

EXEPTIONS

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

CI

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

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.

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).

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`.

Вам полезно знать про коды как про хороший механизм.

Но этот механизм обладает несколькими недостатками:

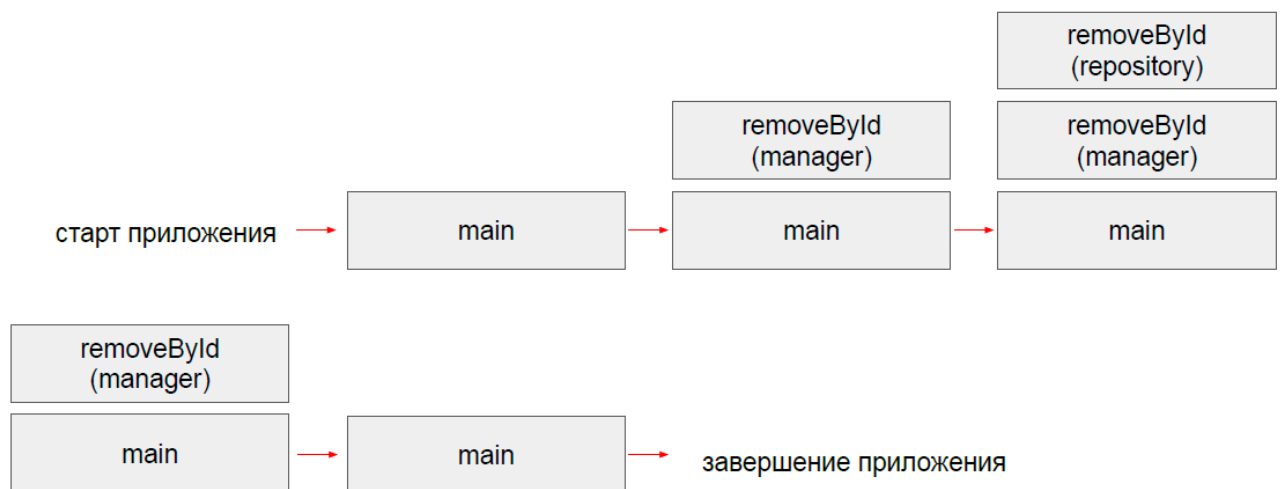
- 1 В больших приложениях с тысячами классов* и десятками тысяч методов — кодов ошибок на всех не хватит
- 2 Наличие кода не заставляет программиста его (этот код) обрабатывать

3. В Java методы могут возвращать только одно значение (а конструкторы вообще ничего не возвращают)

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

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

НОРМАЛЬНЫЙ ХОД ВЫПОЛНЕНИЯ



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.

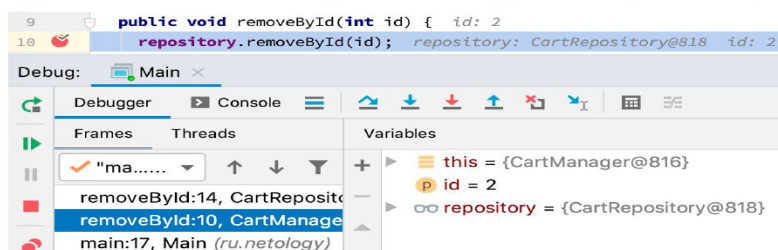
Такая структура **LIFO** (Last Input First Output — последним пришёл, первым ушёл) называется стек. А применительно к методам — **стек вызовов**.

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:



При возникновении исключения прерывается нормальный ход выполнения приложения и:

- Следующие строки в этом методе не выполняются
- Управление возвращается обратно в метод, который вызвал текущий

В вызывающем методе:

- Следующие строки в вызывающем методе не выполняются
- Управление возвращается обратно в метод, который вызвал текущий

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

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 где был вызван метод, в котором произошло исключение

2 где произошло исключение

3 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

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.

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

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

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 произойдет исключение, то попадаем в catch

проверяется тип исключения, попадаем только при совпадении

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

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");  
}
```

этим кодом не выполняется

Если исключение произошло, то ищется соответствующий **catch** исходя из соответствия классов объекта исключения и того, что указан в блоке **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 определяется исходя из того, находится ли проверяемый тип в цепочке наследования*.

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

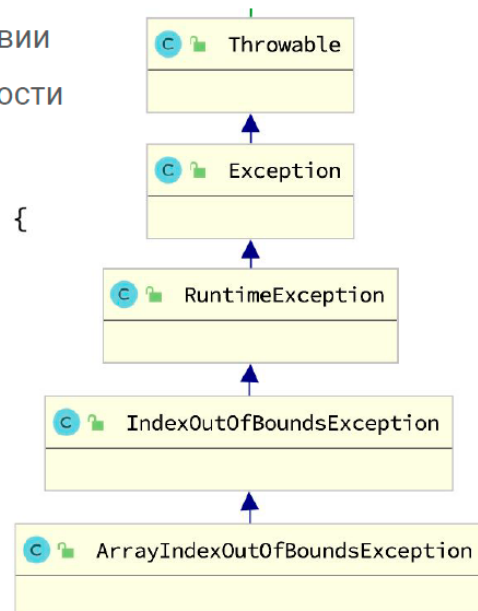
```
} catch (ArrayIndexOutOfBoundsException e) {  
    System.out.println("specific catch");  
}  
} catch (RuntimeException e) {  
    System.out.println("runtime catch");  
}  
} catch (Exception e) {  
    System.out.println("catch");  
}
```

ПРИВОДИМОСТЬ ТИПОВ

Блок **catch** выполняется только при условии возникновении 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  } catch (RuntimeException e) {
    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
```

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

Блок **catch** выполняется либо целиком, либо до точки, где возниклоисключение.

Да-да, теперь в любой точке может возникнуть исключение, даже вблоке **catch** 😺

CATCH

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

Помимо **try** блока допускается использование блока **finally**, который исполняется 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).

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:

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.

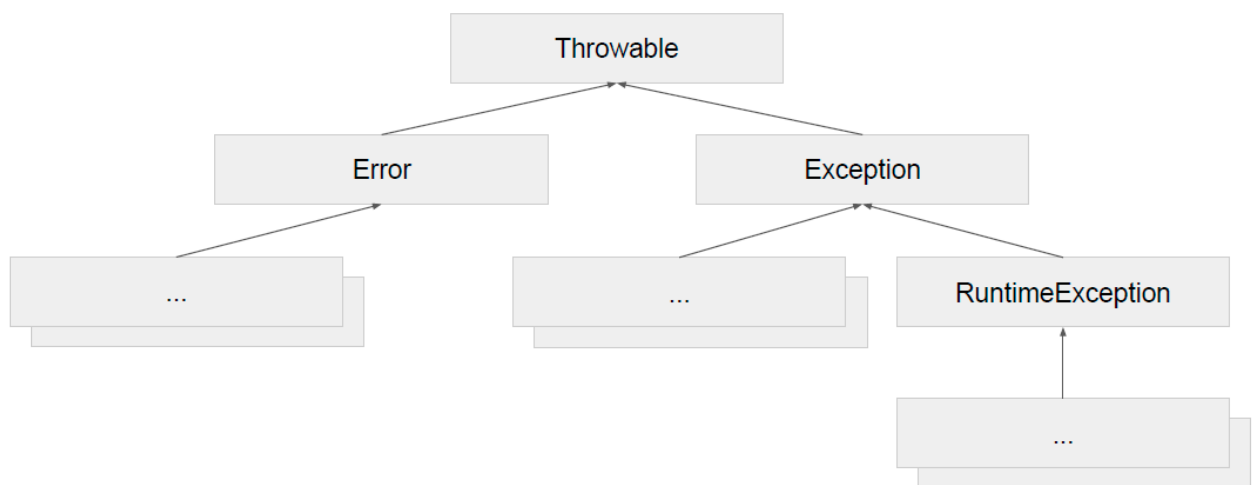
СОЗДАНИЕ ИСКЛЮЧЕНИЙ

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:

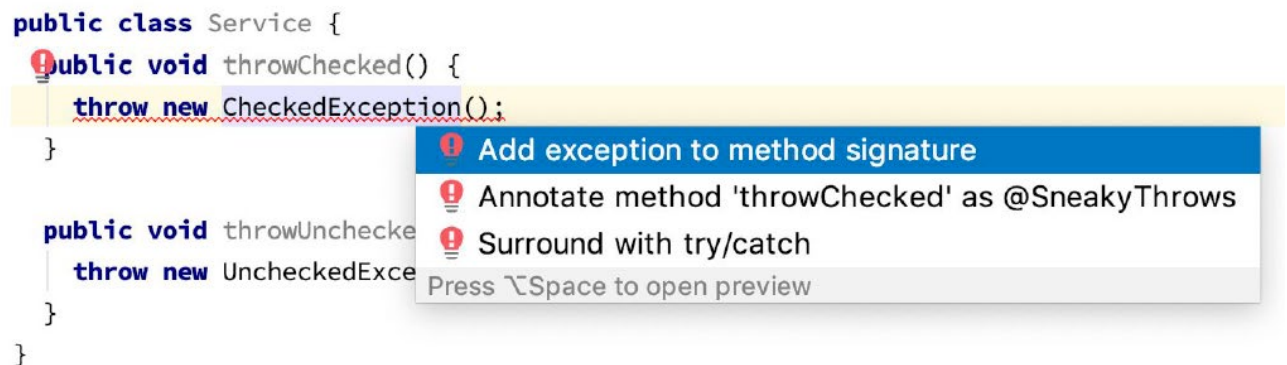


ИЕРАРХИЯ ИСКЛЮЧЕНИЙ

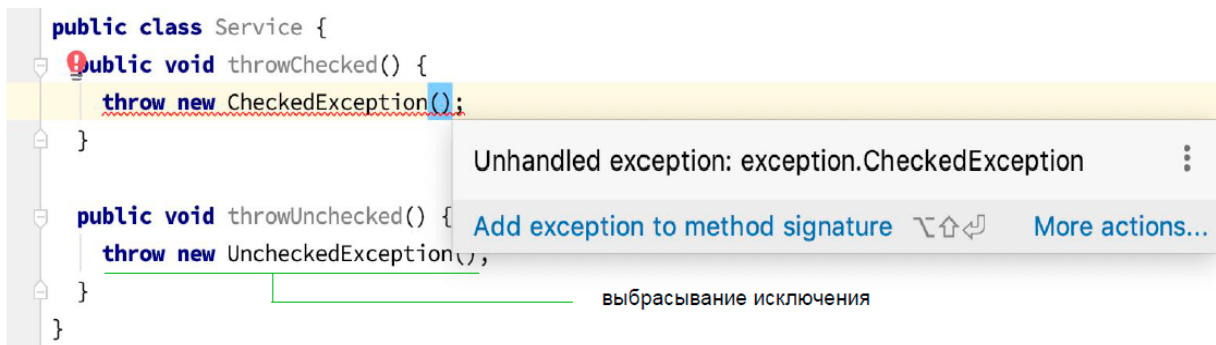
Как тестировщик, вы редко будете сами создавать собственные исключения, но общую идеологию знать должны:

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

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



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).

54

A: А когда стоит «выбрасывать» исключения?

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

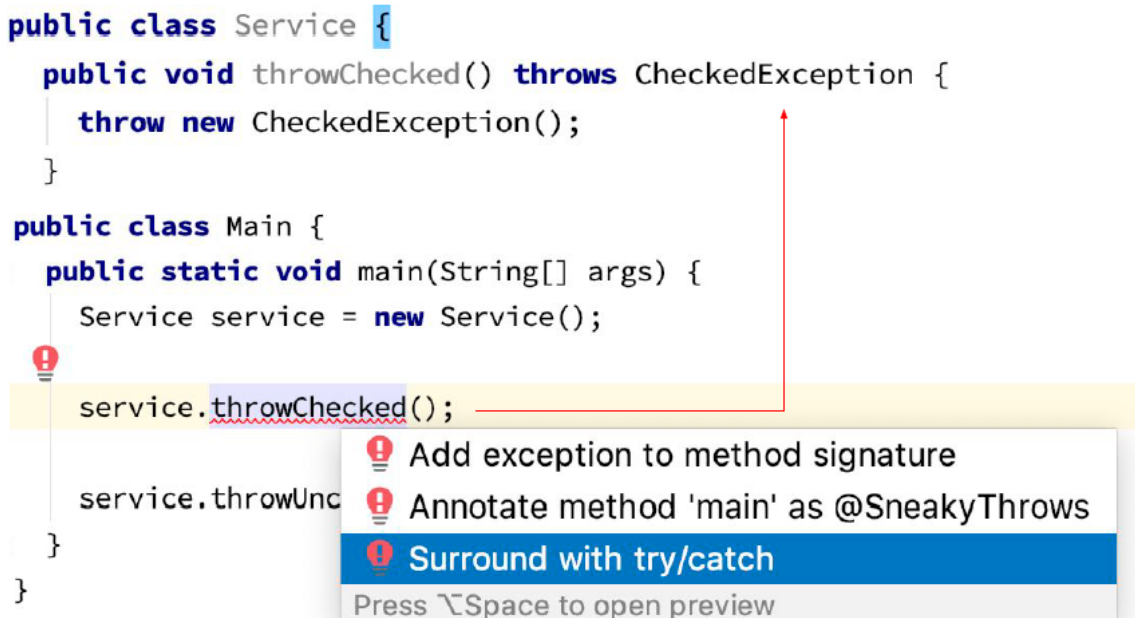
A: Можно ли выкидывать исключения в тестах?

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

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.throwUnchecked();
    }
}
```

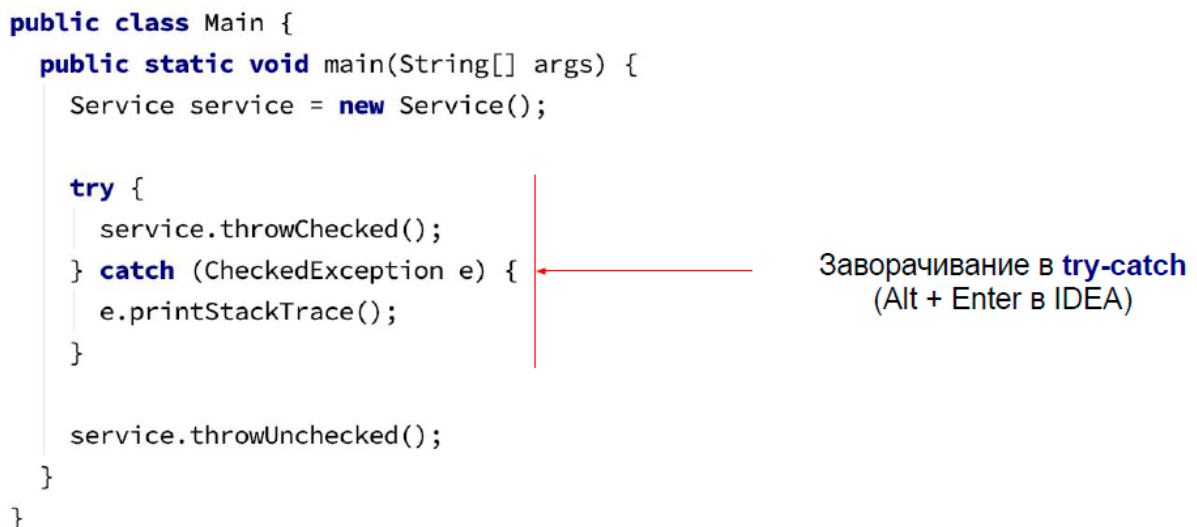


CHECKED & UNCHECKED EXCEPTIONS

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

        try {
            service.throwChecked();
        } catch (CheckedException e) {
            e.printStackTrace();
        }

        service.throwUnchecked();
    }
}
```



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

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.

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());
    }
    // lambda expression

    @Test
    public void shouldThrowUncheckedException() {
        assertThrows(UncheckedException.class, () -> service.throwUnchecked());
    }
    // lambda expression
}
```

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**.