

ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ И ИХ ОБРАБОТКА ТЕСТИРОВАНИЕ ИСКЛЮЧЕНИЙ

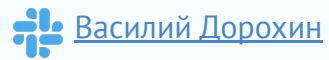


ВАСИЛИЙ ДОРОХИН



ВАСИЛИЙ ДОРОХИН

QuadCode, QA Engineer





План занятия

1. [Проблемы](#)
2. [Подходы к работе с ошибками](#)
3. [try, catch, finally](#)
4. [Ключевые правила](#)
5. [Генерация исключений](#)
6. [Тестирование исключений](#)
7. [Итоги](#)



ПРОБЛЕМЫ

ПРОБЛЕМЫ

На предыдущих лекциях при попытке использования **null** в качестве объекта, при выходе за границы массива или при приведении к неверному типу мы достаточно часто сталкивались с сообщениями об ошибках:

```
java.lang.ClassCastException: class ru.netology.domain.Book cannot be cast to class ru.netology.domain.TShirt (ru.netology.domain.Book and ru.netology.domain.TShirt are in unnamed module of loader 'app')
```

```
+ at ru.netology.domain.BookTest.shouldNotCastToDifferentClass(BookTest.java:26) <19 internal calls>
+ at java.base/java.util.ArrayList.forEach(ArrayList.java:1540) <9 internal calls>
+ at java.base/java.util.ArrayList.forEach(ArrayList.java:1540) <21 internal calls>
```



ПРОБЛЕМЫ

Давайте разбираться, что это и почему в случае возникновения каких-то ошибок, Java реагирует именно так и обрушает нашу программу.



ПОДХОДЫ К РАБОТЕ С ОШИБКАМИ

CI

Системы CI предназначены для запуска различных инструментов и получения общего итога работы: **Success** или **Fail**.

Как они узнают, завершилась работа того же Maven успешно или нет?



КОДЫ ЗАВЕРШЕНИЯ

На самом деле, есть общепринятое соглашение: каждая выполняемая программа может установить определённый код завершения (целое число, которое сигнализирует о том, как завершилась программа).

Общепринято, что 0 — это признак успешного завершения, а любое другое число — признак ошибки.

КОДЫ ЗАВЕРШЕНИЯ

```
public class Main {  
    public static void main(String[] args) {  
        Object object = null;  
        System.out.println(object.toString());  
    }  
}
```

Exception in thread "main" java.lang.NullPointerException
at ru.netology.Main.main([Main.java:6](#))

Process finished with exit code **1**

код завершения



КОДЫ ЗАВЕРШЕНИЯ

Если вы запускаете приложение из командной строки, то проверить код завершения можно:

1. В Windows: **echo %errorlevel%**
2. В *nix: **echo \$?**

Анализируя коды завершения, CI узнаёт, завершилась ли определённая команда успешно (ведь именно CI запускает эти команды).

КОДЫ ЗАВЕРШЕНИЯ

Если почитать документацию, то выяснится, что **только несколько кодов** завершения специфицированы:

- 0 — успешно
- 1 — для всех ошибок общего типа
- $128 + n$ — завершение приложения* с помощью отправки сигнала (n)

Таким образом, никакой унификации и требований к проверке кодов завершения — нет.

Примечание*: приложению (например, JVM) можно отправить сигнал о том, что необходимо завершить свою работу. Тогда JVM завершается с кодом 143.



КОДЫ ЗАВЕРШЕНИЯ

Почему бы не перенести эту модель на все методы? Пусть каждый метод возвращает код завершения.

Вопрос к аудитории: как вы думаете, какие проблемы это может вызвать?



КОДЫ ЗАВЕРШЕНИЯ

Вам полезно знать про коды как про хороший механизм.

Но этот механизм обладает несколькими недостатками:

1. В больших приложениях с тысячами классов* и десятками тысяч методов — кодов ошибок на всех не хватит
2. Наличие кода не заставляет программиста его (этот код) обрабатывать
3. В Java методы могут возвращать только одно значение (а конструкторы вообще ничего не возвращают)

Примечание*: только в стандартной библиотеке Java больше нескольких тысяч классов.

ИСКЛЮЧЕНИЯ

Поэтому, в Java использовали особый механизм, который называется **исключения** (или **исключительная ситуация**).

Пример из жизни: представим, что вы за рулём автомобиля, спокойно передвигаетесь и слушаете музыку.

Вдруг у вас пробивает колесо — конечно же, вы экстренно останавливаетесь и выходите смотреть, что произошло.

Если вы можете самостоятельно устранить проблему, вы это делаете; если нет — то ожидаете эвакуатора.

ИСКЛЮЧЕНИЯ

Поэтому, в Java использовали особый механизм, который называется **исключения** (или **исключительная ситуация**).

Пример из жизни: представим, что вы за рулём автомобиля, спокойно передвигаетесь и слушаете музыку.

обычный ход выполнения
программы

Вдруг у вас пробивает колесо — конечно же, вы экстренно останавливаетесь и выходите смотреть, что произошло.

исключение
прерывание обычного
хода выполнения
программы

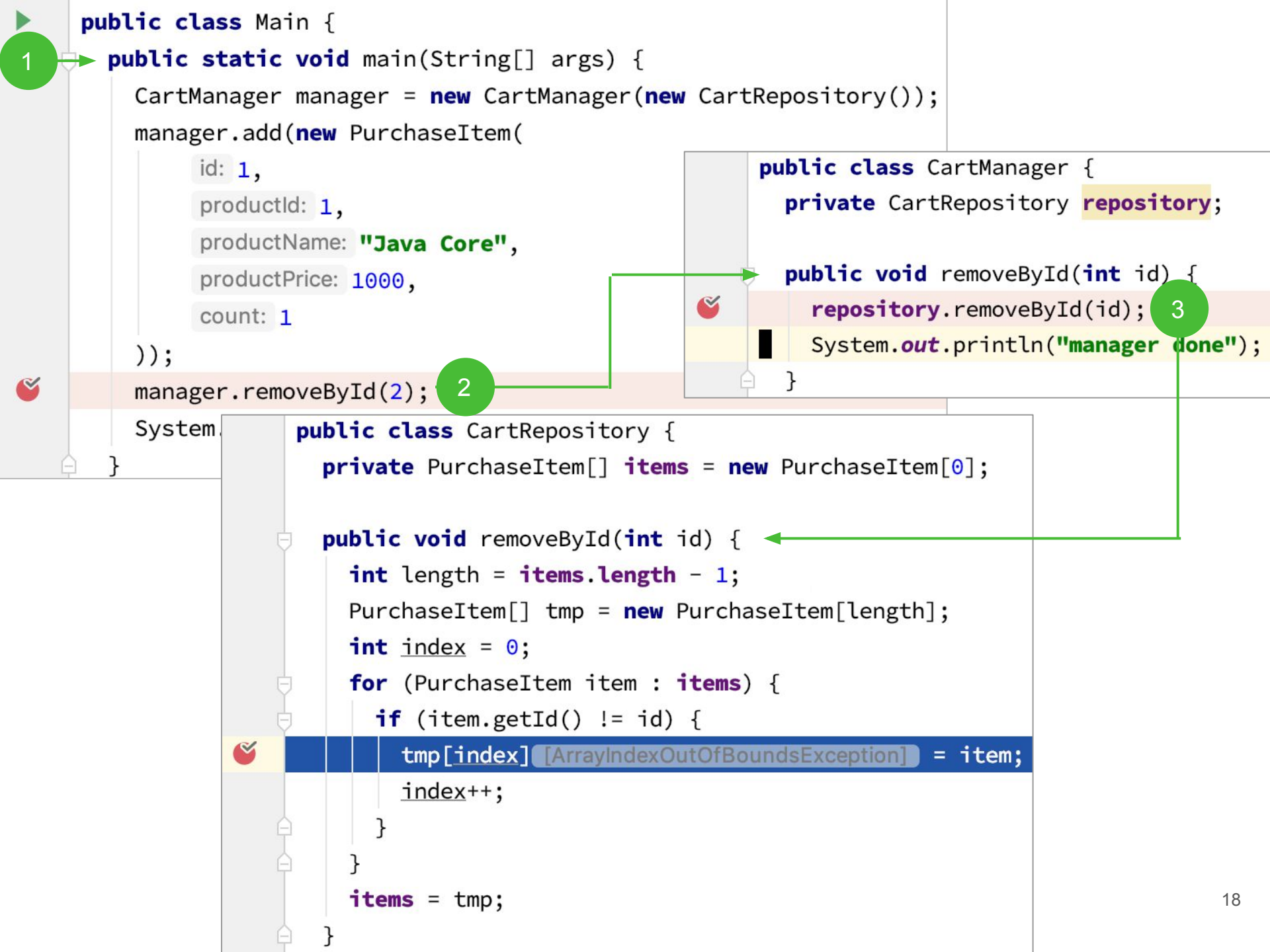
Если вы самостоятельно можете устранить проблему, вы это делаете; если нет — то ожидаете эвакуатора и т.д.

обработка исключения



ИСКЛЮЧЕНИЯ

Исключения — специальный механизм, при срабатывании которого прерывается нормальный ход выполнения программы и стек вызовов раскручивается до тех пор, пока исключение не будет обработано.

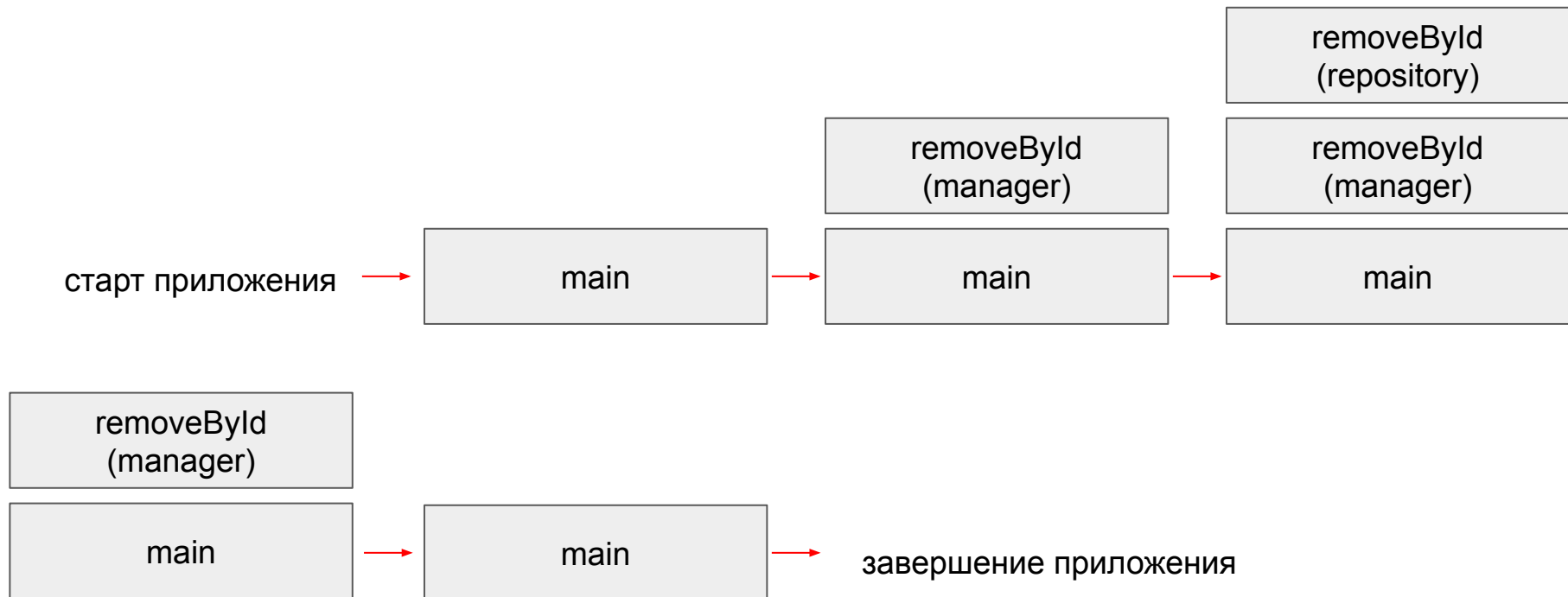




НОРМАЛЬНЫЙ ХОД ВЫПОЛНЕНИЯ

Начинается всё с того, что мы заходим в метод `main` и вызываем `removeByld` менеджера. В методе менеджера мы вызываем метод `removeByld` репозитория. Если метод репозитория успешно завершится, то мы вернёмся в метод `removeByld` менеджера, откуда вернёмся обратно в `main`:

НОРМАЛЬНЫЙ ХОД ВЫПОЛНЕНИЯ



СТЕК ВЫЗОВОВ

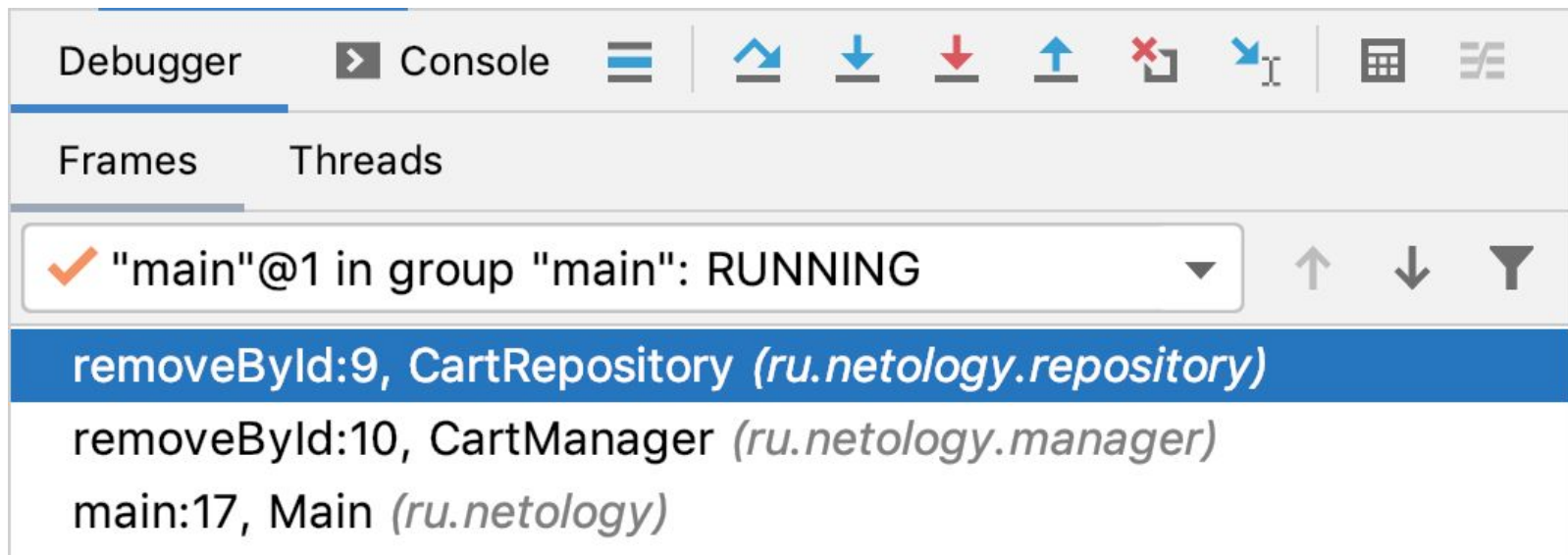
Получается, что вызовы методов «стопочкой» складываются друг на друга.

При этом, когда метод отработывает до конца (доходит до **return** или до закрывающей фигурной скобки), то он «убирается» из этой «стопочки» и продолжается выполнение с того места, где этот метод был вызван.

Такая структура LIFO (Last Input First Output — последним пришёл, первым ушёл) называется стек. А применительно к методам — **стек вызовов**.

СТЕК ВЫЗОВОВ

В дебаггере стек вызовов показан в отдельном окошке:



Продемонстрировать в дебаггере наполнение стека и опустошение его (закомментировать тело цикла)

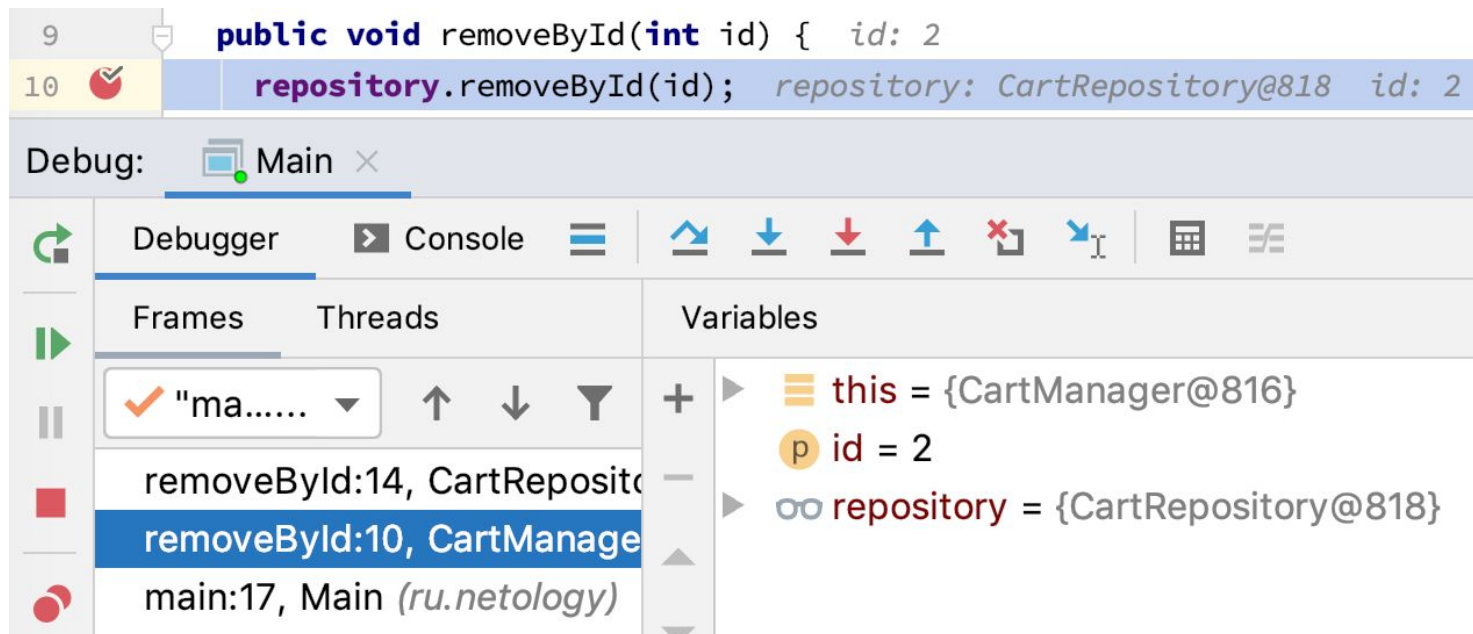
Q & A

Q: Можно ли в дебаггере сделать шаг назад?

A: Нет, нельзя. В IDEA есть опция Drop Frame, которая позволяет «убрать» вызов метода из стека, но это не шаг назад.

Q: Что будет, если кликнуть на метод из стека?

A: Будет показана область видимости метода, на котором кликнули:





ИСКЛЮЧЕНИЯ

Мы разобрали, что такое нормальный ход выполнения и стек вызовов, теперь обсудим, что происходит при возникновении исключения.


```

public class Main {
    public static void main(String[] args) {
        CartManager manager = new CartManager(new CartRepository());
        manager.add(new PurchaseItem(
            id: 1,
            productId: 1,
            productName: "Java Core",
            productPrice: 1000,
            count: 1
        ));
        manager.removeById(2);
        System.out.println("manager done");
    }
}

```

```

public class CartManager {
    private CartRepository repository;

    public void removeById(int id) {
        repository.removeById(id);
        System.out.println("manager done");
    }
}

```

```

public class CartRepository {
    private PurchaseItem[] items = new PurchaseItem[0];

    public void removeById(int id) {
        int length = items.length - 1;
        PurchaseItem[] tmp = new PurchaseItem[length];
        int index = 0;
        for (PurchaseItem item : items) {
            if (item.getId() != id) {
                tmp[index] = item;
                index++;
            }
        }
        items = tmp;
    }
}

```

ИСКЛЮЧЕНИЯ

При возникновении исключения прерывается нормальный ход выполнения приложения и:

1. Следующие строки в этом методе не выполняются
2. Управление возвращается обратно в метод, который вызвал текущий

В вызывающем методе:

1. Следующие строки в вызывающем методе не выполняются
2. Управление возвращается обратно в метод, который вызвал текущий

И так происходит до тех пор, пока не дойдём до `main`.

STACK TRACE

Далее JVM обрабатывает это исключение, печатая Stack Trace, и аварийно завершает работу с ненулевым кодом:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds for length 0
    at ru.netology.repository.CartRepository.removeById(CartRepository.java:14)
    at ru.netology.manager.CartManager.removeById(CartManager.java:10)
    at ru.netology.Main.main(Main.java:17)
```

Process finished with exit code **1**

Stack Trace

Кликабельно

Важно: Stack Trace печатается от точки, где произошло исключение, то точки, где приложение было завершено.

STACK TRACE

Класс
исключения

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds for length 0
at ru.netology.repository.CartRepository.removeById([CartRepository.java:14](#))
at ru.netology.manager.CartManager.removeById([CartManager.java:10](#))
at ru.netology.Main.main([Main.java:17](#))

1

2

3

где был вызван метод,
в котором произошло
исключение

где произошло
исключение

где был вызван метод,
в котором был вызван
метод, в котором
произошло исключение

IDEA подчёркивает в
виде ссылок ваш код
(который находится в
вашем проекте),
он кликабелен

STACK TRACE

Умение читать Stack Trace критически важно: **вы должны научиться их читать.**

Техника очень простая: вы пролистываете лог до тех пор, пока не встречаете строку «**Exception in thread**» и дальше целиком читаете строку исключения: «**java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds for length 0**», чтобы понять что конкретно пошло не так.

А дальше уже кликаете по Stack Trace, чтобы понять, какие вызовы привели к возникновению исключения.



STACK TRACE & BUG REPORTS

Всегда включайте полный Stack Trace в баг-репорт в качестве приложения!

Это **ключевая информация** при анализе поведения приложения.



TRY CATCH FINALLY



ИСКЛЮЧЕНИЯ

При возникновении исключения также создаётся объект исключения, содержащий информацию о возникшем исключении.

Этот объект «пробрасывается» через все вызовы методов.

TRY CATCH

Java предоставляет синтаксическую конструкцию **try-catch**, которая позволяет перехватывать исключения и обрабатывать их (восстанавливая «нормальный ход выполнения приложения»:

```
try {  
    System.out.println("before remove");  
    manager.removeById(2);  
    System.out.println("after remove");  
} catch (ArrayIndexOutOfBoundsException e) {  
    System.out.println("specific catch");  
} catch (RuntimeException e) {  
    System.out.println("runtime catch");  
} catch (Exception e) {  
    System.out.println("catch");  
}
```

если в этом блоке произойдёт исключение, то попадаем в catch

проверяется тип исключения, попадаем только при совпадении

```
System.out.println("main done"); // for demo only
```

TRY

Блок **try** срабатывает либо целиком, либо до той точки, в которой произошло исключение:

```
try {  
    System.out.println("before remove");  
    manager.removeById(2);  
    System.out.println("after remove");  
}
```



ЭТОТ КОД НЕ ВЫПОЛНИТСЯ

Если исключение произошло, то ищется соответствующий **catch** исходя из соответствия классов объекта исключения и того, что указан в блоке **catch**



CATCH

Блок **catch** сопоставляет классы следующим образом: сверху вниз выбирает первый (остальные игнорируются).

Важное замечание: **catch** и **instanceof** на самом деле смотрят не соответствие типов, а **приводимость**.

ПРИВОДИМОСТЬ ТИПОВ

Приводимость типов определяется исходя из того, находится ли проверяемый тип в цепочке наследования*.

Таким образом, самое важное правило: **первым всегда писать самый специфичный тип.**

```
} catch (ArrayIndexOutOfBoundsException e) {  
    System.out.println("specific catch");  
}  
} catch (RuntimeException e) {  
    System.out.println("runtime catch");  
}  
} catch (Exception e) {  
    System.out.println("catch");  
}
```

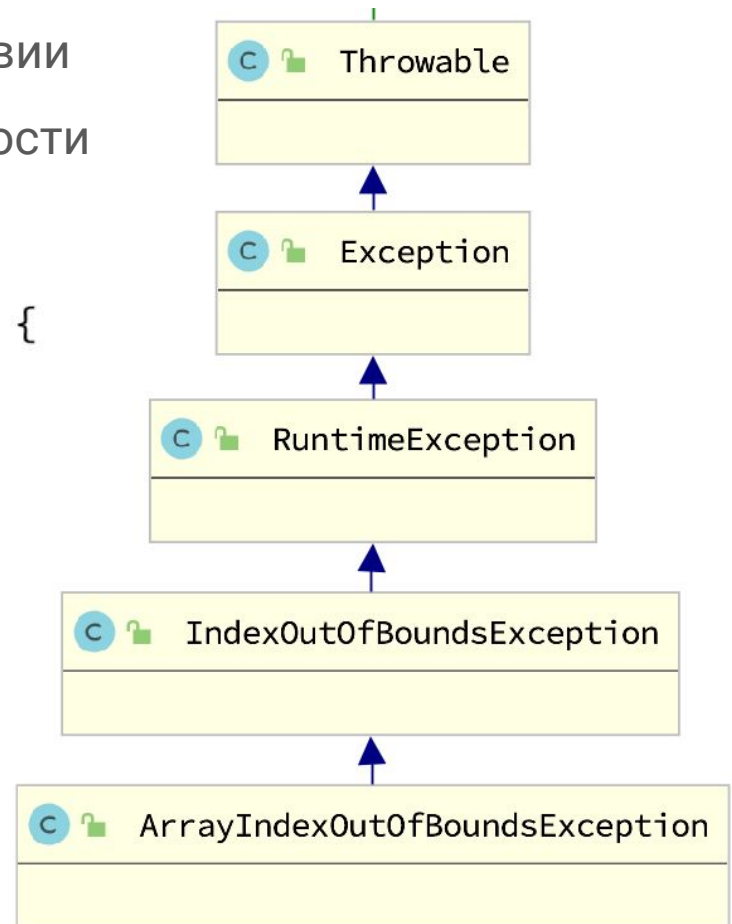
Примечание*: мы дополним это определение в лекции про интерфейсы.

ПРИВОДИМОСТЬ ТИПОВ

Блок **catch** выполняется только при условии возникновения исключения и приводимости типа исключения к указанному.

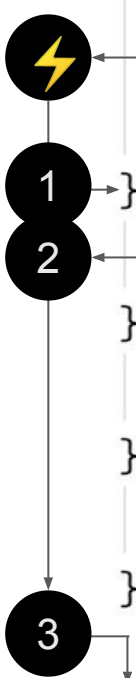
```
} catch (ArrayIndexOutOfBoundsException e) {  
→ System.out.println("specific catch");  
}  
} catch (RuntimeException e) {  
  System.out.println("runtime catch");  
}  
} catch (Exception e) {  
  System.out.println("catch");  
}
```

выполнится только этот блок:
первый, приводимый по типу



CATCH

```
try {  
    System.out.println("before remove");  
    manager.removeById(2);  
    System.out.println("after remove");  
1 → } catch (ArrayIndexOutOfBoundsException e) {  
2 →     System.out.println("specific catch");  
    } catch (RuntimeException e) {  
        System.out.println("runtime catch");  
    } catch (Exception e) {  
        System.out.println("catch");  
    }  
3 → }  
    System.out.println("main done"); // for demo only
```



CATCH

Блок **catch** выполняется, только если возникают исключения и приводимости типа исключения к указанному.

Блок **catch** выполняется либо целиком, либо до точки, где возникло исключение.

Да-да, теперь в любой точке может возникнуть исключение, даже в блоке **catch** 🐱



CATCH

Если в блоке **catch** возникнет исключение, то текущая конструкция **try** уже не обрабатывает исключение, оно «уходит» вверх.



FINALLY

Помимо **try** блока допускается использование блока **finally**, который выполняется независимо от того, было исключение в блоке **try** или нет (а также было ли исключение в блоке **catch**, который перехватил исключение в блоке **try** или нет).

FINALLY

```
try {  
    System.out.println("before remove");  
    manager.removeById(2);  
    System.out.println("after remove");  
1  } catch (ArrayIndexOutOfBoundsException e) {  
2  System.out.println("specific catch");  
    } catch (RuntimeException e) {  
        System.out.println("runtime catch");  
    } catch (Exception e) {  
        System.out.println("catch");  
3  } finally {  
4  System.out.println("finally");  
    }  
5  
    System.out.println("main done"); // for demo only
```



TRY, CATCH, FINALLY

Блок **try** в этой конструкции обязан встречаться только один раз.

Блоков **catch** может быть сколько угодно (0+).

Блок **finally** в этой конструкции может встречаться только один раз.

При этом **try** обязателен + должен быть* хотя бы один **catch** и/или **finally**.

Примечание*: мы уточним это требование при разборе конструкции **try with resources**.



КЛЮЧЕВЫЕ ПРАВИЛА

КОГДА ИСПОЛЬЗОВАТЬ

После изучения блока конструкции **try-catch** иногда возникает желание использовать её везде, чтобы сделать нашу программу стабильной! Ведь мы можем перехватить всё, и программа не обрушиться!

Это очень **плохая идея**. Использовать нужно только там, где вы действительно знаете, как вы можете обработать исключение. Например, вы шлёте запрос во внешнюю систему и знаете, что могут быть проблемы — тогда вы ставите **try-catch** (как говорят, «оборачиваете в **try-catch**”) и перехватив исключение, можете сообщить пользователю, что операция не удалась.



LET IT CRASH

Достаточно часто мы специально не обрабатываем исключения (поскольку не можем предусмотреть всё) и даём системе (либо её части) упасть.

Потому что когда она упадёт, мы об этом узнаем и начнём анализировать, что пошло не так и почему.

PRINTSTACKTRACE

У объекта исключения есть замечательный метод, который и печатает стектрейс:

```
try {  
    System.out.println("before remove");  
    manager.removeById(2);  
    System.out.println("after remove");  
} catch (ArrayIndexOutOfBoundsException e) {  
    e.printStackTrace();
```

```
java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds for length 0  
    at ru.netology.repository.CartRepository.removeById(CartRepository.java:14)  
    at ru.netology.manager.CartManager.removeById(CartManager.java:10)  
    at ru.netology.Main.main(Main.java:23)
```

Обязательно его используйте, иначе в логах приложения не сохранится информация о том, что действительно пошло не так.

EMPTY CATCH

Пустой блок считается одним из «грехов» программиста и крайне не рекомендуется к использованию:

```
try {  
    System.out.println("before remove");  
    manager.removeById(2);  
    System.out.println("after remove");  
} catch (Throwable e) {  
  
}
```

Empty 'catch' block

[Rename 'catch' parameter to 'ignored'](#)



[More actions...](#)



Q: Почему?

A: Потому что ваше приложение работает не по «обычному сценарию», а об этом никто никогда не узнает (пока не станет поздно).



TRY-CATCH И ЛОГИКА

Старайтесь не строить на **try-catch** бизнес-логику, для этого есть стандартные конструкции (**if** и другие)*.

Примечание*: на самом деле в коде стандартной библиотеки и внешних библиотек достаточно часто **try-catch** используется именно для организации логики, но сделано это не от «хорошей жизни». Просто другой возможности организовать подобную логику либо нет, либо получается в разы сложнее.



ГЕНЕРАЦИЯ ИСКЛЮЧЕНИЙ



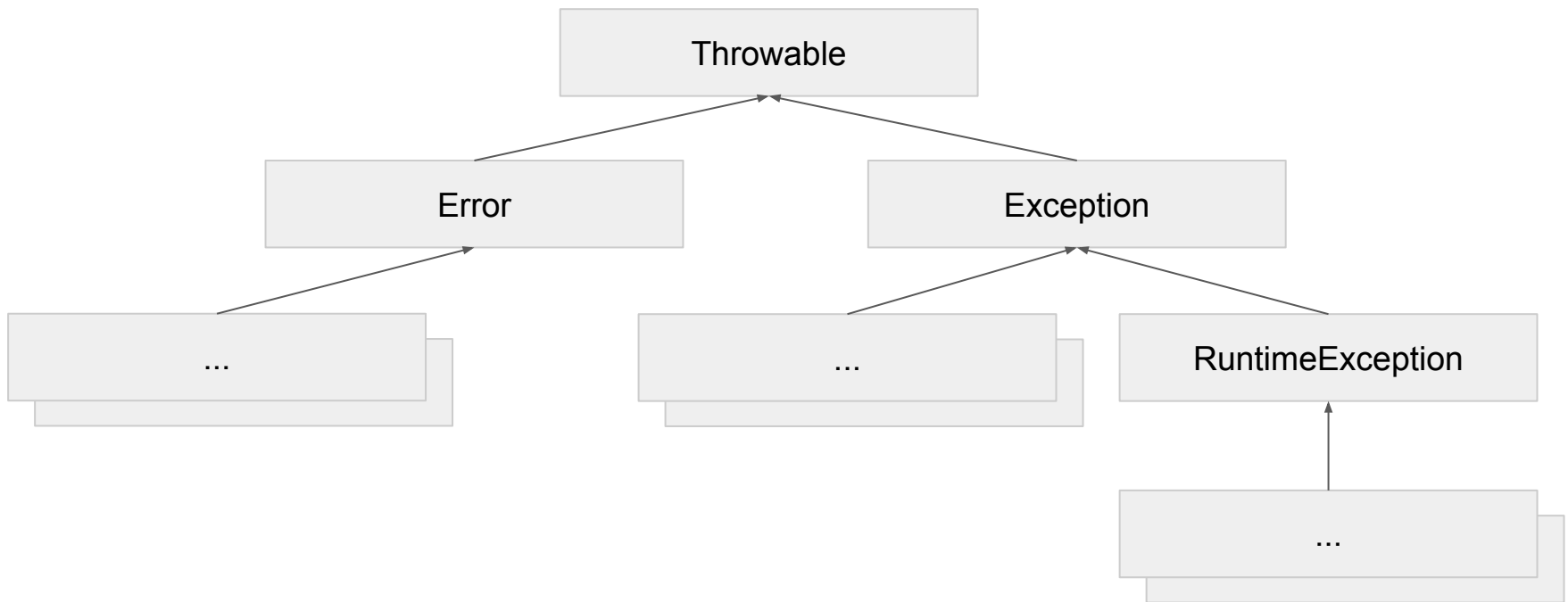
СОЗДАНИЕ ИСКЛЮЧЕНИЙ

Q: Мы посмотрели как обрабатывать исключения. Нужно ли создавать собственные исключения? Если да, то как?

A: Да, обязательно и желательно разрабатывать свои собственные исключения, чтобы вы могли отличить их от исключений стандартной библиотеки или других библиотек.

СОЗДАНИЕ ИСКЛЮЧЕНИЙ

Для того, чтобы создать исключение, надо отнаследоваться от класса `Throwable` или одного из его наследников:



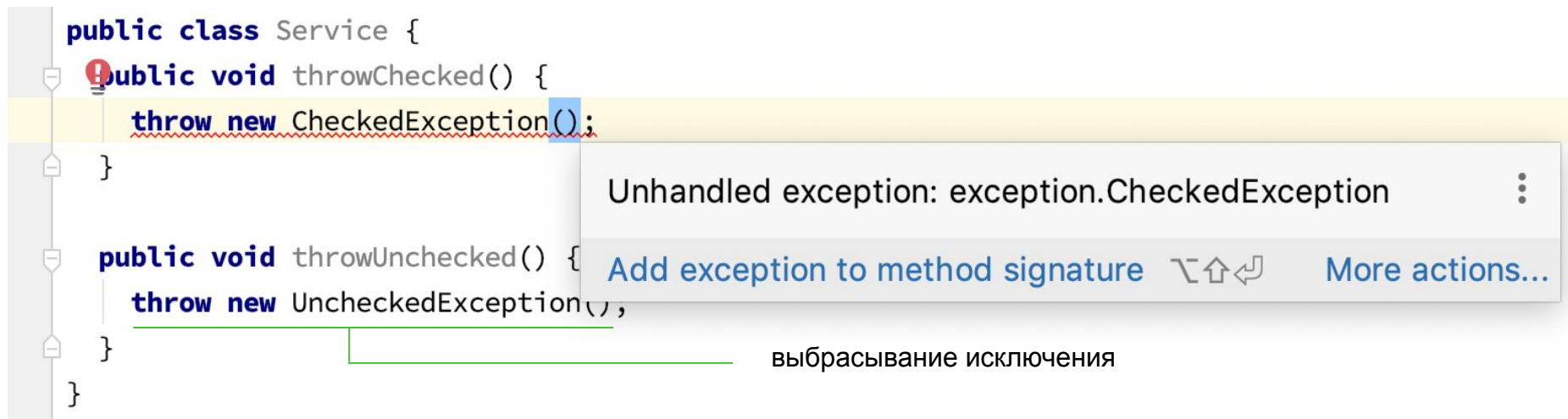
ИЕРАРХИЯ ИСКЛЮЧЕНИЙ

Как тестировщик, вы редко будете сами создавать собственные исключения, но общую идеологию знать должны:

1. `Throwable` — только объекты класса, унаследованного от этого класса, могут быть исключениями (сам `Throwable` наследуется от `Object`).
2. `Error` — ошибки, не предназначенные для перехватывания (например, JVM не хватает памяти, ошибка чтения файла из-за проблем с ФС) и т.д.
3. `Exception` — «проверяемые» (checked) исключения, методы должны их либо обрабатывать, либо указывать в сигнатуре.
4. `RuntimeException` — «непроверяемые» (unchecked) исключения, методы могут их обрабатывать (на своё усмотрение).

ИСКЛЮЧЕНИЯ

В подавляющем большинстве случаев вы будете работать только с Checked и Unchecked исключениями. Давайте посмотрим, в чём заключаются отличия при работе с ними:



Т.е. мы не можем просто так «выбрасывать» Checked исключения (а Unchecked можем).

CHECKED EXCEPTIONS

Checked Exceptions должны быть либо завёрнуты в блок **try-catch**, либо вынесены в сигнатуру метода:

```
public class Service {  
    public void throwChecked() {  
        throw new CheckedException();  
    }  
  
    public void throwUnchecked() {  
        throw new UncheckedException();  
    }  
}
```

! Add exception to method signature

! Annotate method 'throwChecked' as @SneakyThrows

! Surround with try/catch

Press \Space to open preview

CHECKED EXCEPTIONS

Но если их вынести в сигнатуру метода, то любой метод, вызывающий наш метод, должен будет либо обернуть его в **try-catch**, либо записать себе в сигнатуру генерируемое исключение:

```
public class Service {  
    public void throwChecked() throws CheckedException {  
        throw new CheckedException();  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Service service = new Service();  
        service.throwChecked();  
    }  
}
```



service.throwChecked();



Add exception to method signature



Annotate method 'main' as @SneakyThrows




Surround with try/catch

Press `\Space` to open preview

CHECKED & UNCHECKED EXCEPTIONS

```
public class Main {  
    public static void main(String[] args) {  
        Service service = new Service();  
  
        try {  
            service.throwChecked();  
        } catch (CheckedException e) {  
            e.printStackTrace();  
        }  
  
        service.throwUnchecked();  
    }  
}
```



Заворачивание в **try-catch**
(Alt + Enter в IDEA)

Таким образом, Checked Exceptions используются тогда, когда хотят заставить программиста явно обрабатывать исключения (т.к. в противном случае код не скомпилируется).



CHECKED & UNCHECKED EXCEPTIONS

В современном мире двоякое отношение к Checked Exceptions, вплоть до того, что многие популярные инструменты стараются их избегать и даже делают из Checked Unchecked (как это происходит, вы сможете прочесть в доп.материалах к лекции).

Q & A

A: А когда стоит «выбрасывать» исключения?

Q: Для начинающих программистов правило звучит так: «в любой непонятной ситуации кидай Exception».

В целом же, исключения — это тот механизм, который позволяет отреагировать на неправильное использование API: например, вы пытаетесь создать Кондиционер с отрицательной максимальной температурой — прямо в конструкторе можно выкинуть исключение, или можно выкидывать исключения при попытке удаления несуществующего объекта из корзины.

Q & A

A: Можно ли выкидывать исключения в тестах?

Q: Нет, у вас есть `assert`'ы, а если вы хотите просто «завалить» тест, есть метод *fail*.



ТЕСТИРОВАНИЕ ИСКЛЮЧЕНИЙ

ТЕСТИРОВАНИЕ ИСКЛЮЧЕНИЙ

Для тестирования исключений мы будем использовать специальную конструкцию, которая называется лямбда-выражение:

```
» public class ServiceTest {  
    private Service service = new Service();  
  
    @Test  
    public void shouldThrowCheckedException() {  
        assertThrows(CheckedException.class, () -> service.throwChecked());  
    }  
    }  
  
    @Test  
    public void shouldThrowUncheckedException() {  
        assertThrows(UncheckedException.class, () -> service.throwUnchecked());  
    }  
}
```

lambda expression

lambda expression

ЛЯМБДА-ВЫРАЖЕНИЕ

Пока для нас **Лямбда-выражение** будет представлять конструкцию вида:

() -> вызов метода, который должен сгенерировать исключение

Такая конструкция в совокупности с *assertThrows* позволит нам избежать использования **try-catch**.



ЛЯМБДА-ВЫРАЖЕНИЕ

Что такое лямбда выражения на самом деле, как их создавать и как с ними работать, мы будем разбирать на лекциях, связанных с интерфейсами и обобщённым программированием.

ВАЖНО

За использование **try-catch** в автотестах ДЗ будут отправляться на доработку, т.к. это считается «незнанием» возможностей JUnit (т.к. внутри это всё уже сделано):

```
try {
    executable.execute();
}
catch (Throwable actualException) {
    if (expectedType.isInstance(actualException)) {
        return (T) actualException;
    }
    else {
        BlacklistedExceptions.rethrowIfBlacklisted(actualException);
        String message = buildPrefix(nullSafeGet(messageOrSupplier))
            + format(expectedType, actualException.getClass(), message: "Unexpected exception type thrown");
        throw new AssertionError(message, actualException);
    }
}
```



ИТОГИ



ИТОГИ

Сегодня мы рассмотрели важную тему исключений.

Обязательно разберитесь в ней, поскольку это один из краеугольных камней, на котором строится разработка на Java.

Кроме того, вы обязательно должны научиться «читать» Stack Trace и разбираться, что происходит. При написании автотестов вы будете каждый день сталкиваться со Stack Trace. Неумение разбираться в них фактически будет означать вашу проф.непригодность.



ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше [домашнее задание](#).

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



Задавайте вопросы и напишите отзыв о лекции!

ВАСИЛИЙ ДОРОХИН

 Василий Дорохин