

# TESTABILITY, АВТОТЕСТЫ, ВВЕДЕНИЕ В ООП: ОБЪЕКТЫ И МЕТОДЫ



ОКСАНА МЕЛЬНИКОВА



**ОКСАНА МЕЛЬНИКОВА**

Software testing engineer





# ПЛАН ЗАНЯТИЯ

1. [Автотесты](#)
2. [Функции](#)
3. [Введение в ООП](#)
4. [Классы и объекты](#)
5. [Итоги](#)



# АВТОТЕСТЫ

# АВТОТЕСТЫ

На прошлой лекции мы написали с вами небольшую утилиту (вспомогательное приложение) для упрощения проверок.

```
public class Main {  
    public static void main(String[] args) {  
        boolean registered = true;  
        int percent = registered ? 3 : 1;  
        long amount = 1000_60;  
        long bonus = amount * percent / 100 / 100;  
        long limit = 500;  
        if (bonus > limit) {  
            bonus = limit;  
        }  
        System.out.println(bonus);  
    }  
}
```



# АВТОТЕСТЫ

Само это приложение мы проверяли руками, меняя входные данные и анализируя результат в консоли.

Это очень плохой подход, поскольку с ростом сложности приложения мы будем вынуждены либо перепроверять всё заново, либо просто «надеяться, что ничего не сломалось».

Оба варианта — не очень.



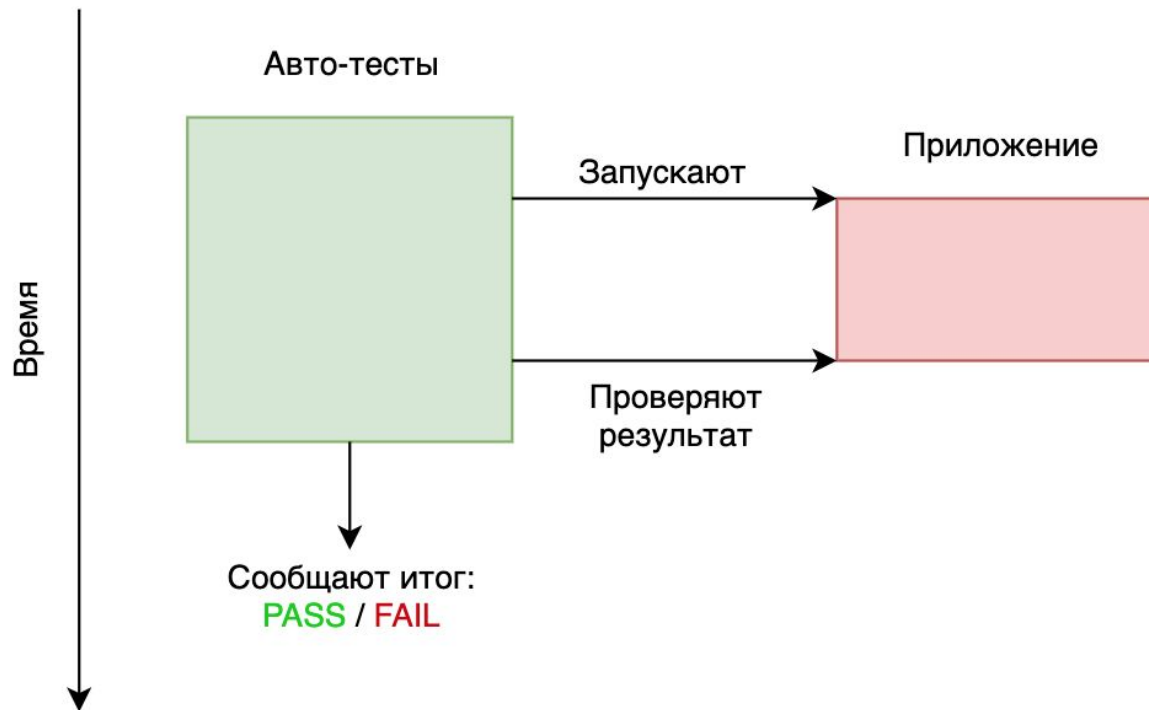
# АВТОТЕСТЫ

Вместо этого, мы сделаем автотесты, которые:

- запускаются одной командой;
- проверяют функциональность приложения без нашего участия;
- сообщают нам о результате проверок (**PASS**/**FAIL**).

# АВТОТЕСТЫ

Фактически, мы просто по определённым правилам напишем код на Java (автотесты), который будет вызывать другой код на Java (наше приложение) с определёнными параметрами и проверять возвращаемый результат.







# АВТОТЕСТЫ

**Q:** т.е. мы будем писать автотесты на автотесты?

**A:** нет, мы будем писать автотесты на утилиту, помогающую тестировщикам.

Писать автотесты на автотесты не нужно. Это то же самое, что писать тест-кейсы на тест-кейсы.

# ТЕСТИРОВАНИЕ

В самом упрощённом сценарии тестирование нашей утилиты сводится к следующему:

1. Запустить наше приложение с нужными входными данными (registered и amount)
2. Получить фактический результат
3. Сравнить фактический результат с ожидаемым (совпадает — PASS, не совпадает — FAIL)



# ИНСТРУМЕНТЫ

У нас есть два варианта решения:

1. Написать подобную утилиту автотестирования самим (долго, дорого, чревато ошибками).
2. Использовать готовые инструменты (необходимо изучить, настроить и подготовить приложение).

Мы выберем второй путь, но перед этим обсудим понятие **Testability**.



# TESTABILITY

**Testability** — степень, с которой система пригодна для тестирования (определение нечёткое и не определяет численной характеристики).

Наше приложение обладает низкой Testability (нужно руками менять значения в коде и запускать).

**Q:** как же повысить Testability нашего приложения?

**A:** посмотрим на ваш предыдущий опыт в ручном тестировании.



# ПАРАМЕТРЫ И РЕЗУЛЬТАТ

Для того, чтобы переиспользовать тест-кейсы, можно параметры вынести во входные данные, при этом саму логику оставить неизменной.

В итоге мы просто «прогоняем» один и тот же тест-кейс с разными данными.

**Ключевой вопрос:** как это сделать?



# ФУНКЦИИ

# ФУНКЦИИ

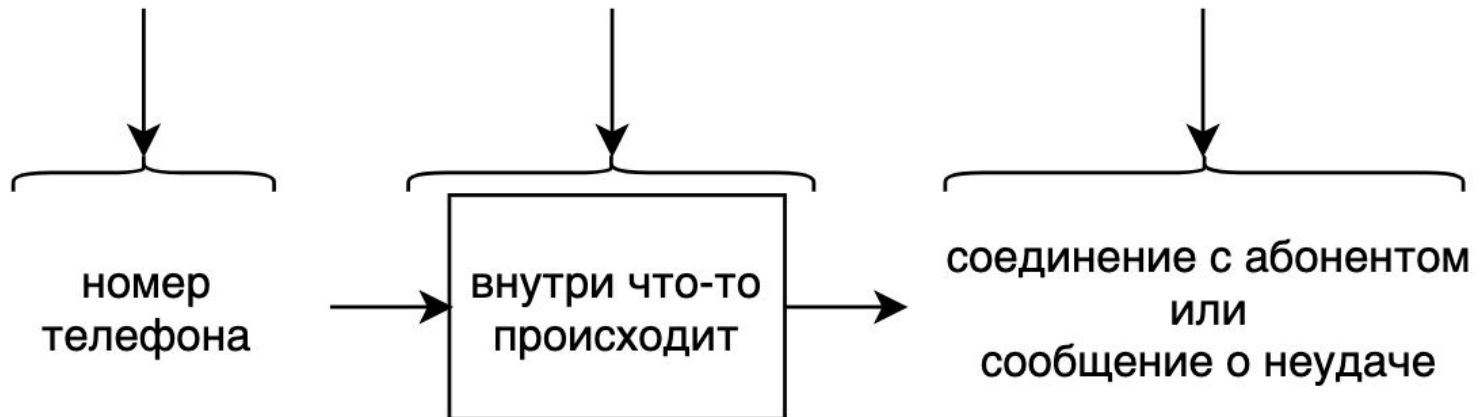
В окружающем мире есть такое понятие «функция», например, «Позвонить» в смартфоне:

- мы выбираем кому позвонить (или вводим номер);
- вызываем эту функцию (нажатием на иконку звонка);
- и через какое-то время получаем результат: удалось соединиться с абонентом или нет.

входные параметры

функция

результат





# ФУНКЦИИ

Функция — это «кусочек» кода, которому дали имя, например «Позвонить».

Мы (или устройство) может запускать эту функцию в ответ на действия пользователя.

Функции могут запускаться и на другие события: например, функция «Проиграть мелодию будильника» запускается при наступлении заданного времени.



# ФУНКЦИИ

Мы могли бы из нашего кода сделать функцию «Рассчитать бонус», которая на вход принимает два параметра:

1. Зарегистрирован пользователь или нет
2. Сумму покупок

А возвращать функция будет сумму бонуса.

Тогда так же, как с функцией «Позвонить» мы сможем «запускать» эту функцию с различной комбинацией параметров\* и проверять результат без необходимости менять саму функцию? Ответ узнаем далее.

Примечание\*: а комбинации мы будем подбирать исходя из знаний комбинаторики в тестировании.



# **ВВЕДЕНИЕ В ООП**



# ОБЪЕКТЫ

В Java функции не могут существовать сами по себе.

В реальном мире вы тоже не можете просто «позвонить»: вам нужен телефон, который умеет «звонить».

Т.е. должен существовать какой-то **объект**, который **обладает нужной функцией** и мы можем у этого **конкретного объекта** «вызвать» эту **функцию** (запустить этот кусочек кода).



# ООП

ООП (объектно-ориентированное программирование) — это подход к моделированию реального мира в программировании, когда мы всё **описываем в виде объектов**, обладающих **свойствами** и **определёнными функциями**.



# СВОЙСТВА И МЕТОДЫ

У объектов есть свойства и функции (в Java они называются методы):

- **свойства** — это уровень заряда, баланс, наличие сети и т.д.;
- **методы** — это «Позвонить», «Заблокировать», «Проиграть музыку».



## ПРИМЕР ИЗ ЖИЗНИ

Представьте, что мы программируем систему, управляющую лифтом.

Система должна уметь:

1. Давать возможность вызывать лифт на определённый этаж — это метод.
2. Давать возможность перемещать лифт на определённый этаж (когда вы внутри лифта) — это тоже метод.



## ПРИМЕР ИЗ ЖИЗНИ

При этом у системы есть свойства:

1. Исправность — исправна или нет.
2. Максимально допустимая для перевозки масса.
3. Перевозимая масса (масса тех, кто находится в лифте).
4. Этаж (на котором находится лифт).
5. Статус лифта — свободен лифт в данный момент или перемещается.

# СОСТОЯНИЕ

Текущее значение всех свойств объекта называется **состоянием**.

В зависимости от состояния может меняться поведение объекта:

- если система неисправна, то методы не перемещают лифт;
- если перевозимая масса выше, чем допустимая, — то лифт остаётся с открытыми дверями и никуда не едет;
- если лифт едет вниз, то он не реагирует на вызовы верхних этажей;
- и т.д.





# СОСТОЯНИЕ

Фактически, мы с вами сделали всё то же самое, что при проектировании тестов:

- выполнили анализ ключевых возможностей (что система умеет делать — методы)
- определили ключевые характеристики (какими свойствами/параметрами система обладает — свойства)
- определили возможные состояния системы (конкретные значения свойств — состояние)

Но только в тестах мы это использовали для генерации тест-кейсов, а в программировании используем для создания самой системы.



# ОБЪЕКТЫ

**Q:** т.е. мы описываем всё, что окружает нас в виде объектов? Но нас же окружает миллион вещей, у которых сотни свойств и функций!

**A:** нет, мы описываем только то, что важно для решения конкретной задачи.

Возвращаясь к нашей задаче с бонусом: нам **нужен объект**, у которого будет **всего один метод** (функция) — рассчитать бонус.



# ОБЪЕКТЫ БЕЗ СОСТОЯНИЯ

Обратите внимание: мы хотим, чтобы был объект, который умеет только вычислять бонус.

Нам важна только эта его функция — больше ничего мы о нём знать (в рамках решения задачи) не хотим.

При этом мы не можем выделить у него никаких свойств (да это нам и не нужно) — поэтому у него нет состояния.



## ОБЪЕКТЫ БЕЗ СОСТОЯНИЯ

Такие объекты — подарок для тестирования: раз нет состояния, то поведение (работа методов) будет зависеть только от входных параметров.

А значит, **количество тестовых комбинаций сокращается в разы** (нет скрытых параметров вроде значений свойств).



# КЛАССЫ И ОБЪЕКТЫ



# КЛАССЫ И ОБЪЕКТЫ

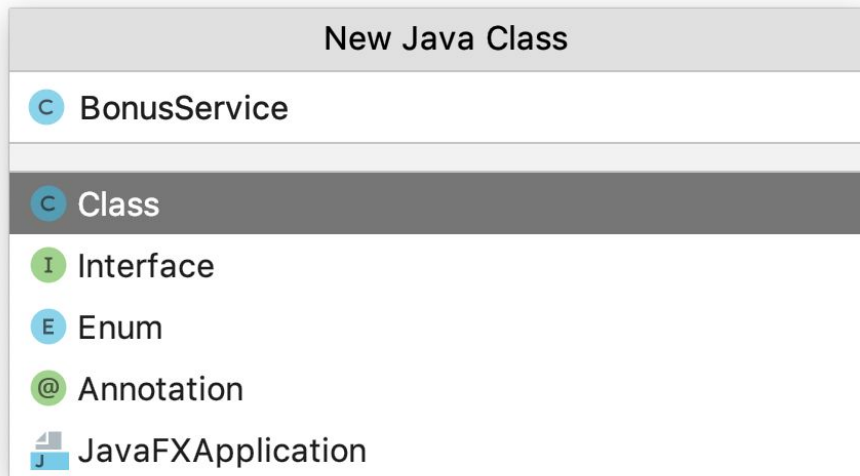
Для того, чтобы получить объект, с которым можно работать, нам нужно сделать два шага:

1. Описать этот объект (какие свойства и методы у него будут).
2. Создать объект из этого описания.

# КЛАСС

Описание объекта мы будем называть классом.

Класс создаётся так же, как `Main` название вводите как `BonusService`:



Все классы должны называться с большой буквы: `BonusService`, а не `bonusService`.

# СЕРВИСЫ

В нашем курсе мы будем называть классы для объектов, не имеющих состояния\* и содержащих только бизнес-логику, с суффиксом `Service`.

Т.е. по названию `BonusService` будет понятно, что это класс, описывающий объекты без состояния, которые содержат бизнес-логику.

Примечание\*: в следующих лекциях внесём некоторые коррективы в это правило (скажем, что и у сервисов может быть состояние).



# МЕТОДЫ

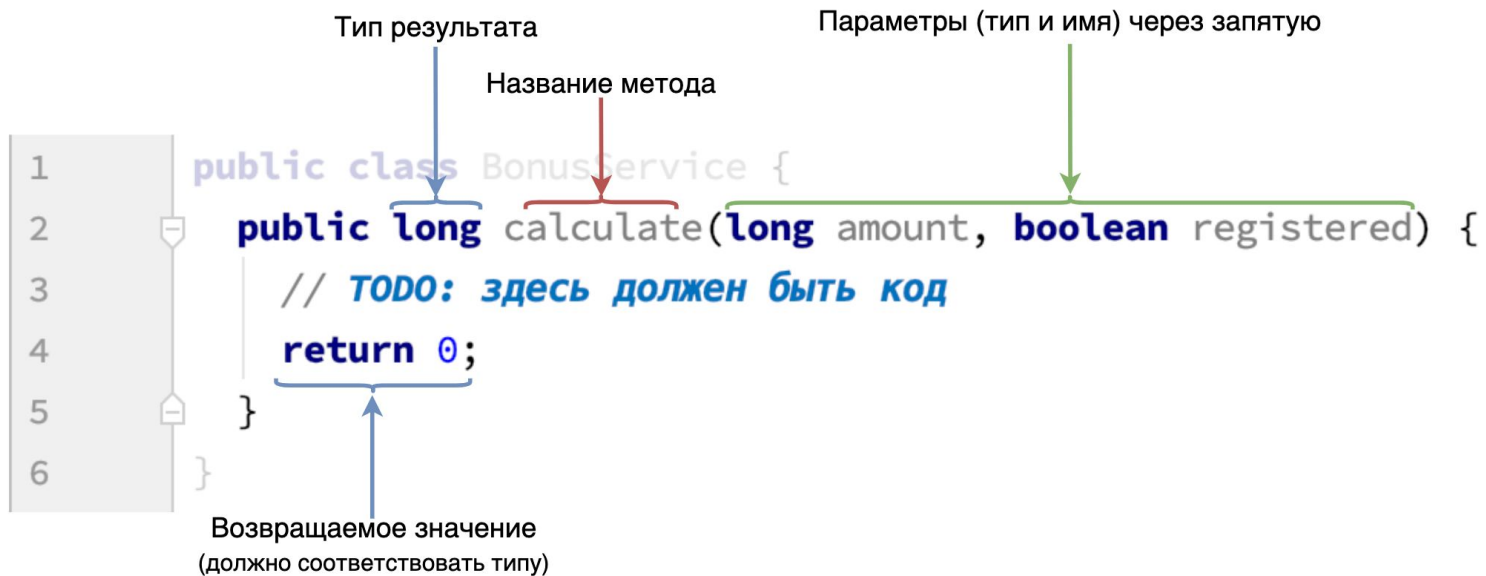
Методы — это функции\*, которые будут у созданного объекта.

У метода есть:

1. **Имя** («Позвонить» и т.д. — но на английском).
2. **Входные параметры** (amount типа long, registered типа boolean).
3. **Тип возвращаемого результата** (в нашем случае — long).

Примечание\*: далее мы везде будем говорить методы.

# МЕТОДЫ



Как это читать:

- метод `calculate`;
- возвращает значение типа `long`;
- принимает параметры: `amount` типа `long` и `registered` типа `boolean`.

# МЕТОДЫ

Пока наш метод всегда возвращает 0 (инструкция `return 0;`).

Разберёмся, как его вызывать:

```
public class Main {  
    public static void main(String[] args) {  
        // создаём объект из класса  
        BonusService service = new BonusService();  
        // вызываем метод и результат присваиваем переменной bonus  
        long bonus = service.calculate(1000_60, true);  
        // печатаем значение переменной bonus  
        System.out.println(bonus);  
    }  
}
```

# СОЗДАНИЕ ОБЪЕКТА

Пойдём по строчкам:

```
// создаём объект из класса  
BonusService service = new BonusService();
```

↑                    ↑                    ↑  
тип переменной    имя переменной    значение

```
boolean registered = true; // аналог для примитивов
```

↓                    ↓                    ↓

1. У примитивов типы фиксированы, а у объектов тип — это класс, из которого создали объект.
2. У примитивов инициализация через литерал (значение в коде), у объектов через `new ИмяКласса()`.

# СОЗДАНИЕ ОБЪЕКТА

Q: что такое `new BonusService()`?

A: пока нам нужно запомнить, что это создание объекта из класса (описания объекта).

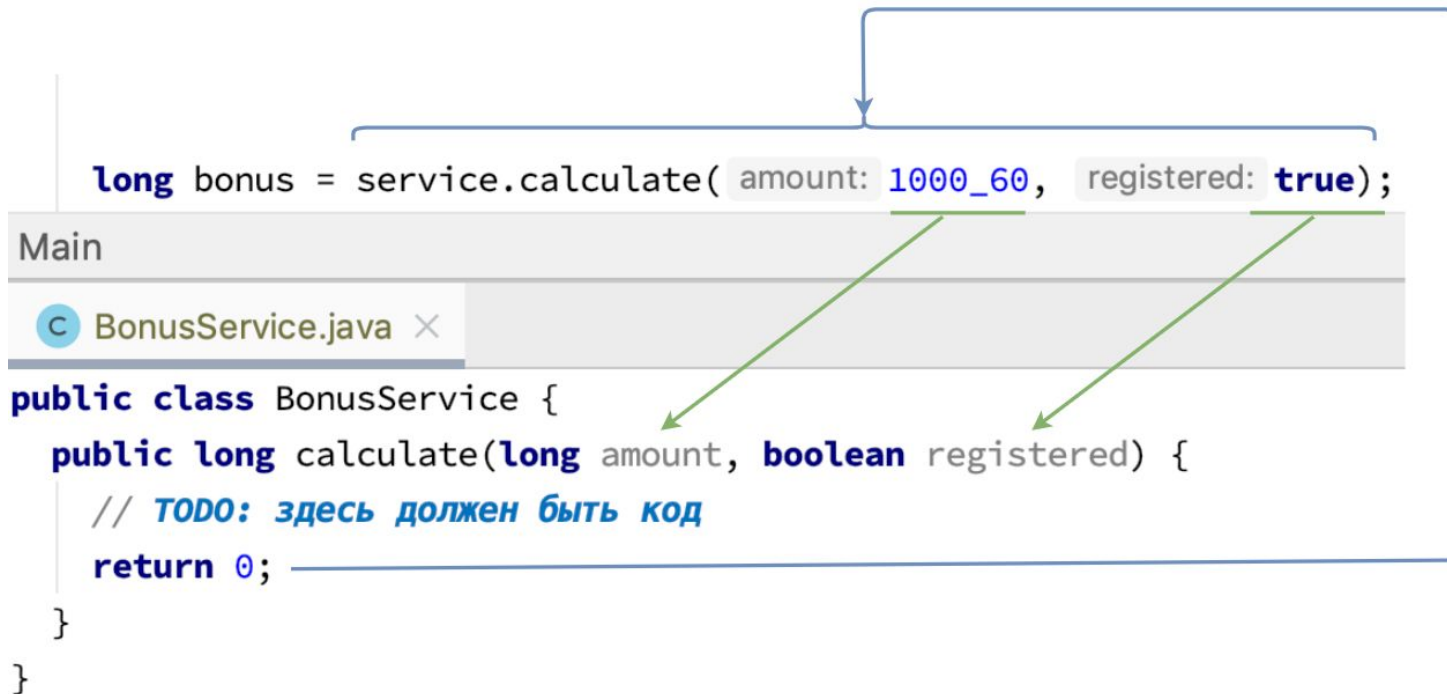
Представьте, что вы запускаете Калькулятор из панели Пуск. Каждый раз, когда вы нажимаете на пункт Калькулятор, создаётся **новое окно** Калькулятора.

Каждое новое окно — это отдельный объект.

Тоже самое происходит и у нас: `new BonusService()` — это создание объекта (только не калькулятора, а нашего, который описан в классе `BonusService`).

# ВЫЗОВ МЕТОДА

Вместо всего выражения подставится  
то, что в return



**Важно:** `amount:` и `registered:` — это подсказки IDEA (писать их текстом не нужно!).

Нужно писать только значения `1000_60` и `true`.

# ВЫЗОВ МЕТОДА

При вызове метода происходит следующее: Java «перескакивает» в наш метод и выполняет код (так это можно себе представлять):

```
{  
    long amount = 1000_60; // подставлено из service.calculate(1000_60, true)  
    boolean registered = true; // подставлено из service.calculate(1000_60, true)  
    // TODO: здесь должен быть код  
    return 0;  
}
```

А именно:

1. Были созданы локальные переменные.
2. В эти локальные переменные были помещены те значения, которые мы указали в скобках при вызове метода.
3. Был выполнен оставшийся код.

# ВЫЗОВ МЕТОДА

Выполнив код, Java «перескакивает» обратно и вместо вызова метода подставляет то, что было в `return`:

```
public class Main {  
    public static void main(String[] args) {  
        ...  
        long bonus = 0; // т.к. в service.calculate было return 0;  
        // печатаем значение переменной bonus  
        System.out.println(bonus);  
    }  
}
```

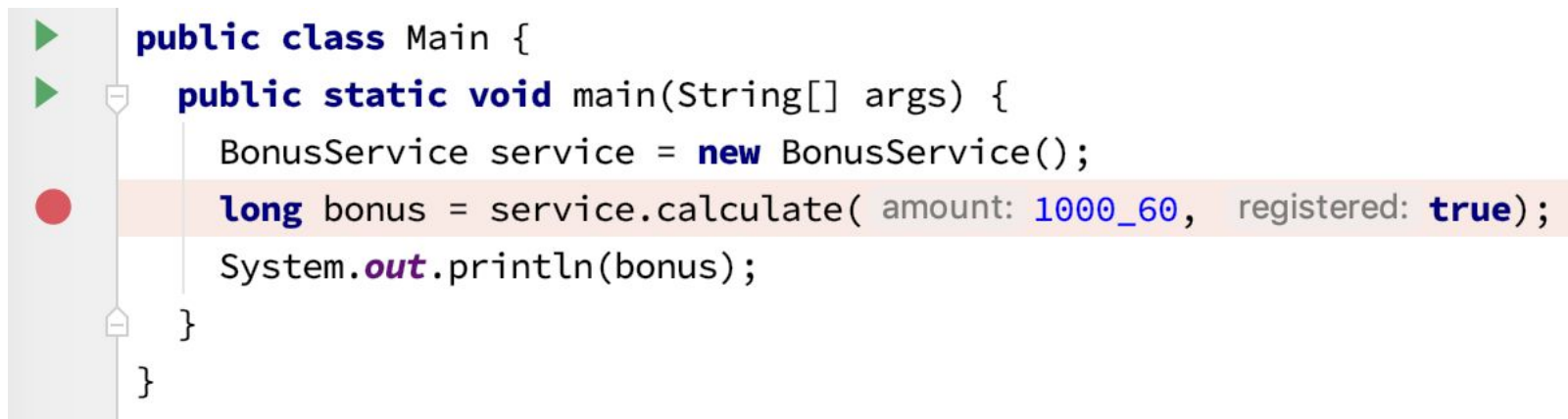


# ОТЛАДКА

Посмотрим работу под отладчиком, чтобы понять, как всё выглядит на самом деле.

Напомним, что отладчик нам нужен для того, чтобы «пройти» нашу программу по «шагам».

1. Создадим конфигурацию запуска (Ctrl + Shift + F10).
2. Поставить точку остановки (Ctrl + F8) там, где вызывается метод.



```
public class Main {  
    public static void main(String[] args) {  
        BonusService service = new BonusService();  
        long bonus = service.calculate( amount: 1000_60, registered: true);  
        System.out.println(bonus);  
    }  
}
```

# ОТЛАДКА (DEBUGGING)

Для того, чтобы попасть в метод (увидеть, как Java «перепрыгивает туда»), нужно нажать **F7** (Step Into):

```
public class BonusService {  
    public long calculate(  
        long amount,  amount: 100060  
        boolean registered  registered: true  
    ) {  
        // TODO: здесь должен быть код  
        return 0;  
    }  
}
```

IDEA покажет, какие значения присвоились **amount** и **registered** (для удобства мы разнесли их на разные строки).

**F8** (Step Over) просто перешагнёт вызов (т.е. не будет вас туда перебрасывать, а сразу подставит результат из **return**).

# ОТЛАДКА (DEBUGGING)

При следующих нажатиях F8 мы после return снова «перепрыгнем» в класс Main:

```
public class Main {  
    public static void main(String[] args) { args: {}  
        BonusService service = new BonusService(); service: BonusService@791  
        long bonus = service.calculate( amount: 1000_60, registered: true); bonus: 0  
        System.out.println(bonus); bonus: 0  
    }  
}
```



# ОТЛАДКА (DEBUGGING)

Демонстрация отладки в IntelliJ IDEA.

# ДОБАВЛЯЕМ КОД

Мы разобрались, как Java обрабатывает вызов метода, давайте доработаем его (чтобы он содержал нужную логику):

```
public class BonusService {  
    public long calculate(long amount, boolean registered) {  
        int percent = registered ? 3 : 1;  
        long bonus = amount * percent / 100 / 100;  
        long limit = 500;  
        if (bonus > limit) {  
            bonus = limit;  
        }  
        return bonus; // возвращаем рассчитанный bonus  
    }  
}
```

## ВАЖНЫЕ ДЕТАЛИ

Обратите внимание: мы убрали строки (теперь значения передаются при вызове):

```
boolean registered = true;  
long amount = 1000_60;
```

И вместо `return 0` написали `return bonus`, т.к. хотим «вернуть» значение, которое положили в эту переменную.



# ОТЛАДКА (DEBUGGING)

Повторная демонстрация отладки в IntelliJ IDEA (с обновлённым кодом).

# ЧТО НАМ ЭТО ДАЛО?

Теперь мы можем вызывать метод много раз с разными параметрами:

```
public class Main {  
    public static void main(String[] args) {  
        BonusService service = new BonusService();  
  
        long bonusBelowLimitForRegistered = service.calculate(1000_60, true);  
        System.out.println(bonusBelowLimitForRegistered);  
  
        long bonusOverLimitForRegistered = service.calculate(1_000_000_60, true);  
        System.out.println(bonusOverLimitForRegistered);  
  
        long bonusBelowLimitForUnRegistered = service.calculate(1000_60, false);  
        System.out.println(bonusBelowLimitForUnRegistered);  
  
        long bonusOverLimitForUnRegistered = service.calculate(1_000_000_60, true);  
        System.out.println(bonusOverLimitForUnRegistered);  
    }  
}
```





## ЧТО НАМ ЭТО ДАЛО?

Осталось только сделать так, чтобы не мы проверяли "глазами" фактический результат и в уме сравнивали его с ожидаемым.

Об этом мы и поговорим на следующей лекции.



# ИТОГИ



# ИТОГИ

Сегодня мы затронули тему объектов, классов и методов.

Это очень важная тема: поскольку в Java большая часть построена именно на этом фундаменте.

Не стоит расстраиваться, если не сразу всё понятно: мы будем практиковаться в написании классов и методов весь курс.

# HOTKEYS

Ключевые клавиатурные сокращения\*:

1. **Ctrl + F8** — установка/снятие точки остановки.
2. **Shift + F9** — запуск под отладчиком.
3. **F8** — исполнение следующей строки (без захода в метод) в режиме отладки.
4. **F7** — исполнение следующей строки (с заходом в метод) в режиме отладки.

На Mac OS посмотреть клавиатурные сокращения можно через пункт меню Help -> Keymap Reference.



# ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задаем в чате Slack!
- Задачи можно сдавать по частям.
- Зачет по домашней работе проставляется после того, как приняты **все задачи**.



**Задавайте вопросы и напишите отзыв о лекции!**

**ОКСАНА МЕЛЬНИКОВА**

 Оксана Мельникова