

СЕМИНАР 2

QFile и QJSON

Конфигурационные файлы

Логирование

JSON

- **JSON** (JavaScript Object Notation) – текстовый формат обмена данными, основанный на подмножестве языка программирования JavaScript, определенного в стандарте ECMA-262 3rd Edition - December 1999.

Особенности:

- Удобен для чтения;
- Самостоятелен и структура совместима с любым языком программирования;
- Существует множество готовых библиотек для работы с JSON на различных языках программирования(<https://www.json.org/json-ru.html>).

JSON. Типы значений

Объект

- QJsonObject

Массив

- QJsonArray

Число

- Qvariant, double и т.п.

Литерал

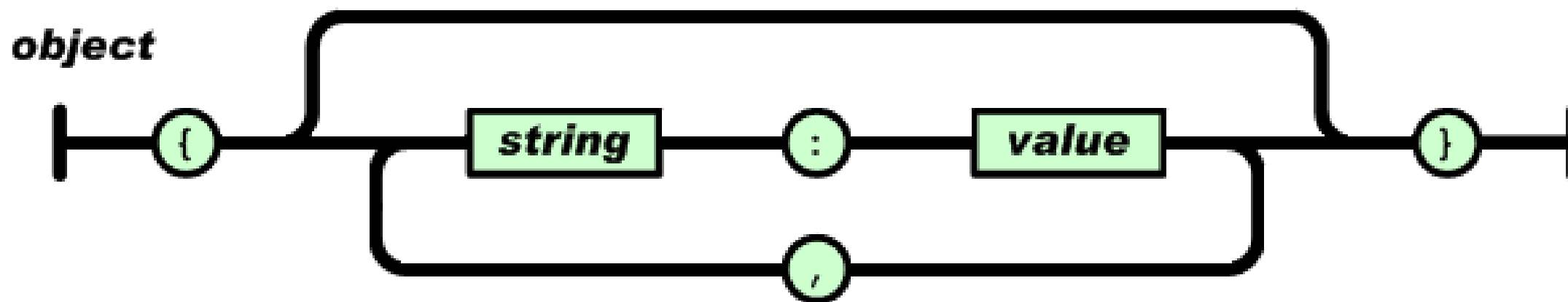
- True, false, null

Строка

- QString, stdString и т.п.

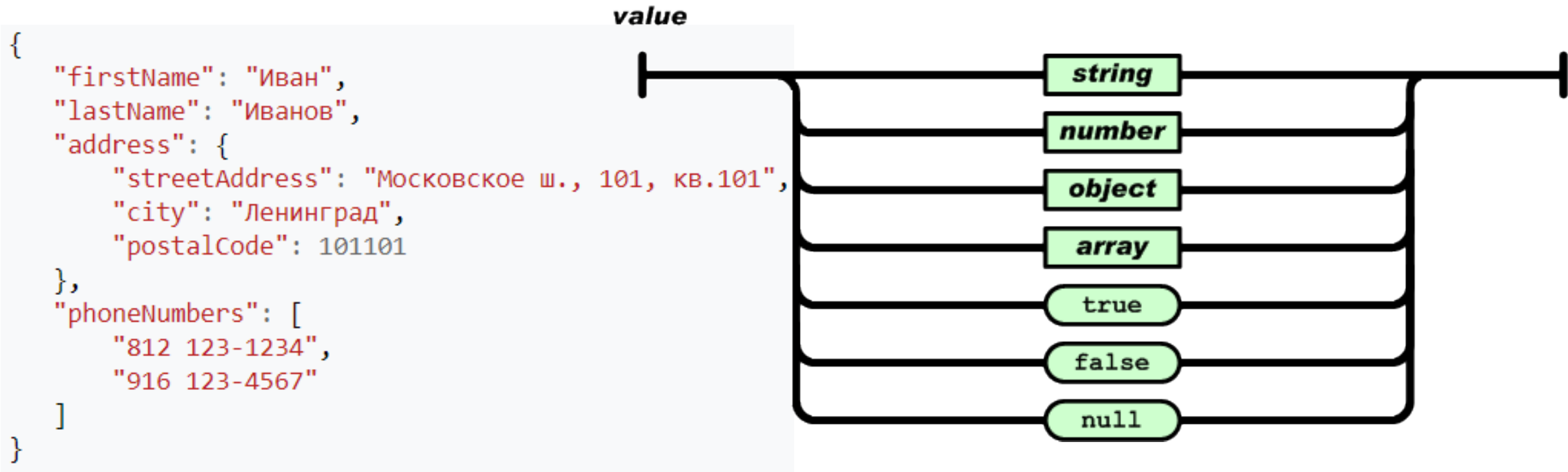
JSON – Объект. (QJsonObject)

- *Объект* - неупорядоченный набор пар ключ/значение. Объект начинается с { (открывающей фигурной скобки) и заканчивается } (закрывающей фигурной скобкой). Каждое имя сопровождается : (двоеточием), пары ключ/значение разделяются , (запятой).



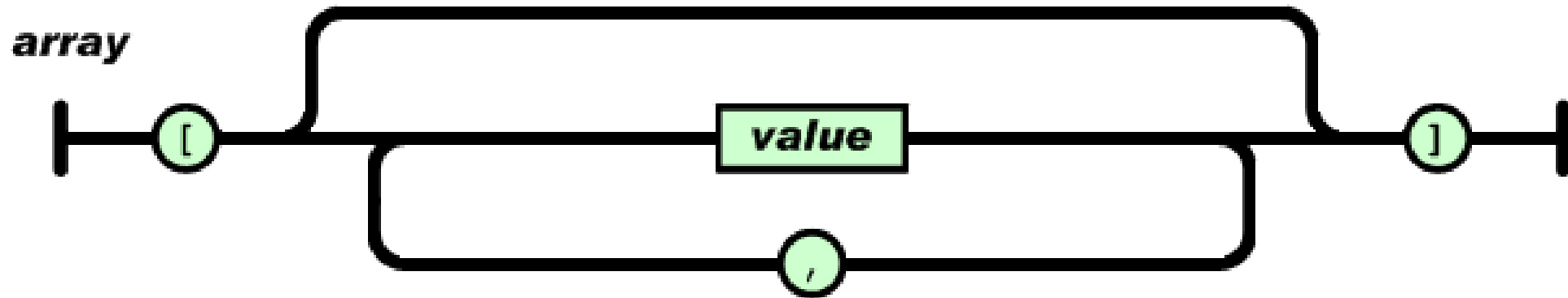
JSON – Объект. Значение (QJsonValue)

- *Ключ* – всегда строка. *Значение* может быть *строкой* в двойных кавычках, *числом*, *true*, *false*, *null*, *объектом* или *массивом*. Эти структуры могут быть вложенными



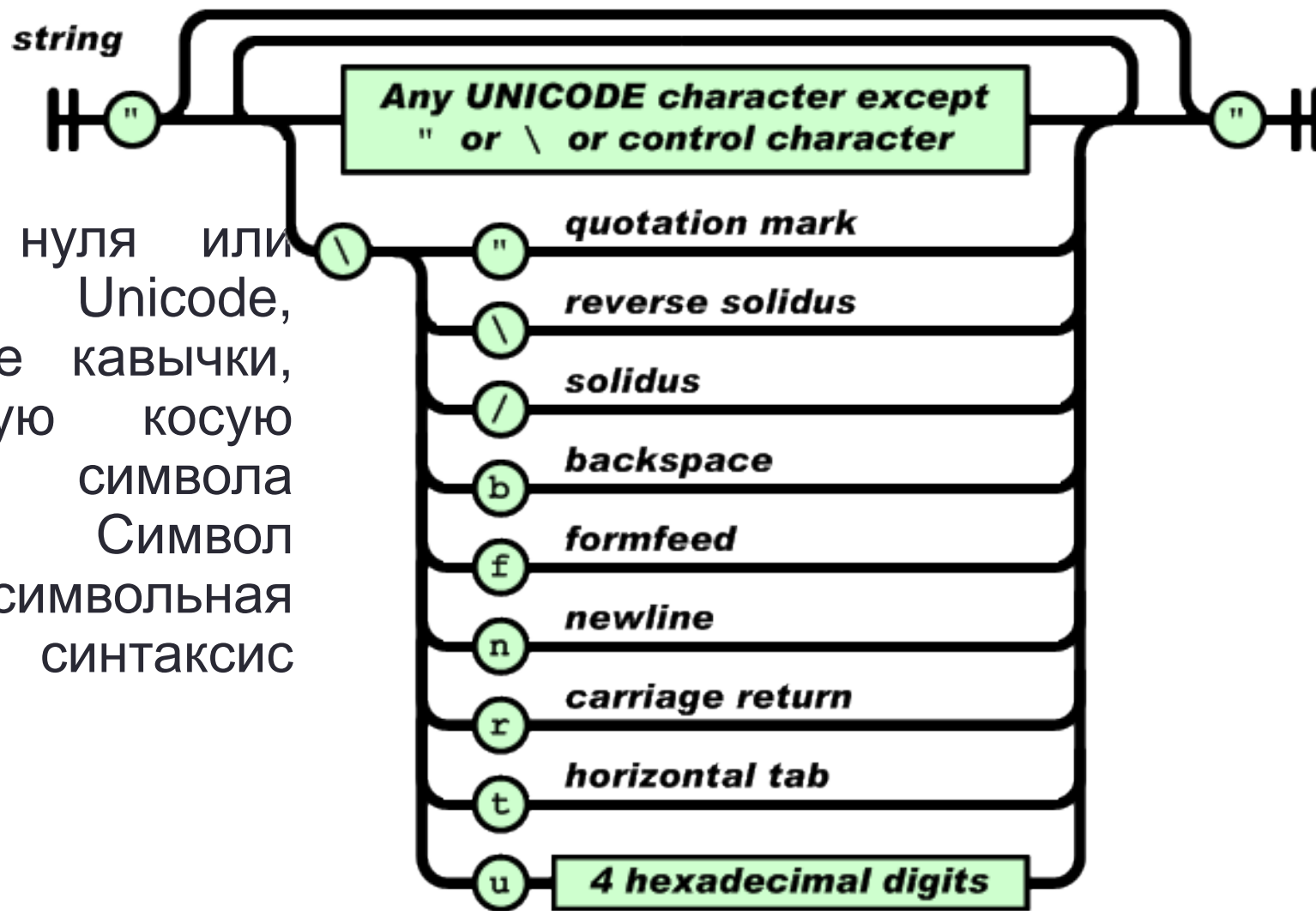
JSON – Массив. (QJsonArray)

- *Массив* - упорядоченная коллекция значений. Массив начинается с [(открывающей квадратной скобки) и заканчивается] (закрывающей квадратной скобкой). Значения разделены , (запятой).

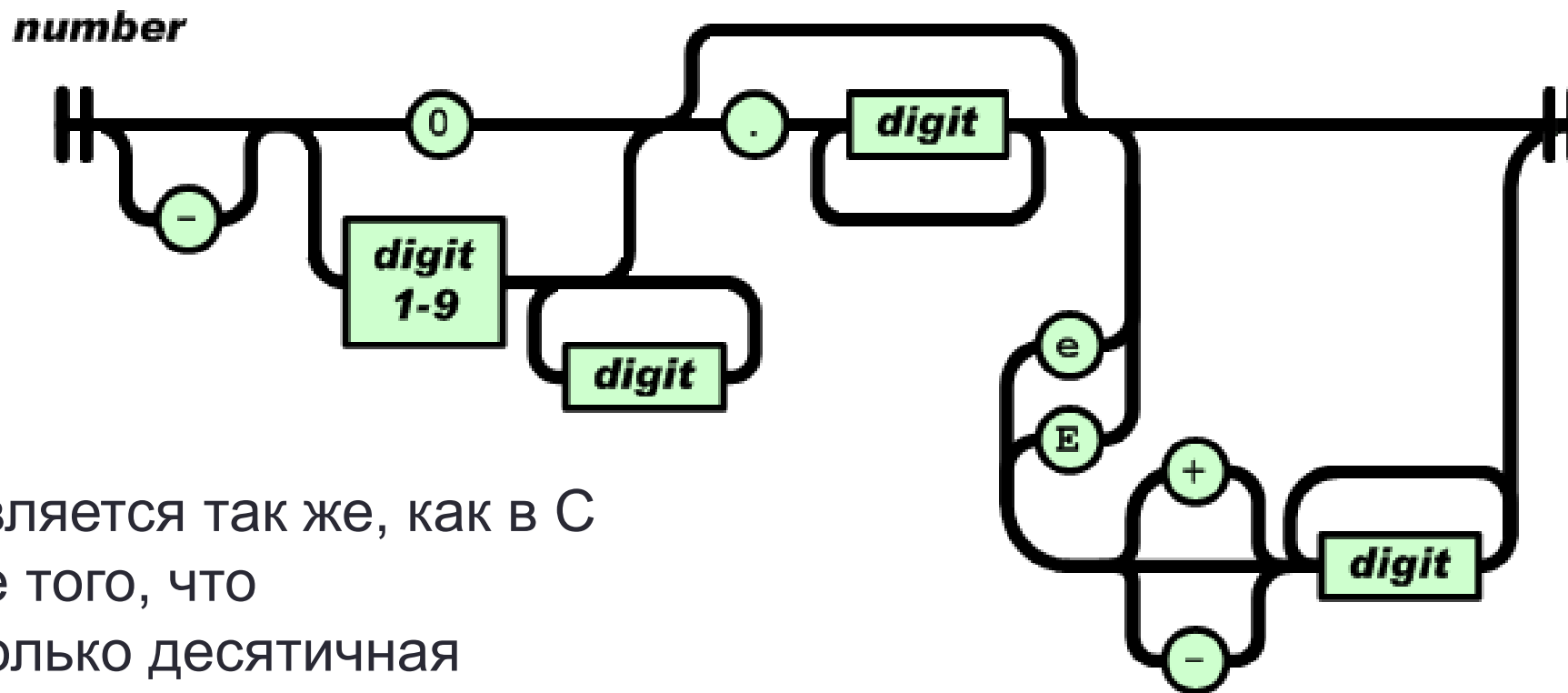


JSON – Строка. (QJsonValue)

- *Строка* - коллекция нуля или больше символов Unicode, заключенная в двойные кавычки, используя \ (обратную косую черту) в качестве символа экранирования. Символ представляется как односимвольная строка. Похожий синтаксис используется в C и Java.



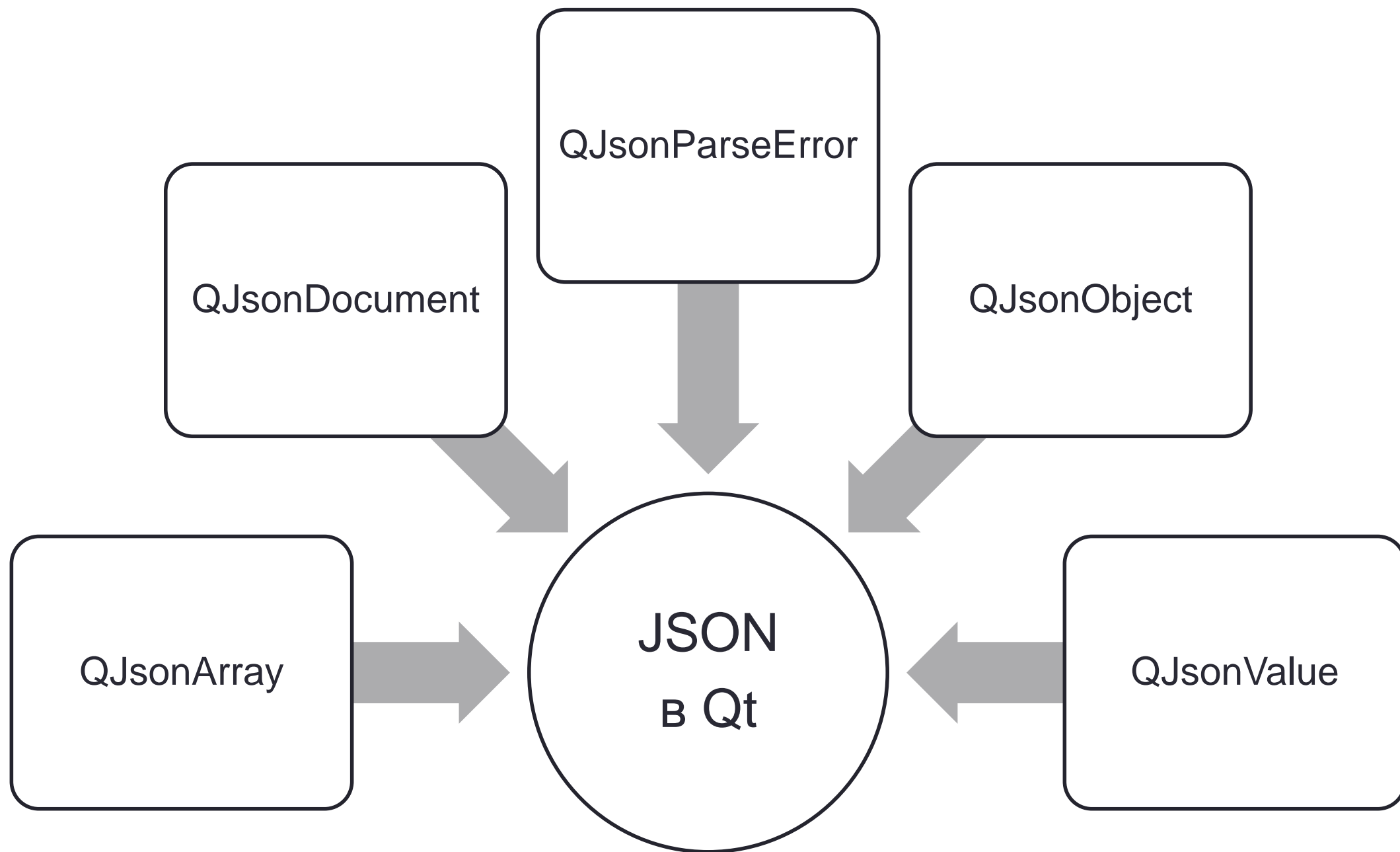
JSON – Число. (QJsonValue)



Число представляется так же, как в C или Java, кроме того, что используется только десятичная система счисления.

JSON – Пример

```
{  
  "firstName": "Иван",  
  "lastName": "Иванов",  
  "address": {  
    "streetAddress": "Московское ш., 101, кв.101",  
    "city": "Ленинград",  
    "postalCode": 101101  
  },  
  "phoneNumbers": [  
    "812 123-1234",  
    "916 123-4567"  
  ]  
}
```



QJsonArray

QJsonArray - класс для работы с массивами JSON.

- Элементы массива – объекты QJsonValue.
- QJsonArray может быть преобразован из и в объекты QVariantList, QStringList.
- Массив можно преобразовывать в текст и из текста, используя класс QJsonDocument.

Пример создания массива:

```
QJsonArray array = { 1, 2.2, QString() };
```

Добавление значения:

```
QJsonArray::append(const QJsonValue &value)
```

QJsonObject

- Класс для работы с объектами JSON. Данные организованы как ассоциативный массив из пар (*ключ (QString), значение(QJsonValue)*)
- Можно преобразовывать из и в объект класса QVariantMap;
- Массив можно преобразовывать в текст и из текста, используя класс QJsonDocument.
- *Пример создания:*
- ```
QJsonObject object {
 • {"property1", 1},
 • {"property2", 2}
 • };
```

Добавление элемента

iterator QJsonObject::insert(const QString &key, const QJsonValue &value)

# QJsonValue

QJsonValue - класс для работы со значениями JSON - объектов, массивов, строк.

- Можно преобразовывать из и в объект класса QVariant;

QJsonValue работает с 6 типами данных:

- `bool QJsonValue::Bool`
- `double QJsonValue::Double`
- `string QJsonValue::String`
- `array QJsonValue::Array`
- `object QJsonValue::Object`
- `null QJsonValue::Null`

# QJsonDocument

- Класс для чтения и записи JSON – «документов».
- Обертка к JSON – документу позволяет производить чтение и запись в формате JSON
- Преобразование текста в QJsonDocument – метод `QJsonDocument::fromJson()`
- Преобразование QJsonDocument в текст – метод `QJsonDocument::toJson()`
- Создание QJsonDocument может производиться также с использованием методов `fromBinaryData()` or `fromRawData()`.

# QJsonParseError

- Класс для сообщения ошибок парсинга JSON – сообщений, документов

# Debug и логирование данных

Глобальные макросы Qt для записи warning и debug сообщений:

- `QDebug()`
  - `qInfo()`
  - `qWarning()`
  - `qCritical()`
  - `qFatal()`
- 
- \*для работы с ними нужно подключить соответствующий заголовочный файл `<QDebug>..etc`



# Как можно использовать эти макросы?

Printf() – подобный вывод:

```
qDebug("hello");
qWarning("hello!!");
qCritical("HELLO!?!");
qFatal("giving up :(");
```

```
QDebug("Items in list: %d", myList.size());
```

Stream - вывод

```
· qInfo() << "config.json opened";
·
· qInfo() << "ip:port = " << obj.value("ip").toVariant().toString() << \
· ":" << obj.value("port").toVariant().toString();
· qInfo() << "frequency = " << obj.value("frequency").toVariant().toString();
```

# Вывод данных можно настраивать

В файле описания проекта проект.pro можно включить/ выключить определенный тип сообщений:

```
DEFINES+=\
QT_NO_DEBUG_OUTPUT, \
QT_NO_INFO_OUTPUT, \
QT_NO_WARNING_OUTPUT
```

# Можно сделать свой `messageHandler`

`messageHandler()`:

- Функция, которая обрабатывает все сообщения `QDebug()`, `qInfo()`, `qWarning()`, `qFatal()`;
- Такая функция одна в программе
- В этой функции отслеживается состояние переменных `QT_NO_DEBUG_OUTPUT` и т.п.
- Для того, чтобы установить свой `messageHandler` используют метод `qInstallMessageHandler()`

# Как сделать свой messageHandler?

1. Создать в main.cpp функцию, которая имеет следующую сигнатуру:  
void myMessageOutput([QtMsgType](#) type, const [QMessageLogContext](#) &context, const [QString](#) &msg)

| QtMsgType type<br>Тип сообщения                                        | const <a href="#">QMessageLogContext</a> &context<br>Контекст сообщения | const <a href="#">QString</a> &msg<br>Само сообщение |
|------------------------------------------------------------------------|-------------------------------------------------------------------------|------------------------------------------------------|
| QtInfoMsg<br>QtDebugMsg<br>QtWarningMsg<br>QtCriticalMsg<br>QtFatalMsg | Здесь доступна категория сообщения                                      |                                                      |

# Как сделать свой `messageHandler`?

2) Установить `messageHandler`

В `main.cpp`:

```
int main(int argc, char **argv) {
 qInstallMessageHandler(myMessageOutput);
 QApplication app(argc, argv);
 ...
 return app.exec();
}
```

# Создание собственных категорий логирования

QLoggingCategory – класс представляющий категорию логирования.

- По умолчанию, она выставлена в default
- Можно отключать/включать вывод информации для определённой категории
- Разные объекты могут иметь одну и ту же категорию логирования

QLoggingCategory

# Создание собственных категорий логирования

1. Объявить категорию в заголовочном файле (.h),  
[который является общим для разных частей программы]:

`Q_DECLARE_LOGGING_CATEGORY(name)`

*name* – название категории, которое будет использоваться дальше  
в коде

2. Определить категорию в исходном файле (.cpp):

1. `Q_LOGGING_CATEGORY(name, string)`

2. *name* – название категории,

3. *string* – строка, которая будет записываться в лог, если в `messageHandler` будет вывод с учетом категории

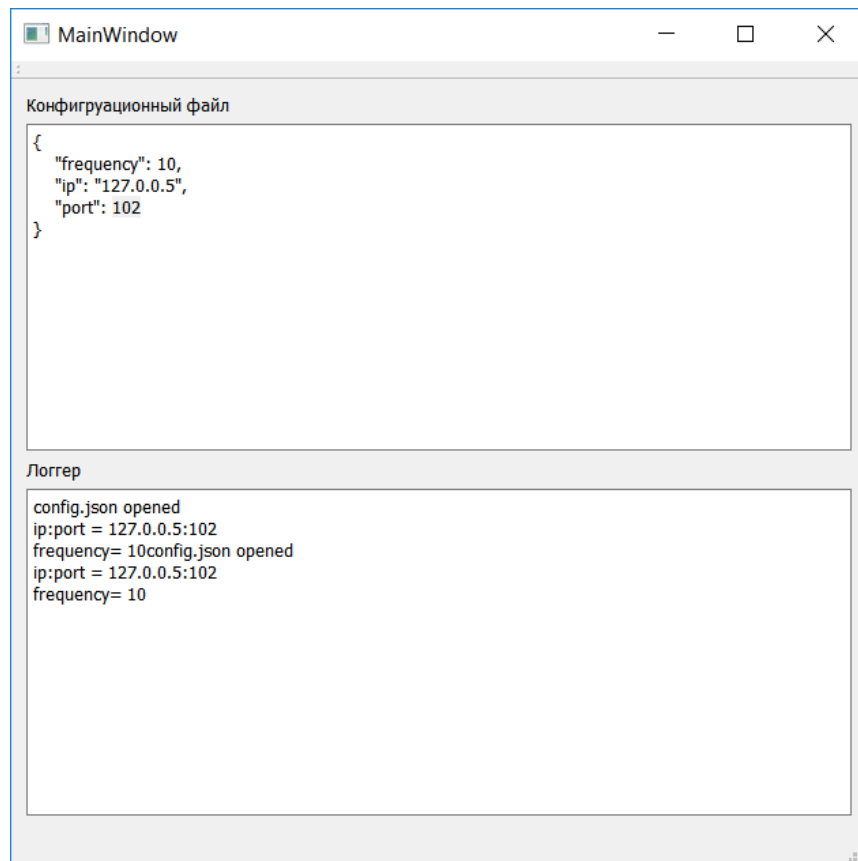
# Правила названия категорий

- Использовать только буквы, цифры и точки
- Использовать точки для структурирования и обобщения категорий:  
`category.subcategory.subsubcategory[...]`
  - Пример : “devices.usb”
- В названиях категорий избегать упоминания «debug, info, warning, and critical»
- Названия категорий с префиксом qt зарезервированы для модулей Qt
- Далее можете включать/отключать логирование тех или иных данных:  
`QT_LOGGING_RULES=«category.subcategory=true»`



# JSON. Практическая часть 1.

Создадим приложение, которое считывает свою конфигурацию из файла в формате JSON и записывает в файл своё текущее состояние.



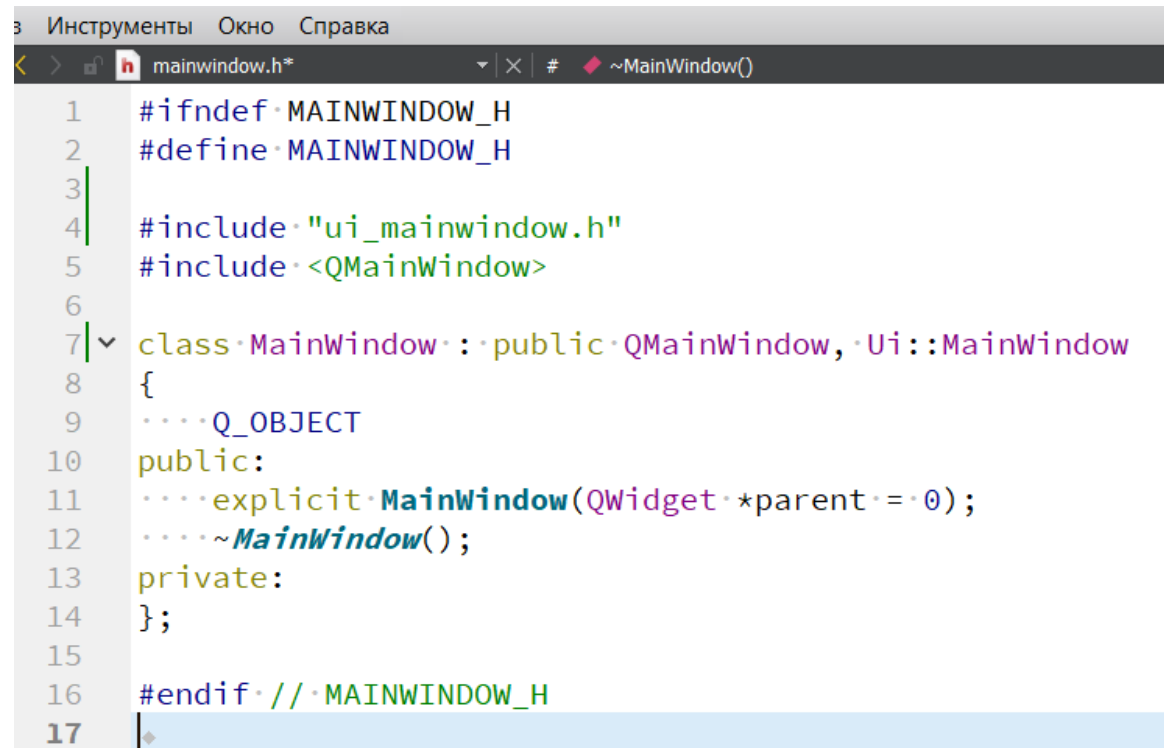
# QFile. Практическая часть 1.

В проекте JSON:

- Конфигурационный файл – файл с названием config.json;
  - Файл для логирования текущего состояния программы logfile.txt.
1. Конфигурационный файл необходимо разместить в директории сборки проекта.

# QFile. Практическая часть 1.

## Mainwindow.h

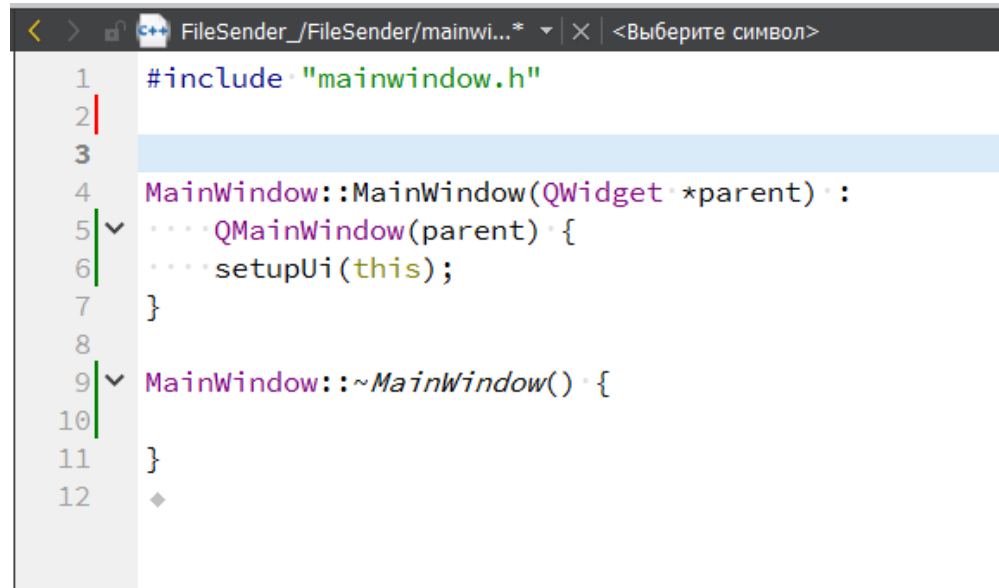


```
Инструменты Окно Справка
mainwindow.h* ~MainWindow()
1 #ifndef MAINWINDOW_H
2 #define MAINWINDOW_H
3
4 #include "ui_mainwindow.h"
5 #include <QMainWindow>
6
7 class MainWindow : public QMainWindow, Ui::MainWindow
8 {
9 ... Q_OBJECT
10 public:
11 ... explicit MainWindow(QWidget *parent = 0);
12 ... ~MainWindow();
13 private:
14 };
15
16 #endif // MAINWINDOW_H
17
```

Вид класса MainWindow для тех, кто предпочитает множественное наследование формы)))

# QFile. Практическая часть 1.

Mainwindow.cpp



```
< > C++ FileSender_/FileSender/mainwi... * | X | <Выберите символ>
1 #include "mainwindow.h"
2
3
4 MainWindow::MainWindow(QWidget *parent) :
5 QMainWindow(parent) {
6 setupUi(this);
7 }
8
9 MainWindow::~MainWindow() {}
10
11 }
12 ◆
```

Вид класса MainWindow для тех, кто предпочитает множественное наследование формы

# QFile. Практическая часть 1.

Mainwindow.ui

Пишите здесь

Конфигурационный файл

Форма для вывода содержимого конфига

Логгер

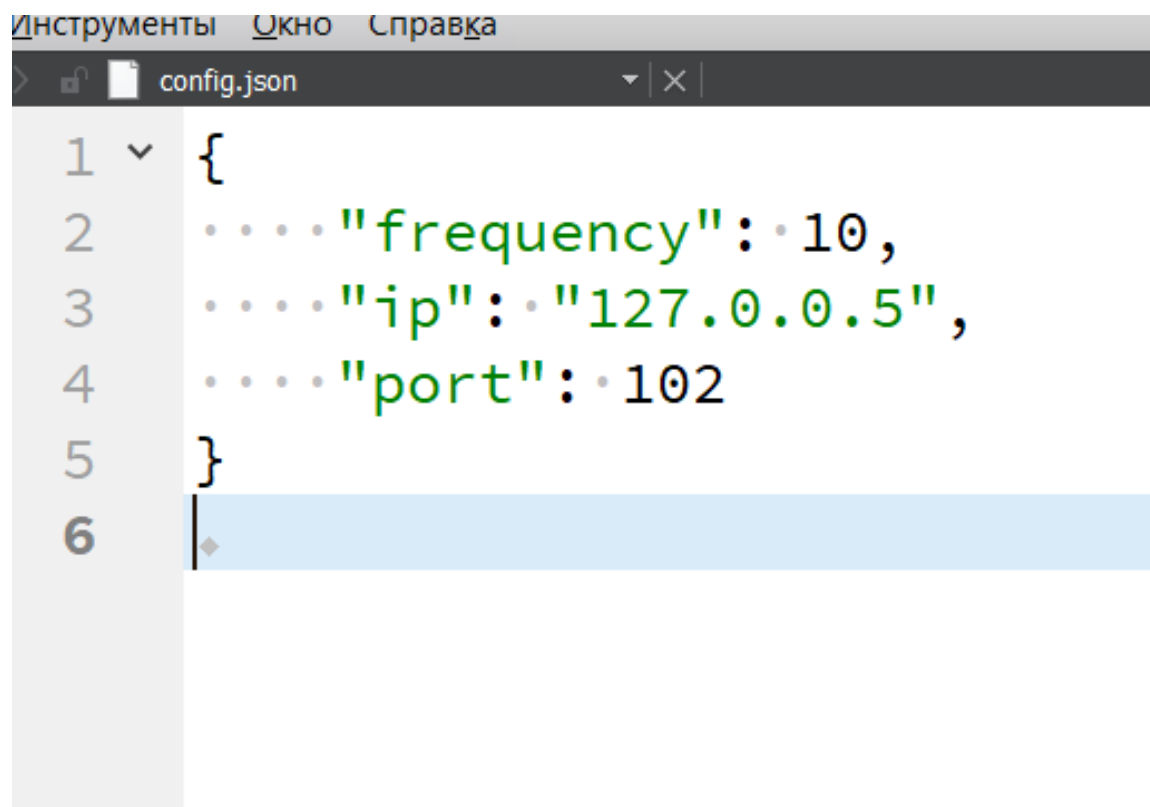
Форма для вывода содержимого лог-файла

Названия виджетов формы

| Объект          | Класс        |
|-----------------|--------------|
| ▼ MainWindow    | QMainWindow  |
| ▼ centralWidget | QWidget      |
| lblConf         | QLabel       |
| lblLog          | QLabel       |
| txtConfig       | QTextBrowser |
| txtLog          | QTextBrowser |
| menuBar         | QMenuBar     |
| mainToolBar     | QToolBar     |
| statusBar       | QStatusBar   |

# QFile. Практическая часть 1.

Содержимое конфигурационного файла



The screenshot shows a code editor window with a menu bar containing 'Инструменты', 'Окно', and 'Справка'. The title bar of the editor shows 'config.json'. The code is written in JSON format and is as follows:

```
1 {
2 "frequency": 10,
3 "ip": "127.0.0.5",
4 "port": 102
5 }
6
```

The line numbers 1 through 6 are visible on the left side of the editor. The code is color-coded: curly braces and strings are black, while the property names and values are green. A light blue horizontal bar is visible at the bottom of the editor area.

# QFile. Практическая часть 1.

Mainwindow.cpp

```
1 #include "mainwindow.h"
2 #include <QJsonObject>
3 #include <QJsonDocument>
4 #include <QDebug>
5 #include <qlogging.h>
6 #include <QFile>
7
8 MainWindow::MainWindow(QWidget *parent) :
9 ... QMainWindow(parent)
10 {
11 ... setupUi(this);
12
13 ... //Создадим объект для работы с JSON конфигом
14 ... QJsonDocument jsonDoc;
15
16 ... //откроем лог-файл для чтения
17 ... QFile logFile("logfile.txt");
18 ...
19 ... //создадим объект файла для открытия config.json
20 ... QFile configFile("config.json");
21 ... if (configFile.exists() && logFile.exists()) { //проверяем существование файлов
22 ... //если файлы существуют, то открываем их для чтения и работаем
23 ... configFile.open(QFile::ReadOnly);
24 ... logFile.open(QFile::ReadOnly);
25 ... //запишем информацию об успешном открытии файла
26 ... qDebug() << "config.json opened";
```

# QFile. Практическая часть 1.

Mainwindow.cpp

```
..... //считаем конфигурационные данные из файла
..... //используя метод fromJson, который преобразует текст (QByteArray)
..... //в формат JSON
..... jsonDoc = QJsonDocument().fromJson(configFile.readAll());
..... //выведем содержимое считанное из конфига в текстовый браузер
..... //используя метод toJson()
..... txtConfig->setText(jsonDoc.toJson());

..... //залогируем принятую информацию:
..... //для этого сначала создадим объект JSON, в который примем
..... //данные из конфига
..... QJsonObject obj = jsonDoc.object();
..... //запишем в лог-файл принятые значение ip адреса и порта:
..... qInfo()<<"ip:port="<<obj.value("ip").toVariant().toString()<<\
..... ":"<<obj.value("port").toVariant().toString();
..... qInfo()<<"frequency="<<obj.value("frequency").toVariant().toString();

..... txtLog->setText(logFile.readAll());
..... //закрываем конфиг после работы с ним
..... configFile.close();
..... logFile.close();
..... }
```



# QFile. Практическая часть 1.

```
 ...else{//если файл не существует, то уведомляем об ошибке
 ...if(!logFile.exists())·qFatal("No file: logfile.txt");
 ...if(!configFile.exists())·qFatal("No file: config.json!");
 ...}
}
```

# QFile. Практическая часть 1.

```
#include "mainwindow.h"
#include <QApplication>
#include <QFile>
#include <QTextStream>
#include <QDebug>

QScopedPointer<QFile> p_logFile;

void messageHandler(QtMsgType type, const QMessageLogContext &context, const QString &msg);

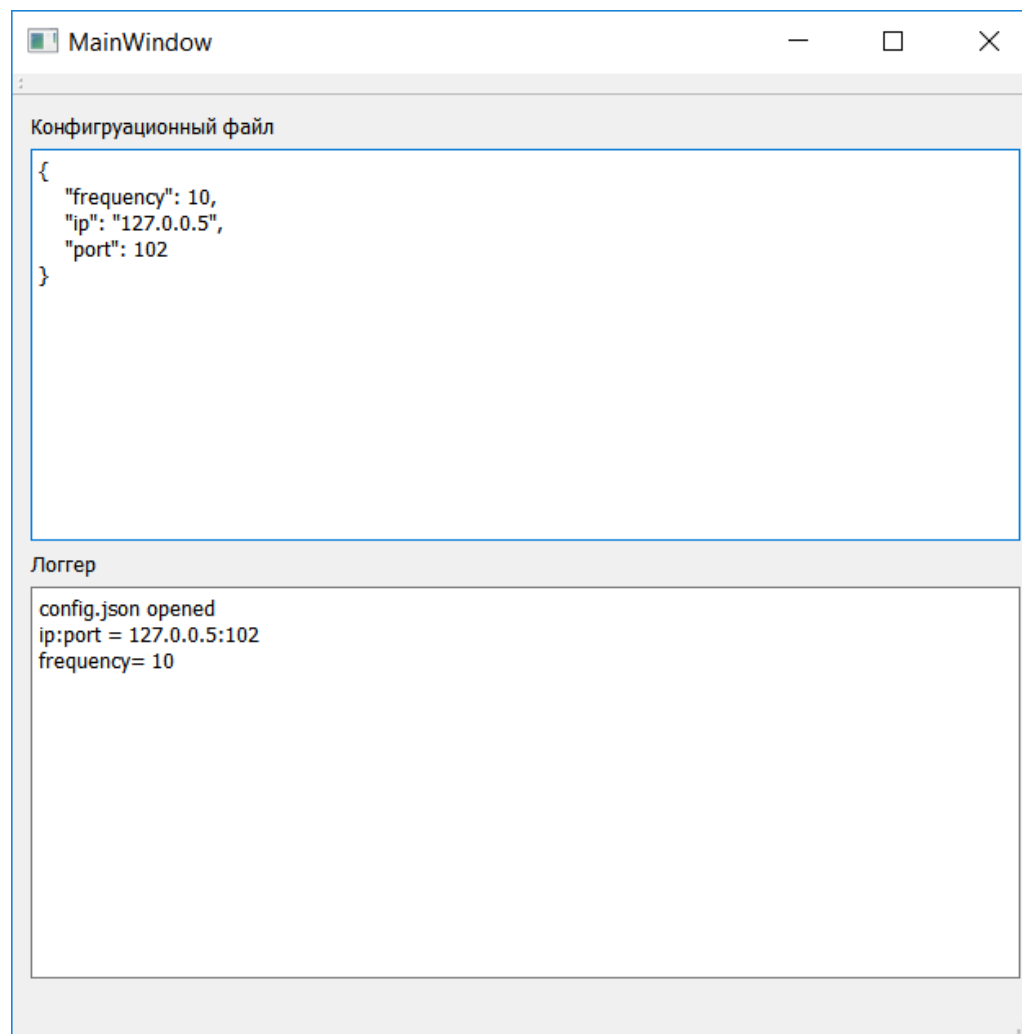
int main(int argc, char *argv[])
{
 ... p_logFile.reset(new QFile("logfile.txt"));
 ... p_logFile.data()->open(QFile::Append | QFile::Text);
 ... qInstallMessageHandler(messageHandler);
 ... QApplication a(argc, argv);
 ... MainWindow w;
 ... w.show();
 ... return a.exec();
}
```

# QFile. Практическая часть 1.

```
21
22 void messageHandler(QtMsgType type, const QMessageLogContext &context, const QString &msg){
23
24 QTextStream out(p_logFile.data());
25 switch(type){
26 case QtDebugMsg: out<<"Debug: "; break;
27 case QtInfoMsg: out<<"Info: "; break;
28 case QtWarningMsg: out<<"Warning: "; break;
29 case QtCriticalMsg: out<<"Critical error: "; break;
30 case QtFatalMsg: out<<"Fatal error: "; break;
31 default: out<<"Unknown: "; break;
32 }
33 //записываем само сообщение
34 out<<context.category<<" : "<<msg<<endl;
35 out.flush();
36 }
37
```

# QFile. Практическая часть 1.

Результат:



# QFile. Практическая часть 1.

- Дописать код проекта таким образом, чтобы при логировании данных фиксировались дата и время записи. ( классы QDateTime, QDateTime)