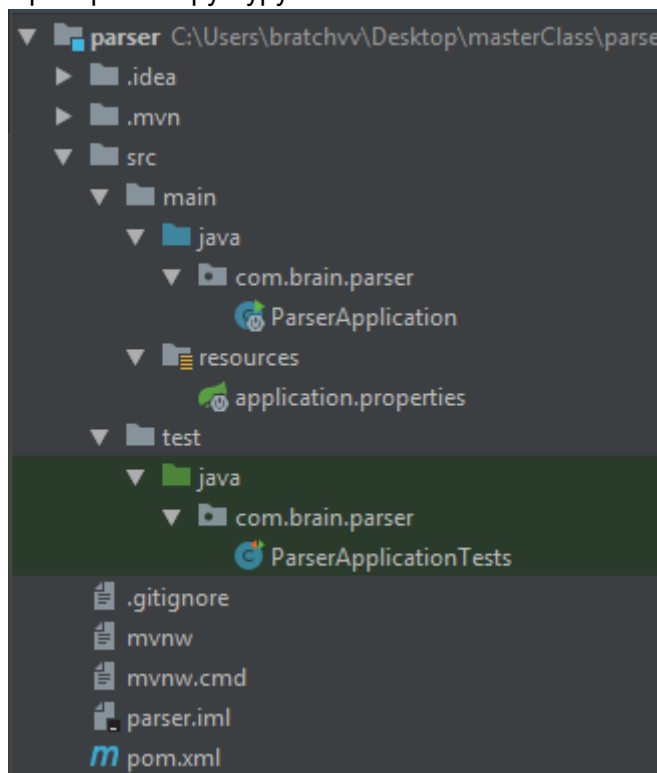


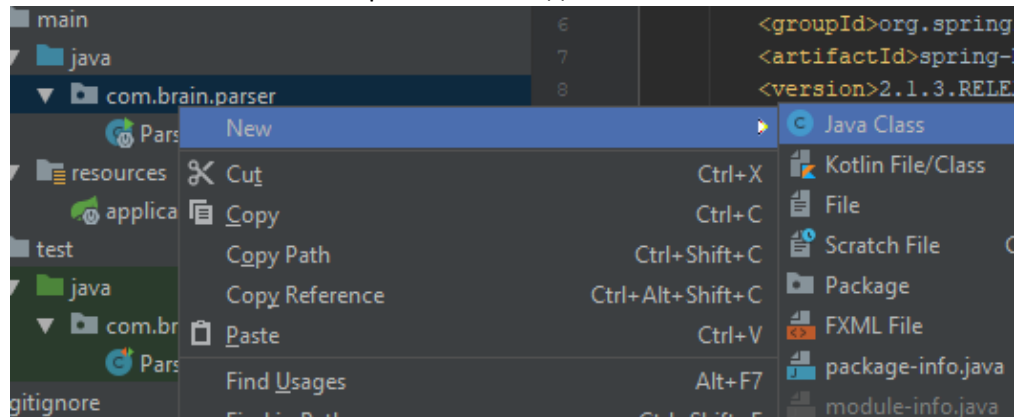
test github repo: <https://github.com/Bratchvv/brainLearnParser>

- 1) Гуглим «spring boot generator».
- 2) Открываем <https://start.spring.io/>
- 3) Оставляем все по умолчанию.
  - a. По желанию, можно поменять
    - i. `group -> "com.brain"`
    - ii. `parser -> "parser"`
- 4) Нажимаем кнопку «Generate Project»
- 5) Выбираем папку, куда сохранить архив с каркасом проекта
- 6) Распаковываем содержимое архива.
- 7) Открываем проект с помощью IntelliJ IDEA (file -> open -> {папка с проектом} -> pom.xml)
- 8) Выбираем вариант «Open as project»
- 9) Проверяем работу сгенерированного проекта.
  - a. Проверяем структуру:



- b. Находим enter-class : `src/main/java/com/brain/parser/ParserApplication.java`
  - c. Запускаем приложение. ПКМ на классе -> Run 'ParserApplication'
  - d. Проверяем консоль на отсутствие ошибок
- 10) Подключаем библиотеку JSOUP
  - a. Находим JSOUP в maven repository. Гуглим «jsoup maven»
  - b. Открываем <https://mvnrepository.com/artifact/org.jsoup/jsoup/1.9.1>
  - c. Копируем dependency
  - d. Открываем pom.xml в корне нашего проекта
  - e. Добавляем скопированную dependency в блок dependencies, рядом с другими.
  - f. Появится Push уведомление "Maven projects need to be imported". Выбираем «Import Changes»
- 11) Теперь у нас есть все необходимое для создания парсера сайтов. Выбираем сайт, например, <http://ex-fs.com/>
- 12) Создаем класс для парсинга сайта ex-fs.

- a. ПКМ на имени пакета в котором хотим создать класс:



- b. Указываем параметры класса. Имя ExFsParser, тип оставляем Class

- c. Проверяем наличие нового класса в пакете.

### 13) Заполняем класс «заглушками методов»

- a. создаем будущий метод, для запуска парсера:

```
public void run() {  
    // todo  
}
```

- b. создаем будущий метод, для сохранения распарсенных картинок сайта

```
private void save() {  
    // todo  
}
```

- c. создаем будущий метод парсинга html страницы, для получения нужных катинок

```
private List<String> parse() {  
    // todo  
    return null;  
}
```

после написания этого метода, компилятор java попросит импортировать сторонний класс List. Устанавливаем курсор мыши на слово List (на середину), нажимаем сочетание клавиш alt + Enter.

После этого в классе появится новый импорт: `import java.util.List;`

### 14) Вызываем заглушки методов

- Открываем класс ParserApplication
- Находим метод: `public static void main(...)`
- Дописываем в конец этого метода создание объекта нашего парсера  
`ExFsParser parser = new ExFsParser();`
- Вызываем метод запуска парсера:  
`parser.run();`
- Пробуем запускать, аналогично п. 9.с. Вывод должен быть аналогичным.

### 15) Добавляем в методы класса ExFsParser тестовые сообщения для имитации работы

- a. В методе run, вместо комментария «//todo», пишем:

```
System.out.println("Executing run method");
```

Примечание: в IntelliJ IDEA существует множество полезных функций, одна из них это сокращения часто используемых команд, например sout для нашего случая.

- b. В методе `save`, вместо комментария «`//todo`», пишем:  
`System.out.println("Try to save");`
- c. В методе `parse`, вместо комментария «`//todo`», пишем:  
`System.out.println("Parsing page");`
- d. Пробуем запускать. В логе можно найти строчку "Executing run method".  
Остальных нет, потому что мы нигде не вызывали их методы.
- e. В методе `run` вызываем метод `parse()` и `save()`

```
public class ExFsParser {  
    public void run() {  
        System.out.println("Executing run method");  
        parse();  
        save();  
    }  
  
    private List<String> parse() {  
        System.out.println("Parsing page");  
        return null;  
    }  
  
    private void save() {  
        System.out.println("Save");  
    }  
}
```

- f. Пробуем запустить еще раз. Теперь на консоле выводятся все нужные сообщения.  
Наше приложение обрело свой функциональный каркас.

#### 16) Реализуем основной метод для парсинга. Метод `parse()`;

- a. Создаем строку с URL нужной страницы.  
`String url = "http://ex-fs.com/page/1";`
- b. Создаем переменную для хранения распарсеных ссылок на картинки:  
`List<String> result = new ArrayList<>();`

Примечание класс `ArrayList` еще не импортирован, его нужно импортировать, аналогично п. 13.с.

- c. Выполняем чтение HTML страницы с помощью библиотеки JSOUP:  
`Document document = Jsoup.connect(url).get();`

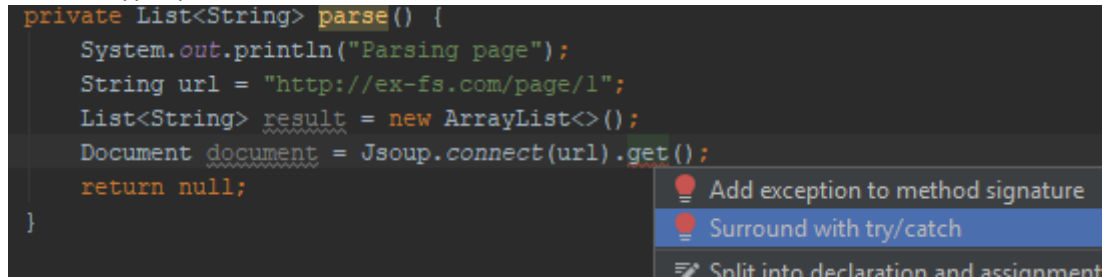
Примечание: классы `Document` и `Jsoup` необходимо импортировать, аналогично предыдущему пункту.

Импорты:

```
import org.jsoup.Jsoup;  
import org.jsoup.nodes.Document;
```

- d. Обращаем внимание что IDE подчеркнула ошибкой метод `get()`; IntelliJ IDEA также часто дает подсказки и предлагает автоматическое решение некоторых проблем.

Ставим курсор на этот метод и нажимаем `alt+Enter`.



Выбираем пункт “Surround with try/catch”.

Проверяем наличие изменений.

Блок `try...catch` нужен для обработки возможных ошибок.

- e. Итак, получение HTML документа у нас работает. Пора написать нужный селектор: `Elements links = document.select(".custom-poster img[src]");`

Примечание: `Elements` нужно импортировать в класс, уже известным способом.

- f. Строка `".custom-poster img[src]"` – это `css` селектор. Открыв «исходники» `html` страницы (в `Chrome` -> `ctrl + U`), попробуем найти (`ctrl + F`) элементы с таким `css` классом. Ищем текст «`custom-poster`».

Находим примерно такое:

```
</div><div class="custom-post">
<div class="custom-poster">
<a href="http://ex-fs.com/ex_film/108096-chernyy-sad.html"

Чёрный сад
</a>
<div class="custom-update">HD</div>
    <div class="custom-text">
<a href="http://ex-fs.com/ex_film/">Фильмы на ex ua fs to
</div>
</div>
```

В `html` коде видно, что в блоках с классом «`custom-poster`», лежат нужные нам ссылки на картинки (в теге `img` у атрибута `src`).

- g. Класс `Elements` представляет собой список этих элементов на странице. Поэтому можно проитерироваться по этому списку для получения нужных картинок. Пишем цикл `for`:

```
for (Element link : links) {
    String imgUrl = "http://ex-fs.com/" + link.attr("src");
    result.add(imgUrl); // добавляем URL картинки к результатам
}
```

- h. Для вывода ссылок в консоль добавляем еще один “sout” в конец тела цикла (после result.add(...)):  
`System.out.println(imgUrl);`
- i. Наш текущий метод должен возвращать список URL на картинки, для этого корректируем возвращаемое значение: заменяем “return null;” на “return result;”
- j. Запускаем приложения, видим в консоле список ссылок на картинки. Можно открыть парочку.

Метод parse()

```
/**
 * Parse film posters of ex-fs.com site page.
 * @return img URLs list
 */
private List<String> parse() {
    System.out.println("Parsing page");
    String url = "http://ex-fs.com/page/1"; // URL страницы
    List<String> result = new ArrayList<>(); // пустой объект списка
    try { // блок try...catch для обработки ошибок
        Document document = Jsoup.connect(url).get(); // объект HTML страницы
        Elements links = document.select("cssQuery: ".custom-poster img[src]"); // css селектор
        for (Element link : links) { // цикл по выбранным элементам
            // строим URL картинки:
            // 1. берем host сайта
            // 2. добавляем к нему значение источника картинки (в теге img у атрибута src)
            String imgUrl = "http://ex-fs.com/" + link.attr("attributeKey: "src");
            result.add(imgUrl); // добавляем URL картинки к результатам
            System.out.println(imgUrl);
        }
    } catch (IOException e) { // обработка возможной ошибки
        e.printStackTrace();
    }
    return result; // возвращение результата метода
}
```

- 17) Теперь мы умеем парсить страницу сайта с картинками, теперь попробуем сохранять эти картинки себе на диск.
- a. Метод save должен уметь сохранять картинки из полученных URL. Для того чтобы работать с этими ссылками на картинки в этом методе нужно пробросить их в качестве параметра. Добавим параметр в метод:  
`private void save(List<String> urls) { ... }`
  - b. Сохранять мы можем только по одной картинке, поэтому необходимо создать цикл для итерирования по нашему списку. Примечание: еще одно сокращение от IDEA – **fori** .  
Эта команда создает макет цикла for, необходимо его только немного модифицировать:  
`for (int i = 0; i < urls.size(); i++) {  
 // todo  
}`

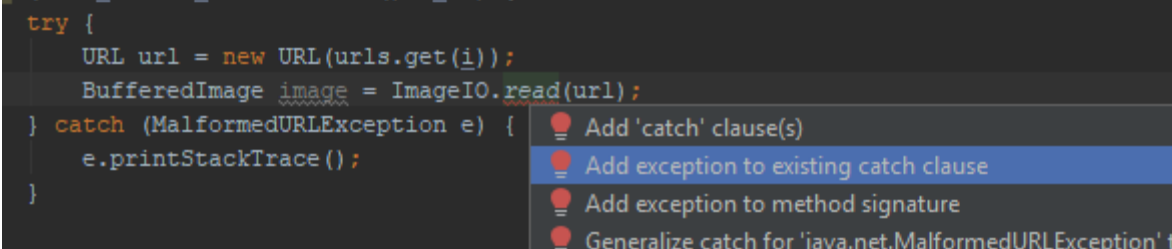
- c. Начинаем заполнять тело цикла. Для работы с URL в Java есть одноименный класс:
- ```
String imgUrl = urls.get(i);  
URL url = new URL(imgUrl);
```

Где `urls.get(i)` – получение *i*-го элемента списка, т.е. наша строка с URL картинки.

- d. Создание объекта URL требует обработки исключения. Оборачиваем в `try...catch`, аналогично п.16.d.
- e. После создания URL, добавляем строку для чтения картинки, по URL:
- ```
BufferedImage image = ImageIO.read(url);
```

Примечание продолжаем писать в текущем блоке `try...catch`.

- f. Метод `read(url)` также требует обработки ошибки, но так как вызов этого метода уже находится в блоке `try...catch` IDE позволяет расширить текущий блок обработки:



```
try {  
    URL url = new URL(urls.get(i));  
    BufferedImage image = ImageIO.read(url);  
} catch (MalformedURLException e) {  
    e.printStackTrace();  
}
```

IDE suggestions:

- Add 'catch' clause(s)
- Add exception to existing catch clause
- Add exception to method signature
- Generalize catch for 'java.net.MalformedURLException'

После этого блок `catch` немного изменится.

- g. Следующим шагом будет подготовка названия типа будущей картинки:

```
String imgType = imgUrl.substring(imgUrl.lastIndexOf('.') + 1);
```

Где метод `substring` возвращает часть строки от начала указанного номера символа и до конца, а `imgUrl.lastIndexOf('.')` возвращает номер последнего символа '.' в строке.

Таким образом мы получаем тип картинки, в нашем случае это `jpg`.

- h. Следующим шагом будет сохранение картинки с новым именем:

```
ImageIO.write(image, imgType, new File("parsed/img_" + i + '.' + imgType));
```

Где:

`image` – «байты» нашей картинки

`imgType` – полученный ранее тип картинки

`new File("parsed/img_" + i + '.' + imgType)` – создание нового файла в папке `parsed` с новым именем.

Метод save();

```
/**
 * Save all images from URL to disk.
 * @param urls URLs list.
 */
private void save(List<String> urls) {
    System.out.println("Save");
    for (int i = 0; i < urls.size(); i++) { // цикл по списку из ссылок на картинки
        try {
            String imgUrl = urls.get(i); // получаем i-ю строку с ссылкой
            URL url = new URL(urls.get(i)); // создаем объект URL из строки
            BufferedImage image = ImageIO.read(url); // читаем картинку
            String imgType = imgUrl.substring(imgUrl.lastIndexOf( 'h' ) + 1); // выделяем тип
            //Сохраняем картинку на диск с новым именем
            ImageIO.write(image, imgType, new File( pathname: "parsed/img_" + i + '.' + imgType));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

18) Теперь поправим метод run().

- a. Ранее у нас не было параметра в этом методе, после его добавления Java показывает ошибку на вызове этого метода.  
Поэтому в него нужно передать нужный список, а именно такой список возвращает предыдущий метод parse(). То есть можно передать в метод save() результат метода parse():

```
List<String> urls = parse();
save(urls);
или вообще:
save(parse());
```

- b. Мы сохраняем картинки в папку parsed, но такой папки у нас нет, и если попробуем запустить приложение - получим ошибку, поэтому необходимо создать эту папку.  
`new File("parsed").mkdir();`

Причем создать папку нужно до вызова метода save.

Метод run

```
/**
 * Completely run parser.
 * Parse ex-fs.com site page and save it to 'parsed' folder
 */
public void run() {
    System.out.println("Executing run method");
    new File( pathname: "parsed").mkdir(); // создаем папку в корне проекта
    List<String> urls = parse(); // вызываем метод парсинга URL картинок
    save(urls); // Сохранение картинок из списка ссылок.
}
```

19) Запускаем приложение и проверяем работу парсера.