлен подклассом **Layout**;
- ❑ Все методы класса **Layout** предназначены только для создания объектов подклассов.

- ❑ В библиотеке определены следующие подклассы:
 - **SimpleLayout** – вывод в простом текстовом формате;
 - **HTMLLayout** – вывод в HTML-формате;
 - **XMLLayout** – вывод в XML-формате;
 - **PatternLayout** – вывод согласно шаблону.

- ❑ Установить **Layout** для **FileAppender** или **ConsoleAppender** можно с помощью метода *setLayout(Layout layout)* или передать его в вышеперечисленные конструкторы этих классов.

Логирование

Пример 6:

```
public class DemoLog {  
    static Logger logger = Logger.getLogger(DemoLog.class);  
  
    public static void main(String[] args) {  
        try {  
            FileAppender appender =  
                new FileAppender(  
                    new SimpleLayout(), "log.txt");  
            logger.addAppender(appender);  
            logger.setLevel(Level.DEBUG);  
  
            factorial(9);  
            factorial(-3);  
        } catch (IllegalArgumentException e) {  
            logger.error("negative argument", e);  
        } catch (IOException e) {  
            logger.error("error i/o", e);  
        }  
    }  
    // .....  
}
```

Получение логгера для
класса DemoLog

Создания аппендера
для файла

Связать аппендер
и логгер

Установить уровень
логгера

Вывод сообщения
уровня ERROR

Логирование

Продолжение примера 6:

// ...

```
public static int factorial(int n) {  
    if (n < 0)  
        throw new IllegalArgumentException(  
            "argument " + n + " less than zero");  
  
    logger.debug("Argument n is " + n);  
    int result = 1;  
    for (int i = n; i >= 1; i--)  
        result *= i;  
  
    logger.info("Result is " + result);  
    return result;  
}
```

Вывод сообщения
уровня DEBUG

Вывод сообщения
уровня INFO

Логирование

- ✓ В результате: выполнение программы завершится нормально, а файл *log.txt* будет содержать следующие сообщения:

```
DEBUG - Argument n is 9
INFO - Result is 362880
ERROR - negative argument
java.lang.IllegalArgumentException: argument -3 less than zero
    at simlpe_1.DemoLog.factorial(DemoLog.java:30)
    at simlpe_1.DemoLog.main(DemoLog.java:21)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:497)
    at com.intellij.rt.execution.application.AppMain.main(AppMain.java:147)
```

- ✓ Можно добавить к регистратору еще один аппендер, *например*:
`logger.addAppender(new ConsoleAppender(new SimpleLayout()));`
- ✓ В результате выполнения программы получим в консоли:

```
DEBUG - Argument n is 9
INFO - Result is 362880
ERROR - negative argument
java.lang.IllegalArgumentException: argument -3 less than zero
    at simlpe_1.DemoLog.factorial(DemoLog.java:28)
    at simlpe_1.DemoLog.main(DemoLog.java:19) <5 internal calls>
```

Логирование

- ❑ Форматирование вывода сообщения с использованием **PatternLayout** требует указания шаблонной строки, состоящей из параметров, *например*:
 - %с – полное имя категории (возможно %с{} с числом уровней);
 - %C – полное имя класса (желательно избегать);
 - %F – имя файла (желательно избегать);
 - %M – имя метода (желательно избегать);
 - %L – номер строки в файле (желательно избегать);
 - %p – уровень логирования;
 - %d – дата и время (возможно %d{}: {hh:mm:ss}, или {HH:MM:SS}, или их комбинация);
 - %m – само сообщение;
 - %t – имя потока;
 - %r - количество миллисекунд с момента инициализации системы логирования;
 - %n – переход на следующую строку.

Конфигурирование

- ❑ Программная конфигурация регистраторов используется редко из-за своей громоздкости и отсутствия гибкости при изменении настроек.
- ❑ Конфигурирование осуществляется через:
 - файл свойств **log4j.properties**;
 - Файл конфигурации **log4j.xml**.
- ❑ При инициализации **Log4J** они ищутся в переменной CLASSPATH: сначала xml-файл, потом properties-файл;
- ❑ При наличии обоих предпочтение отдается xml-файлу.

Файл свойств

❑ Формат ключей:

- **log4j.rootLogger** – глобальный объект журнала, значением этого ключа, необходимо указать уровень логирования;
 - ✓ через запятую перечисляются имена аппендеров;
- **log4j.appender.имя_аппендера** – значение данного ключа это тип аппендера (от него будут зависеть остальные ключи и значения для данного типа). *Например:*
 - ✓ для *org.apache.log4j.ConsoleAppender* такие данные:
log4j.appender.имя_аппендера.Target – куда выводить сообщение (System.out или System.err);
- **log4j.appender. имя_аппендера.layout** – шаблон форматирования строки журнала.

Логирование

Пример 7, файл свойств для вывода сообщений в консоль:

```
log4j.rootLogger = INFO, console
log4j.appender.console = org.apache.log4j.ConsoleAppender
log4j.appender.console.encoding = Cp866
log4j.appender.console.layout = org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern = %d [%5p] [%t] [%c] -
%m%n
```

- ❑ Создадим в папке **resources** проекта примера 6 этот файл **log4j.properties**;
- ❑ Удалим из кода класса **DemoLog** программную конфигурацию системы логирования;
- ❑ Получим результат:

```
2017-08-28 21:36:27,032 [ INFO] [main] [DemoLog] - Result is 362880
2017-08-28 21:36:27,033 [ERROR] [main] [DemoLog] - negative argument
java.lang.IllegalArgumentException: argument -3 less then zero
    at simlpe_1.DemoLog.factorial(DemoLog.java:29)
    at simlpe_1.DemoLog.main(DemoLog.java:19) <5 internal calls>
```

Логирование

Пример 8, файл конфигурации, соответствующий вышеприведенному файлу свойств и расположенный в папке **resources** проекта примера 6:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration PUBLIC
    "-//APACHE//DTD LOG4J 1.2//EN"
    "http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/xml/doc-
files/log4j.dtd">

<log4j:configuration>
  <appender name="console" class="org.apache.log4j.ConsoleAppender">
    <param name="Encoding" value="Cp866"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d [%t] [%5p][%c{1}] - %m%n" />
    </layout>
  </appender>

  <root>
    <priority value="INFO"/>
    <appender-ref ref="console"/>
  </root>

</log4j:configuration>
```

Логирование

- ❑ Поскольку регистраторы с аппендерами связаны отношением "многие ко многим", то добавим в файл **log4j.xml** еще один аппендер для вывода в файл:

➤ добавим тег:

```
<appender name="file" class="org.apache.log4j.RollingFileAppender">  
  <param name="Encoding" value="UTF-8"/>  
  <param name="File" value="log.txt"/>  
  <layout class="org.apache.log4j.PatternLayout">  
    <param name="ConversionPattern" value="%d [%5p][%c{1}] - %m%n"/>  
  </layout>  
</appender>
```

➤ ИЗМЕНИМ ТЕГ:

```
<root>  
  <priority value="INFO"/>  
  <appender-ref ref="console"/>  
  <appender-ref ref="file" />  
</root>
```