CLEAN CODE

ЧИСТЫЙ КОД

ПЛАН ЛЕКЦИИ

1. Чистый код
2. Способы достижения качества кода
3. Типичные ошибки
4. Инструменты

ЧАСТЬ 1. ЧИСТЫЙ КОД

НАЧНЁМ С ПРИМЕРА

ДАН ОБРАЗЕЦ КОДА. НЕОБХОДИМО ОТВЕТИТЬ НА ВОПРОСЫ:

- Что делает это приложение?
- Какой формат входных/выходных значений?
- Можно ли легко(т. е. без значительных изенений кода) "научить" приложение выводить результат на консоль/FTP сервер/...?

ПОЧЕМУ ЧИТАБЕЛЬНОСТЬ КОДА ВАЖНА?

- Легко читать. (c) K.O.
- Легко понимать
- Легко выявлять ошибки
- Легко поддерживать существующий код

ЗАЧЕМ ВСЁ ЭТО НУЖНО?

- Программирование - это вид деятельности по созданию качественного продукта с минимальными затратами в разумные сроки.
- Для сокращения сроков разработка ПО осуществляется командами программистов => код должен быть понятен коллегам.
- Хороший код легко **использовать повторно** = прямой путь к снижению издержек.
- **"Design for change"** - объективная необходимость. Понятный и качественный код легко менять.
- Развитие функциональности и устранение дефектов в хорошем коде не требует больших затрат.

ПРИЗНАКИ ПЛОХОЙ ЧИТАБЕЛЬНОСТИ КОДА:

- Длинный метод или класс
- Длинные коментарии внутри метода
- Ненужный код
- Магические числа
- Абстрактные названия
- Цепочки объявления переменных
- Неявная область видимости
- Излишние префиксы
- Небрежное форматирование
- ...

ПРОБЛЕМА: ДЛИННЫЙ МЕТОД ИЛИ КЛАСС

- Длинный метод(более нескольких экранов) - это плохо
- Класс, содержащий большое количество разнородных методов - признак плохого проектирования
- Исходный код программы в одном файле - это **очень** плохо (!)

ПРОБЛЕМА: НЕНУЖНЫЙ КОД

Нужно исправлять код, если:

- Класс не используется
- Метод не используется
- Переменная не используется
- Параметр метода не используется
- Поле класса не используется
- Классы, перечисленные в import сеции, не используются

ПРОБЛЕМА: ДЛИННЫЕ КОМЕНТАРИИ ВНУТРИ МЕТОДА

```
int id = 5;
int shift = 3;
int result = id << shift;

// преобразовываем результат в строку
// с помощью операции конкатенации с другой строкой
// и читаем из файла сохраненный сериализованный
// объект.
String fileName = result + "_student";
```

ПРОБЛЕМА: МАГИЧЕСКИЕ ЧИСЛА (1/3)

```
// wrong
for (int i = 1; i < 52; i++) {
    int j = i + new Random().nextInt(53 - i) - 1;
    Collections.swap(list, i, j);
}
```

ПРОБЛЕМА: МАГИЧЕСКИЕ ЧИСЛА (2/3)

```
// better
int maxDeckSize = 52;
for (int i = 1; i < maxDeckSize; i++) {
    int j = i + new Random().nextInt(maxDeckSize + 1 - i) - 1;
    Collections.swap(list, i, j);
}
```

ПРОБЛЕМА: МАГИЧЕСКИЕ ЧИСЛА (ЕЩЁ ОДИН ПРИМЕР)

```
// wrong
public void setPassword(String password) {
    if (password.length() > 20) {
        throw new InvalidParameterException("password");
    }
...
}

// correct
public static final int MAX_PASSWORD_SIZE = 20;

public void setPassword(String password) {
    if (password.length() > MAX_PASSWORD_SIZE) {
        throw new InvalidParameterException("Password is too long");
    }
...
}
```

ПРОБЛЕМА: АБСТРАКТНЫЕ НАЗВАНИЯ (1/2)

```
// wrong
int days;

// correct
int daysSinceCreation;
int daysSinceLastModification;
int durationInDays;
```

ПРОБЛЕМА: АБСТРАКТНЫЕ НАЗВАНИЯ (2/2)

Однако, короткие имена можно использовать в случае, когда время жизни переменной тоже коротко.

```
for (int i = 0; i < array.length; i++) {
    // Some code here
}
```

ПРОБЛЕМА: ЦЕПОЧКИ ОБЪЯВЛЕНИЯ ПЕРЕМЕННЫХ

```
// wrong
int id, count, year;

// correct
int id;
int count;
int year;
```

ПРОБЛЕМА: НЕЯВНАЯ ОБЛАСТЬ ВИДИМОСТИ (1/3)

```
// wrong
if (a == b)
    System.out.println("Where am I?");

// correct
if (a == b) {
    System.out.println("I'm inside of 'if'!");
}
```

ПРОБЛЕМА: НЕЯВНАЯ ОБЛАСТЬ ВИДИМОСТИ. ВОПРОС. (2/3)

```
public static void main(String[] args) {
    int counter = 0;
    for (int i = 0; i < 10; i++);
         counter++;

    System.out.println("Counter = " + counter);
}
```

Какой результат работы приложения?

Counter = 1

ПРОБЛЕМА: НЕЯВНАЯ ОБЛАСТЬ ВИДИМОСТИ(3/3)

java.util.ArrayList – форматирование сохранено

```
public void ensureCapacity(int minCapacity) {
    modCount++;
    int oldCapacity = elementData.length;
    if (minCapacity > oldCapacity) {
        Object oldData[] = elementData;
        int newCapacity = (oldCapacity * 3)/2 + 1;
        if (newCapacity < minCapacity)
            newCapacity = minCapacity;
            elementData = (E[])new Object[newCapacity];
            System.arraycopy(oldData, 0, elementData, 0, size);
    }
}
```

ПРОБЛЕМА: ИЗЛИШНИЕ ПРЕФИКСЫ

java.util.ArrayList – форматирование сохранено

```
private String mFindSubstring(String pInput, int pMaxLenght){
    String sResult = "";
    int iCounter = 0;

...
}
```

ПРОБЛЕМА: НЕБРЕЖНОЕ ФОРМАТИРОВАНИЕ(ПРОБЕЛЫ)

```
foo ( 3+5,9 )  ; // this is ugly
foo(3 + 5, 9); // much better
```

ПРОБЛЕМА: НЕБРЕЖНОЕ ФОРМАТИРОВАНИЕ(ВЫРАВНИВАНИЕ КОДА)

Нельзя располагать выражение и код на одном уровне

```
// wrong
if(someCondition) {processExec();}
else {goToErrorHandler();}

// correct
if (someCondition) {
    processExec();
} else {
    goToErrorHandler();
}
```

ЧАСТЬ 2. СПОСОБЫ ДОСТИЖЕНИЯ КАЧЕСТВА КОДА

ПРИЁМЫ ПОВЫШЕНИЯ ЧИТАБЕЛЬНОСТИ КОДА

- Правильное выставление пробелов
- Группировка логических выражений скобками
- Разбитие сложных выражений на более мелкие
- Группировка логически связанных блоков кода с помощью пустой строки
- Используйте принятый порядок объявлений в классе
- Самодокументируемый код
- Следование общепринятым стандартам и рекомендациям

ПРАВИЛЬНОЕ ВЫСТАВЛЕНИЕ ПРОБЕЛОВ

```
foo ( 3+5,9 ) ; // this is ugly
foo(3 + 5, 9); // much better
```

РУППИРОВКА ЛОГИЧЕСКИХ ВЫРАЖЕНИЙ СКОБКАМИ

```
foo(n, 4) < customerCount && parentId != null

(foo(n, 4) < customerCount) && (parentId != null)
```

РАЗБИТИЕ СЛОЖНЫХ ВЫРАЖЕНИЙ НА БОЛЕЕ МЕЛКИЕ

```
// wrong
int result = foo(id, Math.acos(12))*bar(13*foo(id,Math.asin(3)));

// correct
int a = foo(id, Math.acos(12));
int b = foo(id, Math.asin(3));
int c = bar(13 * b);
int result = a * c;
```

ГРУППИРОВКА ЛОГИЧЕСКИ СВЯЗАННЫХ БЛОКОВ КОДА С ПОМОЩЬЮ ПУСТОЙ СТРОКИ (1/2)

```
List<Integer> primes = new ArrayList<>(3);
primes.add(5);
primes.add(7);
primes.add(3);

Collections.sort(primes);
Collections.reverse(primes);

System.out.println(primes);
```

ГРУППИРОВКА ЛОГИЧЕСКИ СВЯЗАННЫХ БЛОКОВ КОДА С ПОМОЩЬЮ ПУСТОЙ СТРОКИ (2/2)

Но не стоит делать так:

```
doThis();
thenThis();
andThis();
andThisInTheEnd();
```

ИСПОЛЬЗУЙТЕ ПРИНЯТЫЙ ПОРЯДОК ОБЪЯВЛЕНИЙ В КЛАССЕ

1. Константы (public, protected, package, private)
2. Поля
3. Конструкторы
4. Методы
5. Внутренние классы(если они нужны)

САМОДОКУМЕНТИРУЕМЫЙ КОД (1/3)

```
public List<Customer> getCustomers(List<Form> customerForms) {
    List<Long> ids = new ArrayList<>(customerForms.size());
    for (Form form : customerForms) {
            ids.add(form.getCustomerId());
    }

    List<Customer> customers = customerDao.getCustomersByIds(ids);
    Iterator<Customer> iterator = customers.iterator();
    while (iterator.hasNext()) {
        Customer customer = iterator.next();
        if(customer.getFullName() == null) {
            iterator.remove();
        }
    }

    return customers;
```

САМОДОКУМЕНТИРУЕМЫЙ КОД (2/3)

```
public List<Customer> getCustomers(List<Form> customerForms) {
    List<Long> ids = exposeCustomerIds(customerForms);
    List<Customer> customers = customerDao.getCustomersByIds(ids);
    removeUnnamedCustomers(customers);

    return customers;
}
```

САМОДОКУМЕНТИРУЕМЫЙ КОД (3/3)

```
private void removeUnnamedCustomers(List<Customer> customers) {
    Iterator<Customer> iterator = customers.iterator();
    while (iterator.hasNext()) {
        Customer customer = iterator.next();
        if(customer.getFullName() == null) {
           iterator.remove();
        }
    }
}

private List<Long> exposeCustomerIds(List<Form> customerForms) {
    List<Long> ids = new ArrayList<Long>(customerForms.size());
    for (Form form : customerForms) {
         ids.add(form.getCustomerId());
    }
    return ids;
}
```

СЛЕДОВАНИЕ ОБЩЕПРИНЯТЫМ СТАНДАРТАМ И РЕКОМЕНДАЦИЯМ

- Google Java Style - всегда использовать
- Фаулер М., Бек К., Брант Д. и др. Рефакторинг: улучшение существующего кода - обязательно к прочтению
- Изучать образцы хорошего кода, например Spring Framework

ИМЕНОВАНИЕ

- Именование классов, методов, а главное переменных обычно занимает больше времени, чем написание самого кода. Именование очень важно для понимания чужого кода, т.к. читать код часто сложнее, чем писать
- Переменная, метод или класс должны иметь понятное название, по которому можно судить о их назначении
- Умеренная длина
- Спецификаторы – count и index вместо num

ИМЕНОВАНИЕ

Не нужно дублировать контекст класса в именах переменных и методов

```
// wrong
public class Customer {
    private String customerFirstName;
    private String customerLastName;
}

// correct
public class Customer {
    private String firstName;
    private String lastName;
}
```

ИМЕНОВАНИЕ

Не используйте излишние префиксы

```
// might be improved
List<Student> mySortedAndFilteredStudentList;

// better
List<Student> students;
```

ИМЕНОВАНИЕ

- Избегать артиклей и предлогов в именах – a, an, the, of, to и т.д.

```
int theHeightOfTheDoor; // wrong
int doorHeight;         // correct
```

- Не изобретайте ненужные сокращения. (e.g. FCNTL for File control)
- Не используйте венгерскую нотацию

ИМЕНОВАНИЕ

Избегайте абстрактных названий

```
// wrong
int days;

// correct
int daysSinceCreation;
int daysSinceLastModification;
int durationInDays;
```

ИМЕНОВАНИЕ

Короткие имена можно использовать в случае, когда время жизни переменной тоже коротко.

```
for (int i = 0; i < array.length; i++) {
    // some code here
}
```

ЧАСТЬ 3. ТИПИЧНЫЕ ОШИБКИ

ТИПИЧНЫЕ ОШИБКИ

или на что обратить внимание прямо сейчас:

- Злоупотребление модификатором static
- Процедурный стиль программирования
- Игнорирования пакетирования и Naming convention
- Привязка к конкретной реализации
- Пренебрежение идентификаторами доступа
- И спользование коллекции Vector

ОШИБКА: ЗЛОУПОТРЕБЛЕНИЕ МОДИФИКАТОРОМ STATIC

static нужен только для ограниченного круга задач:

- Для метода main

```
public static void main(String[] args){...}
```

- Для создания утилитных методов

```
java.lang.Math.abs()
```

- Поле класса может быть static, если оно используется как *общее* хранилище(in-memory cache, singleton, etc.)
- Для объявления констант(final static), но только если enum не подходит

ОШИБКА: ПРОЦЕДУРНЫЙ СТИЛЬ ПРОГРАММИРОВАНИЯ

Правильная последовательность действий:

- Дробить код программы на методы
- Если метод не вмещается на страницу экрана – это признак плохого кода
- Общие по функционалу методы выделять в новые сущности, порождая **классы**

EXTRACT METHOD(1/3)

```
public List<Customer> getCustomers(List<Form> customerForms) {
    List<Long> ids = new ArrayList<>(customerForms.size());
    for (Form form : customerForms) {
         ids.add(form.getCustomerId());
    }
    List<Customer> customers = customerDao.getCustomersByIds(ids);
    Iterator<Customer> iterator = customers.iterator();
    while (iterator.hasNext()) {
        Customer customer = iterator.next();
        if(customer.getFullName() == null) {
           iterator.remove();
        }
    }
    return customers;
}
```

EXTRACT METHOD(2/3)

```
private void removeUnnamedCustomers(List<Customer> customers) {
    Iterator<Customer> iterator = customers.iterator();
    while (iterator.hasNext()) {
        Customer customer = iterator.next();
        if(customer.getFullName() == null) {
               iterator.remove();
        }
    }
}

private List<Long> exposeCustomerIds(List<Form> customerForms) {
    List<Long> ids = new ArrayList<Long>(customerForms.size());
    for (Form form : customerForms) {
         ids.add(form.getCustomerId());
    }
    return ids;
}
```

EXTRACT METHOD(3/3)

```
public List<Customer> getCustomers(List<Form> customerForms) {
    List<Long> ids = exposeCustomerIds(customerForms);
    List<Customer> customers = customerDao.getCustomersByIds(ids);
    removeUnnamedCustomers(customers);

    return customers;
}
```

ОШИБКА: ИГНОРИРОВАНИЕ ПАКЕТИРОВАНИЯ И NAMING CONVENTION

- Пакеты именуются с помощью иерархического шаблона имен

```
package by.bsu.myprogram
```

- Классы именуются в стиле UpperCamelCase

```
class MyClass
```

- Методы и переменные в стиле lowerCamelCase

```
private void myMethod()
```

- Константы пишутся в верхнем регистре

```
public static final int DEFAULT_USER_ID = 1;
```

ЕЩЕ РАЗ ПРО ИМЕНОВАНИЕ ПЕРЕМЕННЫХ.

- Именование классов, методов, а главное переменных обычно занимает больше времени, чем написание самого кода. Именование очень важно для понимания чужого кода, т.к. читать код сложнее, чем писать.
- Переменная, метод или класс должны иметь понятное название, по которому можно судить о их назначении
- Умеренная длина
- Спецификаторы – count и index вместо num

ОШИБКА: ПРИВЯЗКА К РЕАЛИЗАЦИИ

```
//wrong
private ArrayList<String> wrong = new ArrayList<>();

public void printList(ArrayList<String> arrayList){
        ...
}

//correct
List<String> correct = new ArrayList<>();

public void printList(List<String> list){
        ...
}
```

ОШИБКА: ПРЕНЕБРЕЖЕНИЕ ИДЕНТИФИКАТОРАМИ ДОСТУПА

- Нельзя забывать, что спецификаторы доступа помогают реализовать один из принципов ООП – инкапсуляцию
- Поле класса должно быть приватным. Если нужно обеспечить его изменяемость или доступ к нему – используйте спец. методы
- Если вы делаете метод доступным (не private), то тем самым вы гарантируете, что будете поддерживать его. Если кто-то начнёт его использовать, то удалить его уже так просто не получится
- Используем такое правило:
  1. **public** - для того, что должно быть доступно всем(конструкторы, интерфейсные методы). Не подходят для полей.
  2. **protected** - для того, что должно быть доступно наследникам
  3. **package** - для того, что должно быть доступно только в пакете. Иногда требуется для реализации библиотек или компонентов
  4. **private** - для всего остального

СПЕЦИФИКАТОРЫ ДОСТУПА

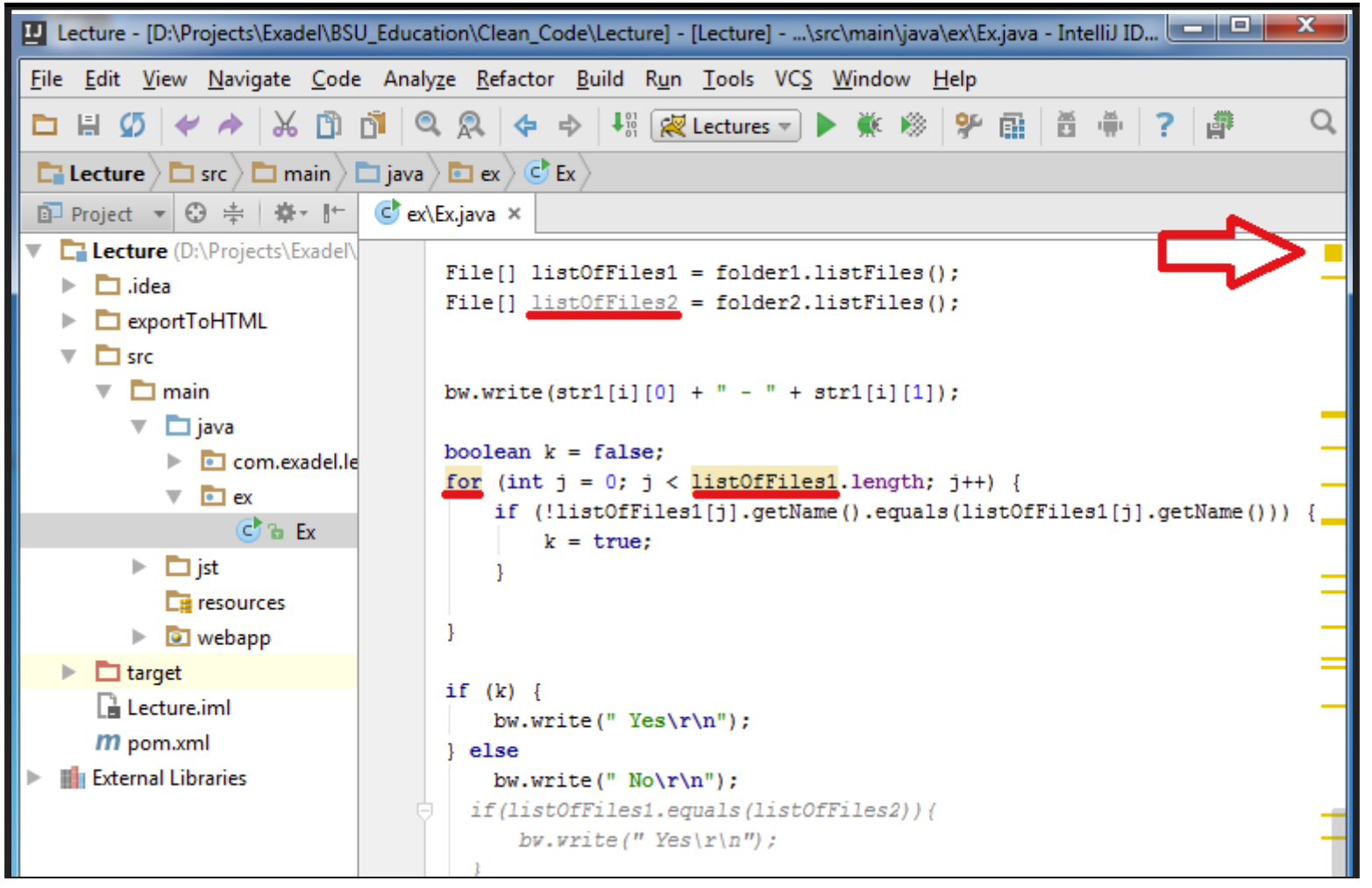
```
public class Student {
    // wrong
    public int id;

    // correct
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

ОШИБКА: ИСПОЛЬЗОВАНИЕ КОЛЛЕКЦИИ VECTOR

As of the Java 2 platform v1.2, this class was retrofitted to implement the List interface, making it a member of the Java Collections Framework. Unlike the new collection implementations, Vector is synchronized. If a thread-safe implementation is not needed, it is recommended to use ArrayList in place of Vector.

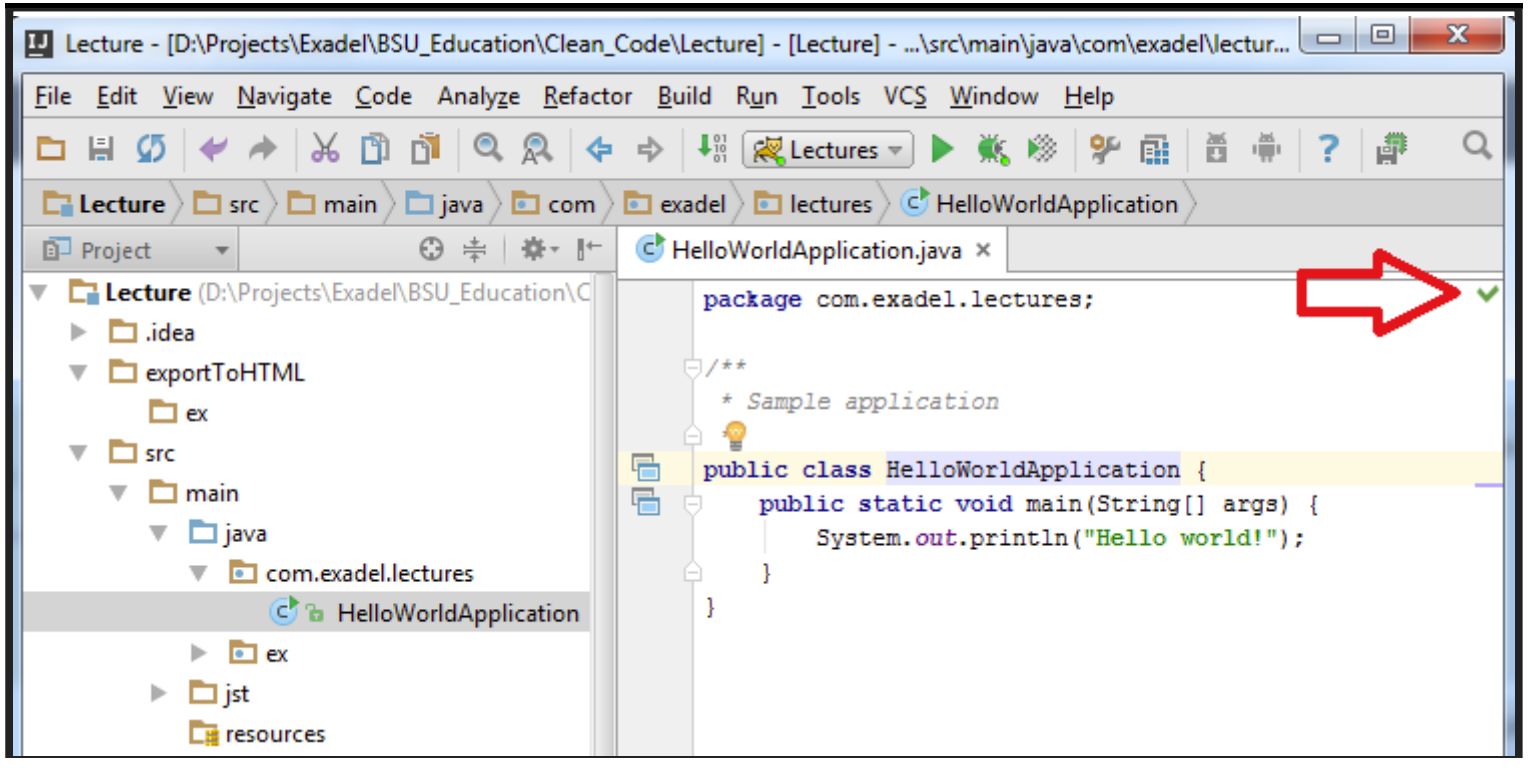
ЧАСТЬ 4. ИНСТРУМЕНТЫ

АНАЛИЗ КОДА В IDEA

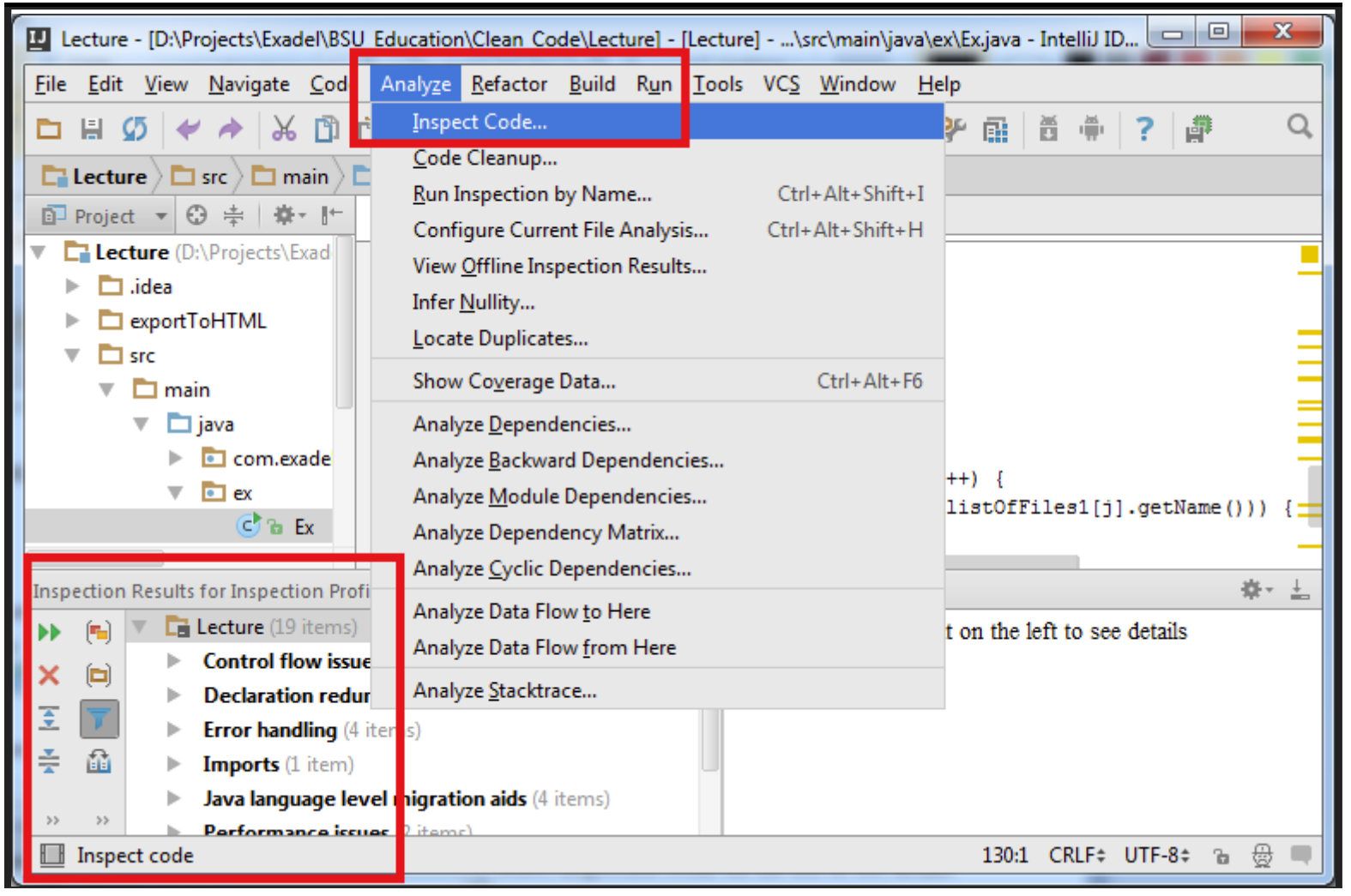


АНАЛИЗ КОДА В IDEA

Как должно быть



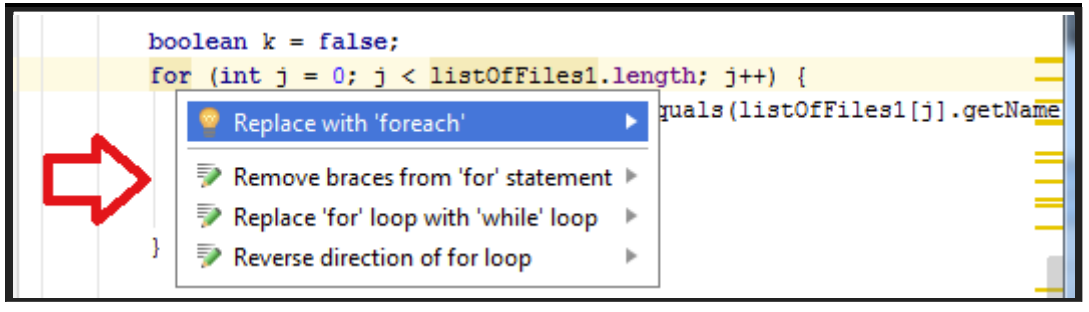
АНАЛИЗ ВСЕГО КОДА В IDEA



Exadel

ПОДСКАЗКИ В IDEA

- F2 / Shift + F2 - следующая/предыдущая ошибка
- Alt + Enter - быстрое исправление(Quick Fix)



CODE FORMATTING IN IDE

Горячие клавиши

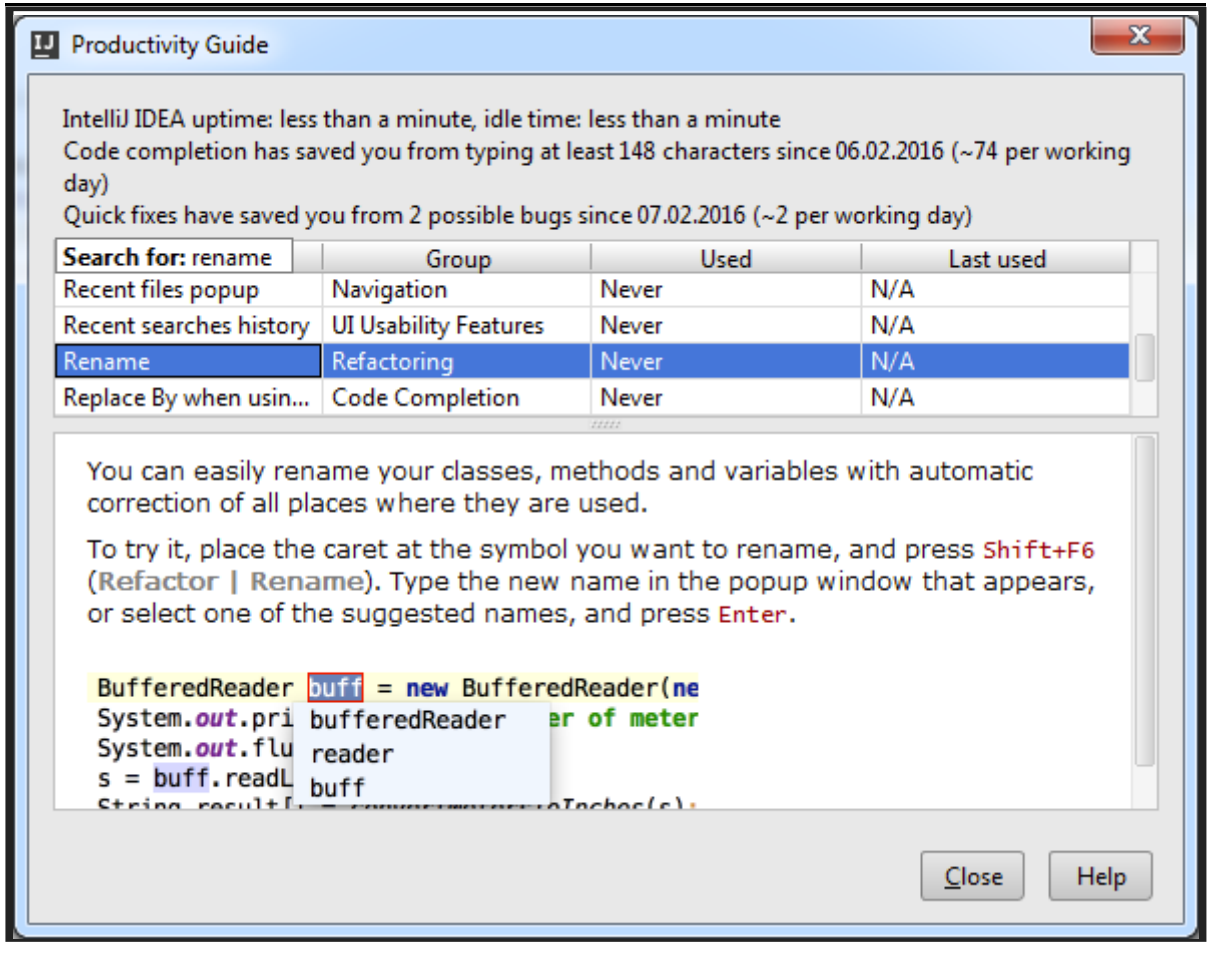
- **Eclipse**: ctrl + shift + F
- **IDEA**: ctrl + alt + L
- **Netbeans**: alt + shift + F

INTRODUCE VAR, METHOD, FIELD, CONSTANT

- **Eclipse**: alt + shift + {v, m, e, c}
- **IDEA**:ctrl + alt + {v, m, f, c}
  см. полный список возможностей в меню **Refactor**(доступно и через контекстное меню).
- **Netbeans**: alt + shift + {v, m, e, c}

IDEA PRODUCTIVITY GUIDE

См. меню Help > Productivity Guide



Exadel

IDEA: ПОЛНЫЙ СПИСОК ВОЗМОЖНОСТЕЙ

- Меню Help > Default Keymap Reference - горячие клавиши.

  Этот же документ доступен on-line

- https://www.jetbrains.com/idea/features/

ЧТО НУЖНО ИЗУЧИТЬ

Обязательно

- Code convension: https://google.github.io/styleguide/javaguide.html
- Фаулер М., Бек К., Брант Д. и др. Рефакторинг: улучшение существующего кода

Дополнительно

- Стив Макконнелл. Совершенный Код
- William J. Brown etc. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis

ВОПРОСЫ?