



Lesson 12



26.04.2021

```
public class Ex1 {  
    public static void main(String[] args) {  
        Long a = null;  
        long b = 0;  
  
        if (a == b) {  
            System.out.println("==");  
        } else {  
            System.out.println("!=");  
        }  
    }  
}
```

```
public class Ex2 {  
    public static void main(String[] args) {  
        System.out.println(null);  
    }  
}
```

```
public class Ex3 extends Ex3_1 {  
    public void print(int d) {  
        System.out.println("EX3");  
    }  
  
    public static void main(String[] args) {  
        Ex3 ex = new Ex3();  
        ex.print(1);  
        ex.print(2.0);  
    }  
  
    class Ex3_1 {  
        public void print(double d) {  
            System.out.println("EX3_1");  
        }  
    }  
}
```

```
public class Ex4 {  
    private static StringBuilder work(StringBuilder a, StringBuilder b) {  
        a = new StringBuilder("a");  
        b.append("b");  
        return a;  
    }  
  
    public static void main(String[] args) {  
        StringBuilder sb1 = new StringBuilder("S1");  
        StringBuilder sb2 = new StringBuilder("S2");  
        StringBuilder sb3 = work(sb1, sb2);  
  
        System.out.println("sb1 -> " + sb1);  
        System.out.println("sb2 -> " + sb2);  
        System.out.println("sb3 -> " + sb3);  
    }  
}
```

```
public class Ex5 {  
    public static void main(String[] args) {  
        TreeSet<String> set = new TreeSet<>();  
        set.add("Java");  
        set.add("The");  
        set.add("Java");  
        set.add("JavaTheBest");  
  
        for (String str : set) {  
            System.out.print(str + " ");  
        }  
        System.out.println("\n");  
    }  
}
```



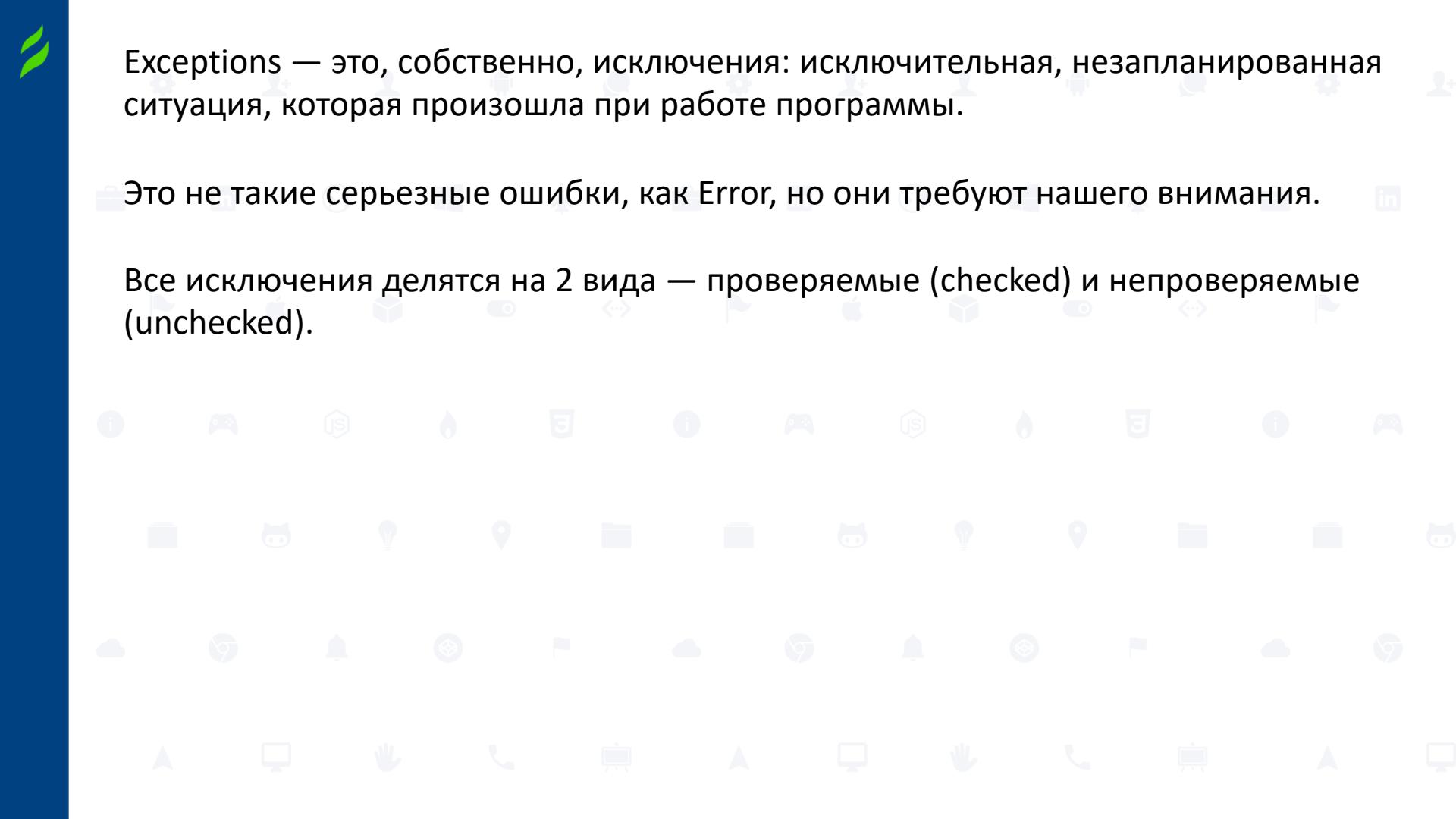
Исключения в Java

В java исключением называется любая ошибка, которая возникает в ходе выполнения программы. Это может быть несоответствие типов данных, деление на ноль, обрыв связи с сервером и многое другое. Операции по их поиску и предотвращению называются обработкой исключений.

У всех исключений есть общий класс-предок ***Throwable***.

От него происходят две большие группы — исключения (***Exception***) и ошибки (***Error***).

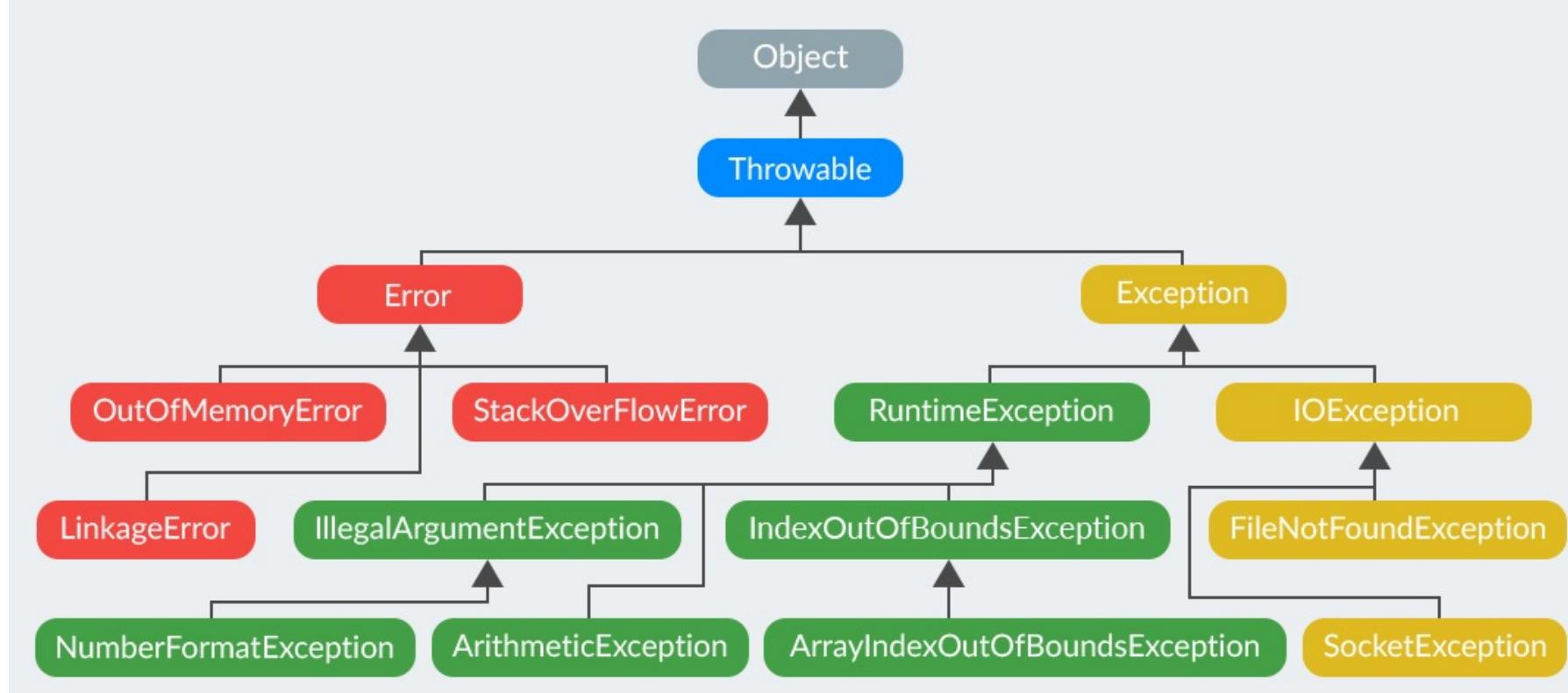
Error — это критическая ошибка во время исполнения программы, связанная с работой виртуальной машины Java. В большинстве случаев Error не нужно обрабатывать, поскольку она свидетельствует о каких-то серьезных недоработках в коде.



Exceptions — это, собственно, исключения: исключительная, незапланированная ситуация, которая произошла при работе программы.

Это не такие серьезные ошибки, как Error, но они требуют нашего внимания.

Все исключения делятся на 2 вида — проверяемые (checked) и непроверяемые (unchecked).





Ключевые слова:

try – определяет блок кода, в котором может произойти исключение;

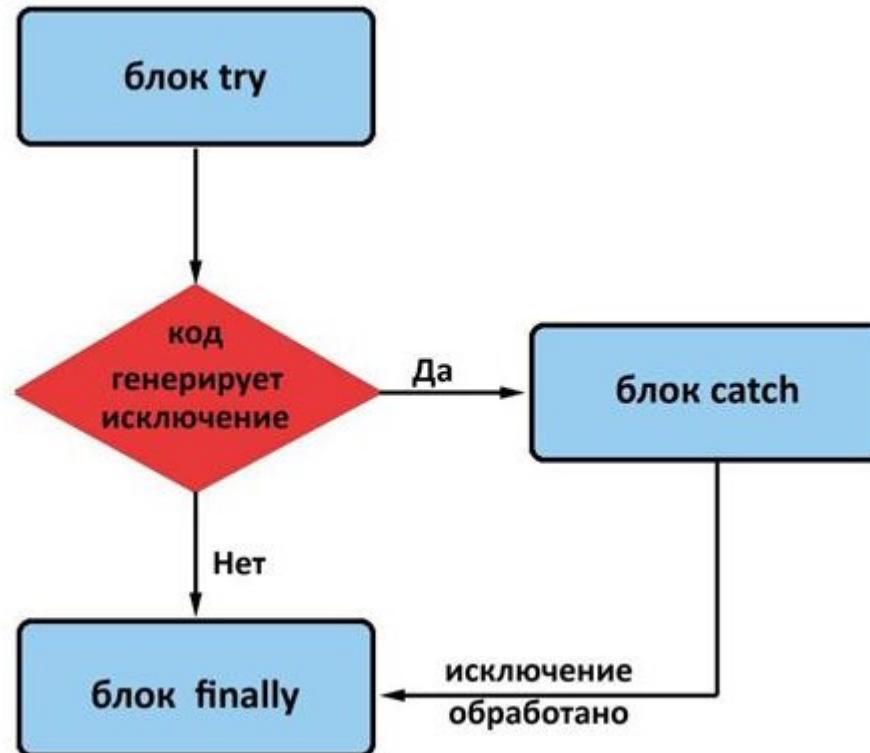
catch – определяет блок кода, в котором происходит обработка исключения;

finally – определяет блок кода, который является необязательным, но при его наличии выполняется в любом случае независимо от результатов выполнения блока try.

throw – используется для возбуждения исключения;

throws – используется в сигнатуре методов для предупреждения, о том что метод может выбросить исключение.

Обработка исключения



Java 10 - var

В качестве разработчика Java мы привыкли дважды вводить типы, один раз для объявления переменной и второй раз для следующего за ней конструктора:

```
List<String> list = new ArrayList<>();
```

Начиная с Java 10 у разработчиков появится альтернатива — они могут позволить компилятору вывести тип с помощью *var*:

```
var listVar = new ArrayList<String>();
```

При обработке *var*, компилятор просматривает правую часть объявления, так называемый инициализатор и использует его тип для переменной. И это нужно не только для внутренних расчётов, полученный тип будет также записан в итоговый байт-код.

try + catch

По первому пункту: catch — полиморфная конструкция, т.е. catch по типу Parent перехватывает экземпляры любого типа исключения, который является родителем (т.е. экземпляры непосредственно Parent-а или любого потомка Parent-а)

```
public class TryCatch {  
    public static void main(String[] args) {  
        try {  
            System.err.print(" 0");  
            if (true) {  
                throw new RuntimeException();  
            }  
            System.err.print(" 1");  
        } catch (RuntimeException e) {  
            System.err.print(" 2");  
        }  
        System.err.println(" 3");  
    }  
}
```

try + catch + catch + ...

Мы можем расположить несколько catch после одного try. Но есть такое правило — нельзя ставить потомка после редка! (RuntimeException после Exception)

```
public class TryCatchCatch {  
    public static void main(String[] args) {  
        try {  
            throw new Exception();  
        } catch (RuntimeException e) {  
            System.err.println("catch RuntimeException");  
        } catch (Exception e) {  
            System.err.println("catch Exception");  
        } catch (Throwable e) {  
            System.err.println("catch Throwable");  
        }  
        System.err.println("next statement");  
    }  
}
```

try + finally

```
public class TryFinally {  
    public static void main(String[] args) {  
        try {  
            throw new RuntimeException();  
        } finally {  
            System.err.println("finally");  
        }  
    }  
}
```

try + catch + finally

```
public class TryCatchFinally {  
    public static void main(String[] args) {  
        try {  
            System.err.print(" 0");  
            if (true) {throw new Error();}  
            System.err.print(" 1");  
        } catch(Error e) {  
            System.err.print(" 2");  
        } finally {  
            System.err.print(" 3");  
        }  
        System.err.print(" 4");  
    }  
}
```

try-with-resources

```
public static void main(String[] args) {
    Scanner scanner = null;
    try {
        scanner = new Scanner(new File( pathname: "test.txt"));
        while (scanner.hasNext()) {
            System.out.println(scanner.nextLine());
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } finally {
        if (scanner != null) {
            scanner.close();
        }
    }
}
```

```
public static void main(String[] args) {
    try (Scanner scanner = new Scanner(new File( pathname: "test.txt")) ) {
        while (scanner.hasNext()) {
            System.out.println(scanner.nextLine());
        }
    } catch (FileNotFoundException fnfe) {
        fnfe.printStackTrace();
    }
}
```



Необходимо понимать, что

- проверка на **checked** исключения происходит в момент компиляции (**compile-time checking**)
- перехват исключений (**catch**) происходит в момент выполнения (**runtime checking**)



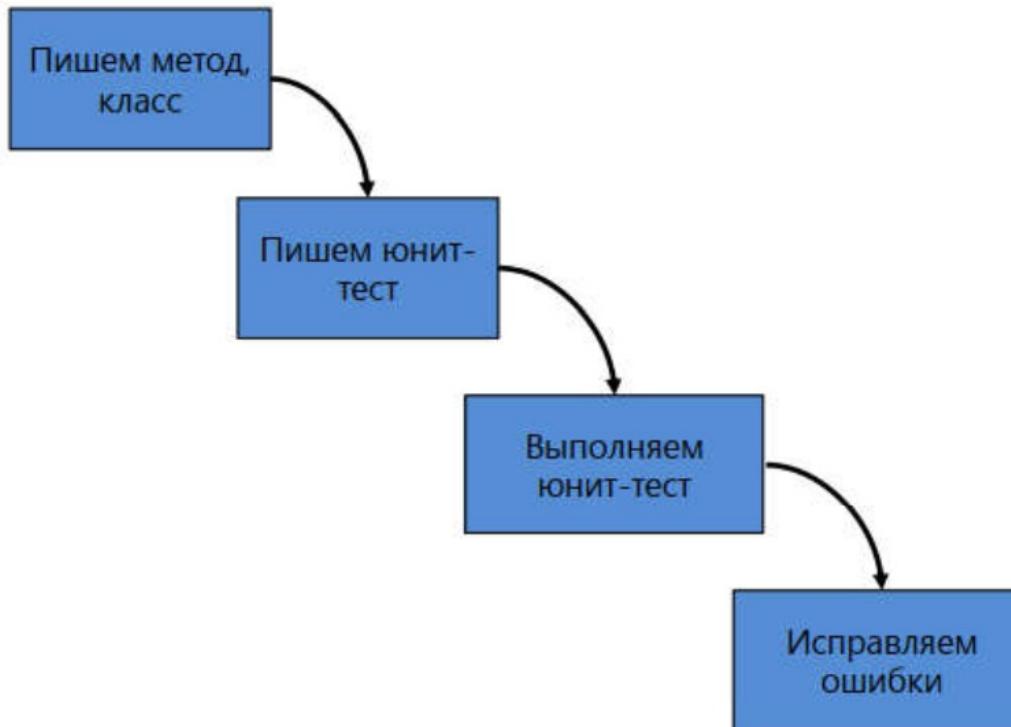
Catch vs Throws

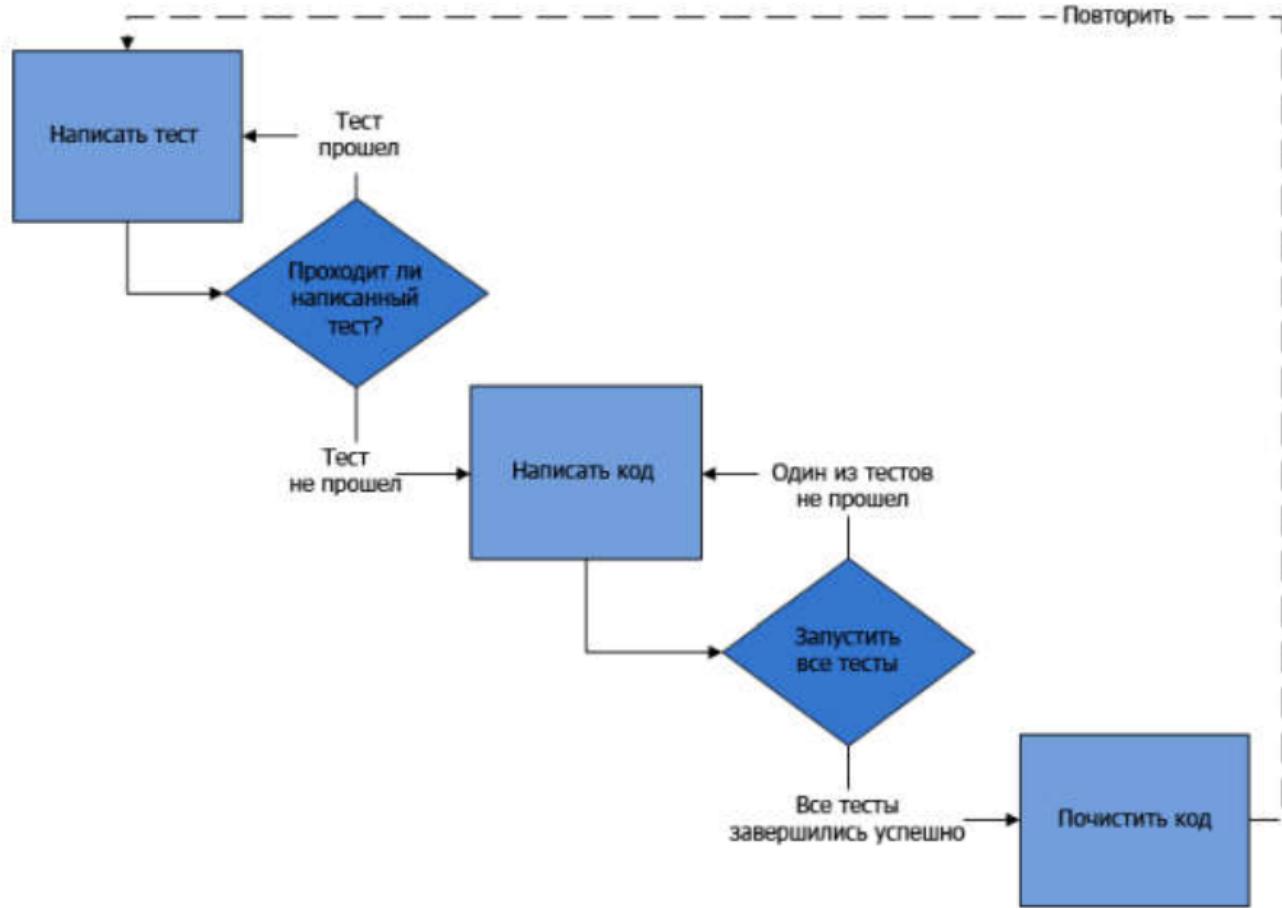


Test-Driven Development(TDD) – разработка через тестирование.
Подход разработки ПО, который заключается в написании тестов перед написанием кода.



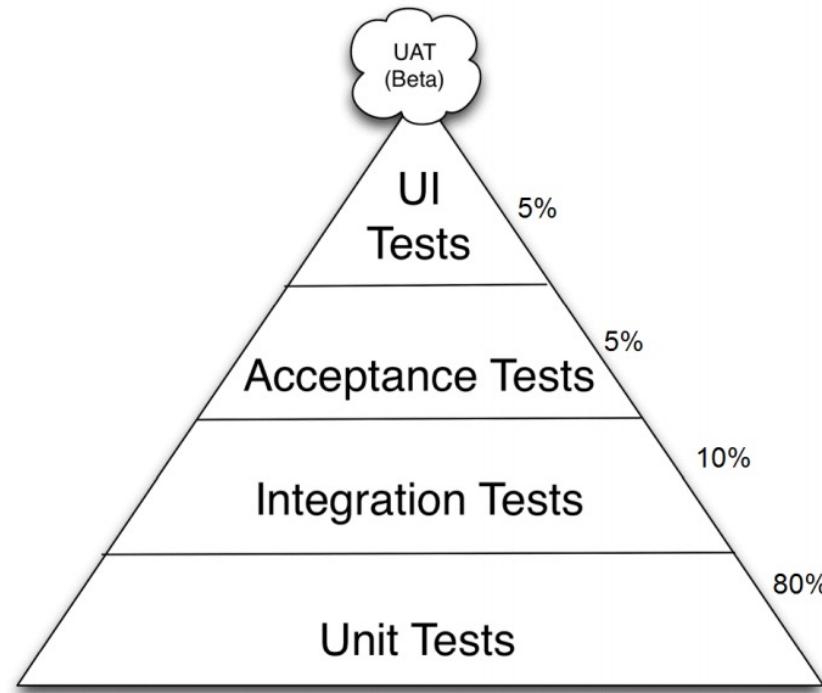
Написание тестов традиционным методом





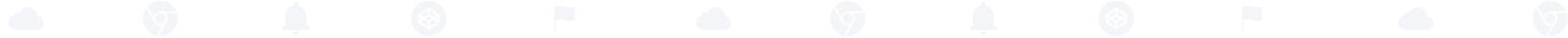


ПИРАМИДА ТЕСТИРОВАНИЯ





Unit тест – блок кода (обычно метод), который вызывает тестируемый блок кода и проверяет его правильность работы. Если результат юнит-теста не совпадает с ожидаемым результатом, тест считается не пройденным.



Свойства хорошего Unit теста (F.I.R.S.T.)

Быстрота (Fast). Тесты должны выполняться быстро. Если тесты выполняются медленно, вам не захочется часто запускать их. Без частого запуска тестов проблемы не будут выявляться на достаточно ранней стадии, когда они особенно легко исправляются. В итоге вы уже не так спокойно относитесь к чистке своего кода, и со временем код начинает загнивать.

Независимость (Independent). Тесты не должны зависеть друг от друга. Один тест не должен создавать условия для выполнения следующего теста. Все тесты должны выполняться независимо и в любом порядке на ваше усмотрение. Если тесты зависят друг от друга, то при первом отказе возникает целый каскад сбоев, который усложняет диагностику и скрывает дефекты в зависимых тестах.

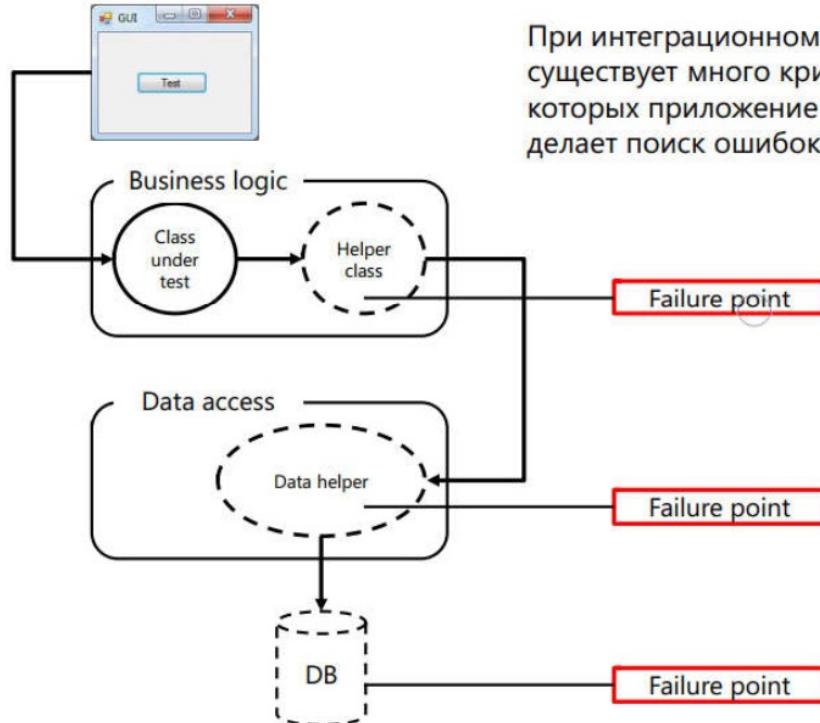
Повторяемость (Repeatable). Тесты должны давать повторяемые результаты в любой среде. Вы должны иметь возможность выполнить тесты в среде реальной эксплуатации, в среде тестирования или на вашем ноутбуке во время возвращения домой с работы. Если ваши тесты не будут давать однозначных результатов в любых условиях, вы всегда сможете найти отговорку для объяснения неудач. Также вы лишитесь возможности проводить тестирование, если нужная среда недоступна.

Очевидность (Self-Validating). Результатом выполнения теста должен быть логический признак. Тест либо прошел, либо не прошел. Чтобы узнать результат, пользователь не должен читать журнальный файл. Не заставляйте его вручную сравнивать два разных текстовых файла. Если результат теста не очевиден, то отказы приобретают субъективный характер, а выполнение тестов может потребовать долгой ручной обработки

Своевременность (Timely). Тесты должны создаваться своевременно. Модульные тесты пишутся непосредственно перед кодом продукта, обеспечивающим их прохождение. Если вы пишете тесты после кода продукта, вы можете решить, что тестирование кода продукта создает слишком много трудностей, а все из-за того, что удобство тестирования не учитывалось при проектировании кода продукта.



Интеграционное тестирование – тестирование нескольких связанных модулей программного обеспечения как единого целого.



При интеграционном тестировании существует много критических точек, в которых приложение может дать сбой, что делает поиск ошибок сложнее.



jUnit аннотации

Аннотация

Описание

`@Test`

Аннотация `@Test` определяет что метод `method()` является тестовым.

`@Before`

Аннотация `@Before` указывает на то, что метод будет выполняться перед каждым тестируемым методом `@Test`.

`@After`

Аннотация `@After` указывает на то что метод будет выполняться после каждого тестируемого метода `@Test`

`@BeforeClass`

Аннотация `@BeforeClass` указывает на то, что метод будет выполняться в начале всех тестов, а точней в момент запуска тестов(перед всеми тестами `@Test`).

`@AfterClass`

Аннотация `@AfterClass` указывает на то, что метод будет выполняться после всех тестов.

`@Ignore`

Аннотация `@Ignore` говорит, что метод будет проигнорирован в момент проведения тестирования.

`@Test (expected = Exception.class)`

(`expected = Exception.class`) – указывает на то, что в данном тестовом методе вы преднамеренно ожидается `Exception`.

`@Test (timeout=100)`

(`timeout=100`) – указывает, что тестируемый метод не должен занимать больше чем 100 миллисекунд.

`@Parameters`

Аннотация `@Parameters` позволяет создавать тесты с разными входными параметрами

`@RepeatedTest(X)`

Аннотация `@RepeatedTest(X)` позволяет запускать тест X раз



jUnit asserts

`fail([message])` — утверждение того, что тест терпит неудачу. Может использоваться для завершения теста и вывода причины провала в случае обнаружения некорректной работы в процессе тестирования.

`assertEquals([message], expected, actual)` — утверждение эквивалентности. Сравнивает ожидаемый и полученный результат. При сравнении примитивных типов проверяется равенство значений. Для сравнения объектов используется метод `equals()`, если он определен. Если же он не определен – работает как `assertSame`. (Для массивов не подходит, т.к. сравнивает просто ссылки!).

`assertArrayEquals([message], expected, actual)` – поэлементно сравнивает массивы.

`assertSame([message], expected, actual)` – просто сравнивает объекты при помощи оператора `==`, то есть проверяет, являются ли параметры ссылками на один и тот же объект.

`assertNotSame([message], expected, actual)` – утверждение обратное `assertSame`.

`assertTrue ([message], boolean condition)` – булево утверждение. Проверяет на истинность логическое условие.

`assertFalse ([message], boolean condition)` – булево утверждение. Проверяет на истинность логическое условие.

`assertNull([message], object)` – Null утверждение. Проверяет содержимое объектной переменной на Null значение.

`assertNotNull([message], object)` – утверждение обратное `assertNull`.



Тестовые объекты

- **Dummy** – пустые объекты, которые передаются в вызываемые внутренние методы, но не используются. Предназначены лишь для заполнения параметров методов.
- **Fake** – объекты, имеющие работающие реализации, но в таком виде, который делает их неподходящими для production-кода (например, [In Memory Database](#)).
- **Stub** – объекты, которые предоставляют заранее заготовленные ответы на вызовы во время выполнения теста и обычно не отвечающие ни на какие другие вызовы, которые не требуются в teste. Также могут запоминать какую-то дополнительную информацию о количестве вызовов, параметрах и возвращать их потом тесту для проверки.
- **Mock** – объекты, которые заменяют реальный объект в условиях teste и позволяют проверять вызовы своих членов как часть системы или unit-теста. Содержат заранее запрограммированные ожидания вызовов, которые они ожидают получить. Применяются в основном для т.н. interaction (behavioral) testing.