Исключения
в Java

Учимся справляться с внештатными ситуациями

Всякое случается. То файл пропадает, то сервер падает.

Как нам это предусмотреть, и как научить программу с этим справляться?

Опасный код

Всегда что-то может пойти не так. Мы должны быть к этому готовы.
Как это контролировать?

Способ №1: использовать статус ошибки и блоки if:

```
FileManager f = new FileManager();
boolean opened = f.openFile();
if (opened) {
    int success = f.readFile(str)
    if (success) {
        int result = f.closeFile();
        if (result!=OK) {
            log("Не могу закрыть файл, ошибка");
        }
    } else {
        log("Не могу прочитать из файла");
    }
} else {
    log("Не могу открыть файл");
}
```

При таком подходе каждый метод должен вернуть сразу 2 результата:

1) результат выполнения
2) статус успешности операции

Средневековье: каждый сам должен думать, как защищаться, как тушить пожар, как позвать врача…

Логика приложения перемешана с логикой обработки внештатных ситуаций.

Безопасный код: используем исключения

Допустим методы FileManager могут возбуждать исключения:

```
class FileManager {
    public void openFile() throws FileNotFoundException { };
    public void readFile() throws IOException { };
    public void closeFile() throws FileCloseException { };
}
try {
    f.openFile();
    Sting str = f.readFile();
    f.closeFile();
} catch(FileNotFoundException e) {
    log("Не могу открыть файл");
} catch(IOException e) {
    log("Не могу прочитать из файла");
} catch(FileCloseException e)
    log("Не могу закрыть файл, ошибка");
}
```

безопасный код

обработчики исключений

Преимущества:

- можем позволить себе расслабиться и заняться основным делом - написанием кода, не думая об опасностях
- все опасные ситуации вынесены в отдельное место, и ими занимается отдельный код: *обработчик исключений*

Исключение – это класс

Допустим, раньше мы использовали if:

```
if (person != null) {
    person.sendMessage(message);
} else {
    log("Person not found!");
}
```

Создадим исключение **PersonNotFoundException**:

```
class PersonNotFoundException extends Exception {}
```

Мы можем начать его использовать так:

```
public Person findPerson(String name) {
    Person person = personCatalog.find(name);
    if (ivanov == null) {
        throw new PersonNotFoundException();
    }
}
```

И теперь код для работы с person будет выглядеть так:

```
try {
    Person person = findPerson("Ivanov");
    person.sendMessage(message);
} catch(PersonNotFoundException e) {
    log("Person not found!");
}
```

Добавим параметр в исключение

Будем сохранять имя человека, которого мы искали:

```
class PersonNotFoundException extends Exception {
        String name;

        public PersonNotFoundException(String name) {
                this.name = name;
        }

        public String getName() {
                return name;
        }
}
```

findPerson() перепишем так:

```
public Person findPerson(String name) {
        Person person = personCatalog.find(name);
        if (ivanov == null) {
                throw new PersonNotFoundException(name);
        }
}
```

И теперь код для работы с person будет выглядеть так:

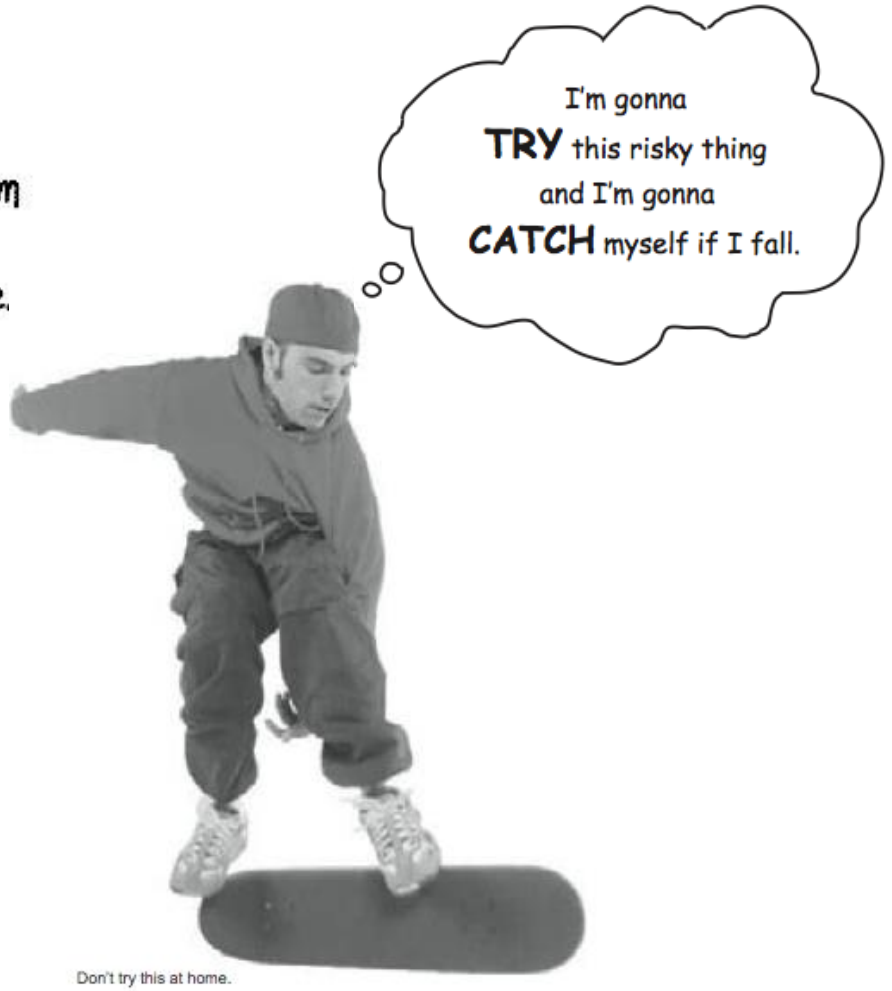
```
try {
        Person person = findPerson("Ivanov");
        person.sendMessage(message);
} catch(PersonNotFoundException e) {
        log("Person "+e.getName()+" not found!");
}
```

Exception – это объект… типа Exception

```
try {
    // Делаем опасные вещи
} catch(Exception ex) {
    // Пытаемся все исправить
}
```

То же самое, что объявление аргумента для метода.

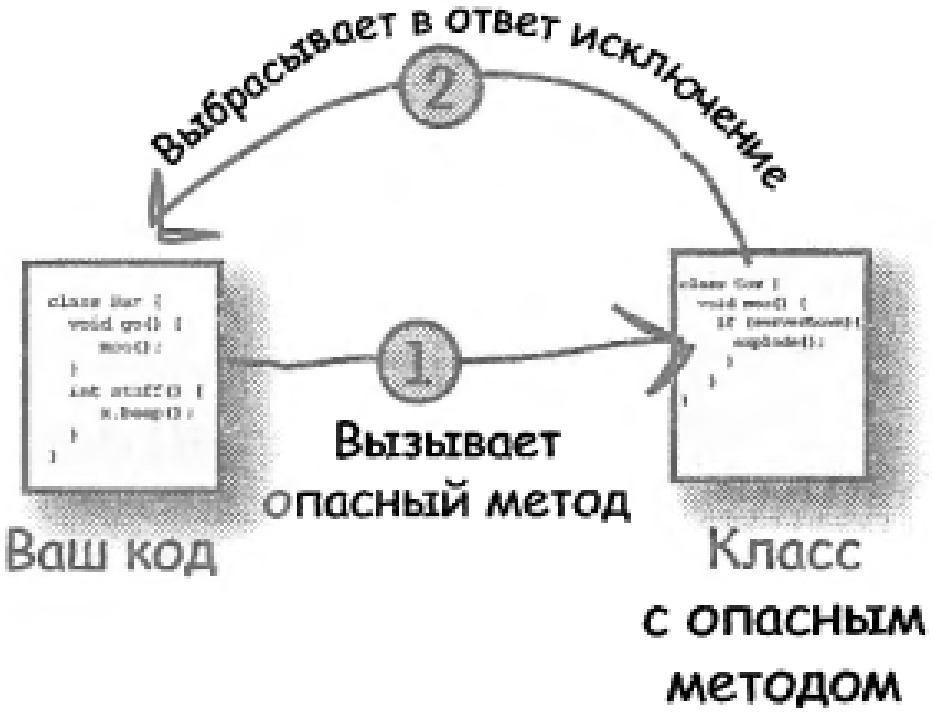
Этот код срабатывает только тогда, когда выброшено исключение.



Как все исправить?

- если сервер не отвечает, вы можете использовать блок catch для попытки связаться с другим сервером
- если файла не оказалось на месте, можно попросить пользователя помочь его найти

Если ваш код отлавливает исключение, то чей код его выбрасывает?



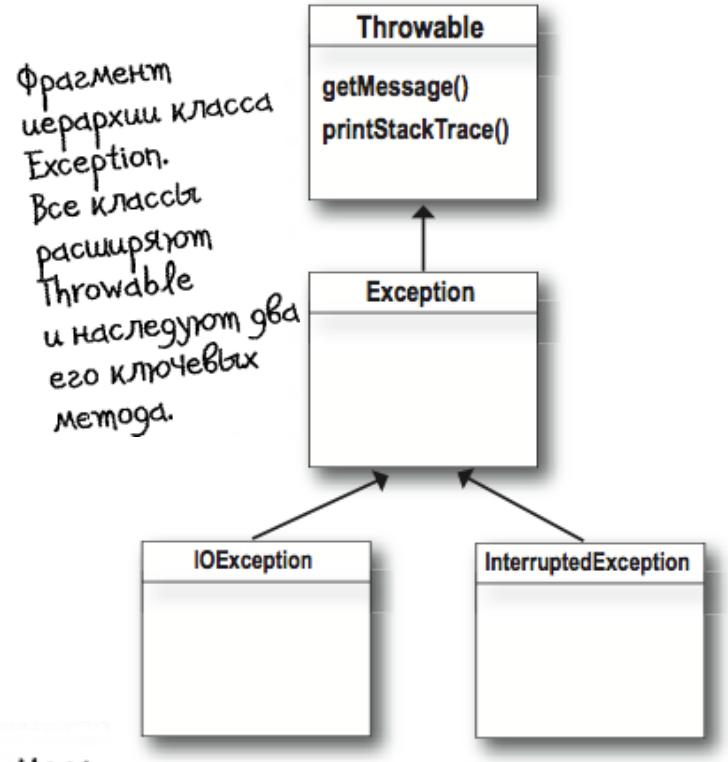
Если ваш код отлавливает исключение, то чей код его выбрасывает?

1 Опасный код, выбрасывающий исключение:

```
public void takeRisk() throws BadException {
    if (abandonAllHope) {
      throw new BadException();
    }
}
```

Этот метод должен сообщить всем (через объявление), что он выбрасывает BadException.

Создаем новый объект Exception и выбрасываем его.



Фрагмент иерархии класса Exception. Все классы расширяют Throwable и наследуют два его ключевых метода.

2 Ваш код, который вызывает опасный метод:

```
public void crossFingers() {
    try {
      anObject.takeRisk();
    } catch (BadException ex) {
      System.out.println("Aax!");
      ex.printStackTrace();
}
```

Если вы не можете справиться с исключением, по крайней мере получите трассировку стека с помощью метода printStackTrace(), который наследуют все потомки Exception.

Непроверяемые исключения: вспомните обо мне при случае

непроверяемые исключения – потомки RuntimeException

Создадим исключение: ошибка при сохранении документа

```
class SaveDocumentException extends RuntimeException {}

public void saveDoc() {
    try {
        document.writeOnDisk();
    } catch (Exception e) {
        throw new SaveDocumentException();
    }
}
```

вам не нужно писать throws SaveDocumentException

Вы можете вызвать saveDoc(), не обрабатывая исключение:

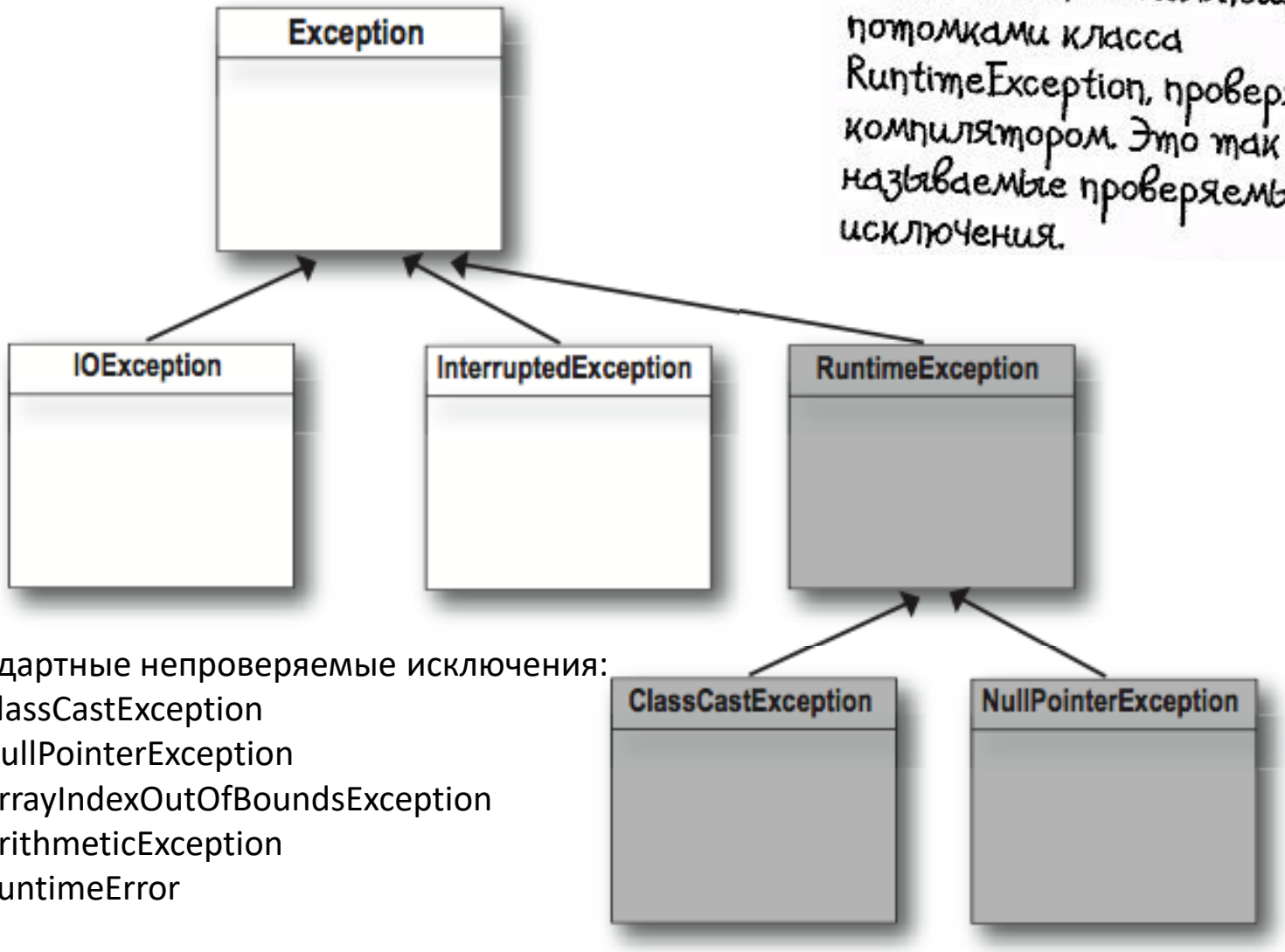
```
public void everyMinuteAutoSave() {
    saveDoc();
}
```

В то же время при желании вы можете поймать исключение и обработать его:

```
public void saveCommand() {
    try {
        saveDoc();
    } catch (SaveDocumentException e) {
        showDialog("Не удалось сохранить файл!");
    }
}
```

Компилятор проверяет все исключения, кроме RuntimeException.

Исключения, не являющиеся потомками класса RuntimeException, проверяются компилятором. Это так называемые проверяемые исключения.



Стандартные непроверяемые исключения:

- ClassCastException
- NullPointerException
- ArrayIndexOutOfBoundsException
- ArithmeticException
- RuntimeError

Стандартные непроверяемые исключения

ArrayIndexOutOfBoundsException

Выход за границы массива.

```
int[] array = new int[10];
array[20] = 0; // здесь будет брошен ArrayIndexOutOfBoundsException
```

Как предовратить? Проверяйте индекс перед использованием.

ArithmeticException

Ошибка целочисленной арифметики. Например, деление на ноль.

```
int a = 10 / 0; // здесь будет брошен ArithmeticException
```

Как предовратить? Проверяйте делитель на 0.

ClassCastException

Ошибка приведения типов.

```
int x = toInt("I am String!");
...
int toInt(Object a) {
    Integer x = (Integer) a; // будет брошен ClassCastException
    return x.intValue();
}
```

Как предовратить? Проверяйте на тип при приведении, используя **instanceof**:

```
if (a instanceof Integer) {
    Integer x = (Integer) a;
}
```

Стандартные непроверяемые исключения

NullPointerException

Ошибка обращения к полю или методу переменной, имеющей значение null.

```
Bar bar;
bar.foo(); // здесь будет брошен NullPointerException
```

Как предовратить? Проверяйте переменную на null перед использованием.

IllegalArgumentException

Стандартное исключение, возбуждаемое, если использовались некорректные аргументы.

```
public class MyBadCode {
  public static void main(String[] args) {
    Percentage percentage = new Percentage(121);
    System.out.println(percentage.getValue());
  }
}
```

```
Exception in thread "main"
java.lang.IllegalArgumentException: 121
  at Percentage.(Percentage.java:12)
  at MyBadCode.main(MyBadCode.java:3)
```

Код, возбуждающий IllegalArgumentException:

```
public Percentage(int value) {
  if (value < 0 || value > 100) {
    throw new IllegalArgumentException(Integer.toString(value));
  }
  this.value = value;
}
```

Стек вызовов и исключения

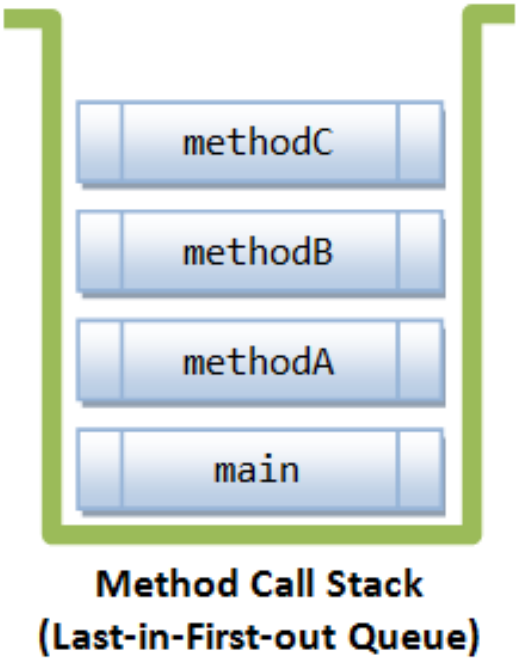
```
public class MethodCallStackDemo {
    public static void main(String[] args) {
        System.out.println("Enter main()");
        methodA();
        System.out.println("Exit main()");
    }

    public static void methodA() {
        System.out.println("Enter methodA()");
        methodB();
        System.out.println("Exit methodA()");
    }

    public static void methodB() {
        System.out.println("Enter methodB()");
        methodC();
        System.out.println("Exit methodB()");
    }

    public static void methodC() {
        System.out.println("Enter methodC()");
        System.out.println("Exit methodC()");
    }
}
```

```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exit methodC()
Exit methodB()
Exit methodA()
Exit main()
```



Method Call Stack
(Last-in-First-out Queue)

Стек вызовов и исключения

Допустим, в методе methodC() происходит ArithmeticException:

```
public static void methodC() {
    System.out.println("Enter methodC()");
    System.out.println(1 / 0);  // divide-by-0 triggers an ArithmeticException
    System.out.println("Exit methodC()");
}
```

Результат выполнения программы будет следующим:

```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at MethodCallStackDemo.methodC(MethodCallStackDemo.java:22)
        at MethodCallStackDemo.methodB(MethodCallStackDemo.java:16)
        at MethodCallStackDemo.methodA(MethodCallStackDemo.java:10)
        at MethodCallStackDemo.main(MethodCallStackDemo.java:4)
```

Это называется стек выполнения или call stack.
Именно он выводится с помощью метода exception.**printStackTrace()**

Управление программным потоком в try/catch

Если блок try успешно завершается

(doRiskyThing() не выбрасывает исключения)

```
1  Foo f = x.doRiskyThing();
   int b = f.getNum();

} catch (Exception ex) {
   System.out.println("failed");
}
2  System.out.println("We made it!");
```

Сначала выполняется блок try, потом запускается код, следующий за catch.

Код в блоке catch никогда не запускается.

```
File Edit Window Help RiskAll
%java Tester
We made it!
```

Блок try терпит неудачу

(так как doRiskyThing() выбрасывает исключение)

```
try {
1  Foo f = x.doRiskyThing();
   int b = f.getNum();

} catch (Exception ex) {
2  System.out.println("failed");
}
3  System.out.println("We made it!");
```

Запускается блок try, но вызов метода doRiskyThing() выбрасывает исключение, поэтому оставшаяся часть блока не выполняется.

Запускается блок catch, затем метод продолжает выполняться.

Оставшаяся часть блока try не выполняется, и это хорошо, так как она зависит от успешности вызова метода doRiskyThing().

```
File Edit Window Help RiskAll
%java Tester
failed
We made it!
```

Finally: для действий, которые нужно выполнить несмотря ни на что

Выключить плиту ***несмотря ни на что!***

```
try {
  turnOvenOn();
  x.bake();
} catch (BakingException ex) {
    ex.printStackTrace();
} finally {
    turnOvenOff();
}
```

блок try завершился неудачей?

прыгает блок catch {}

блок finally {}

блок try завершился успешно?

блок finally {}

блоки try или catch содержат return?

блок finally {} все равно выполнится!

Метод, который выбрасывает несколько исключений

```
public class Laundry {

    public void doLaundry() throws PantsException, LingerieException {
       // Код, который может выбросить оба исключения
    }
}
public class Foo {
    public void go() {
       Laundry laundry = new Laundry();
       try {
            laundry.doLaundry();
        } catch(PantsException pex) {
            // recovery code

        } catch(LingerieException lex) {
            // recovery code
        }
    }
}
```

В этом методе объявлено сразу 2 исключения.

Если doLaundry() выбросит исключение PantsException, то управление перейдет к блоку catch с PantsException.

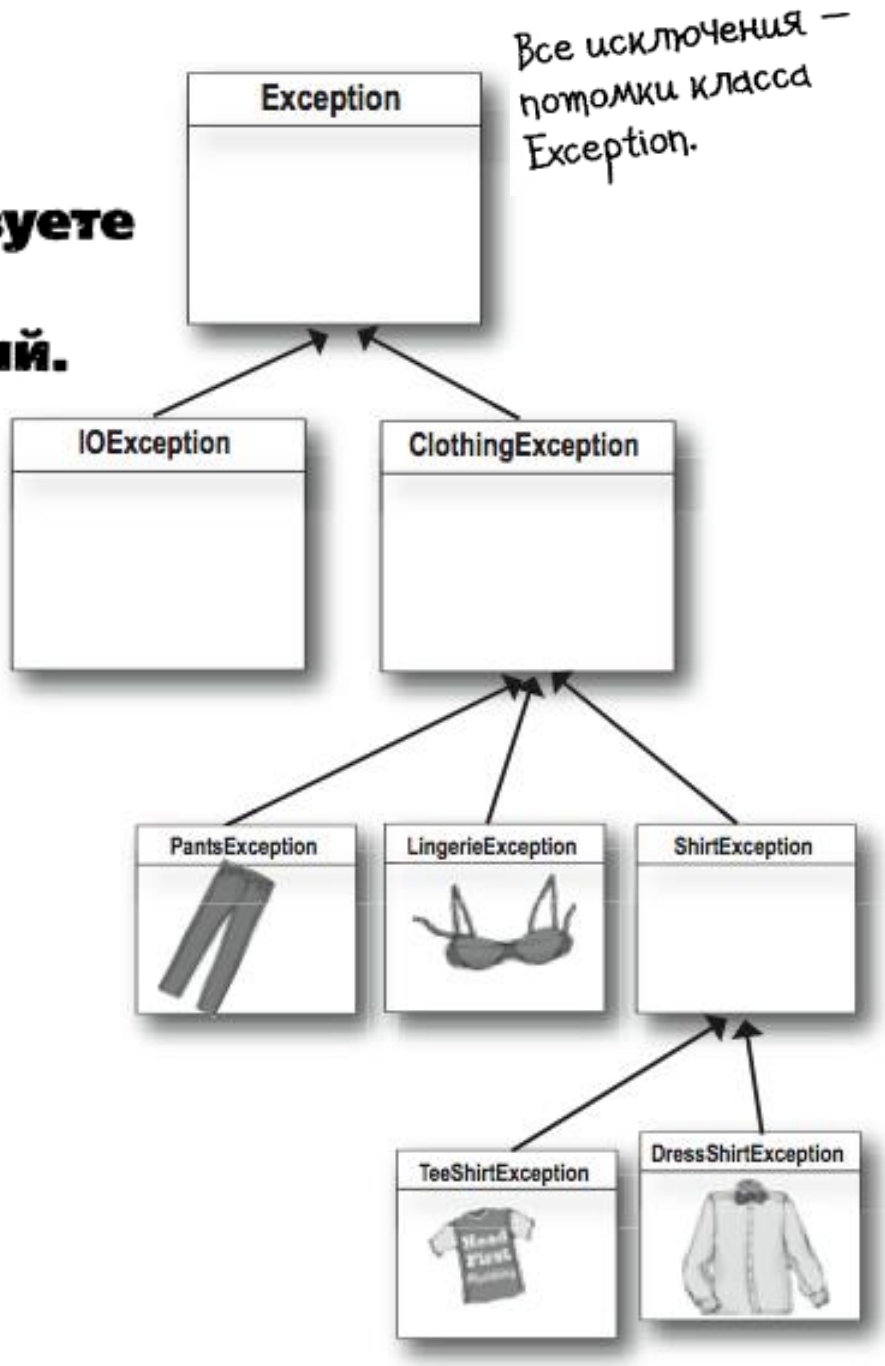
Если doLaundry() выбросит исключение LingerieException, то управление перейдет к блоку catch с LingerieException.

Исключения поддерживают полиморфизм

1. **При объявлении вы используете родительский тип для выбрасываемых исключений.**

```
public void doLaundry()
```

```
    throws ClothingException {
```



2 Вы можете перехватывать выбрасываемые исключения с помощью их родительского типа.

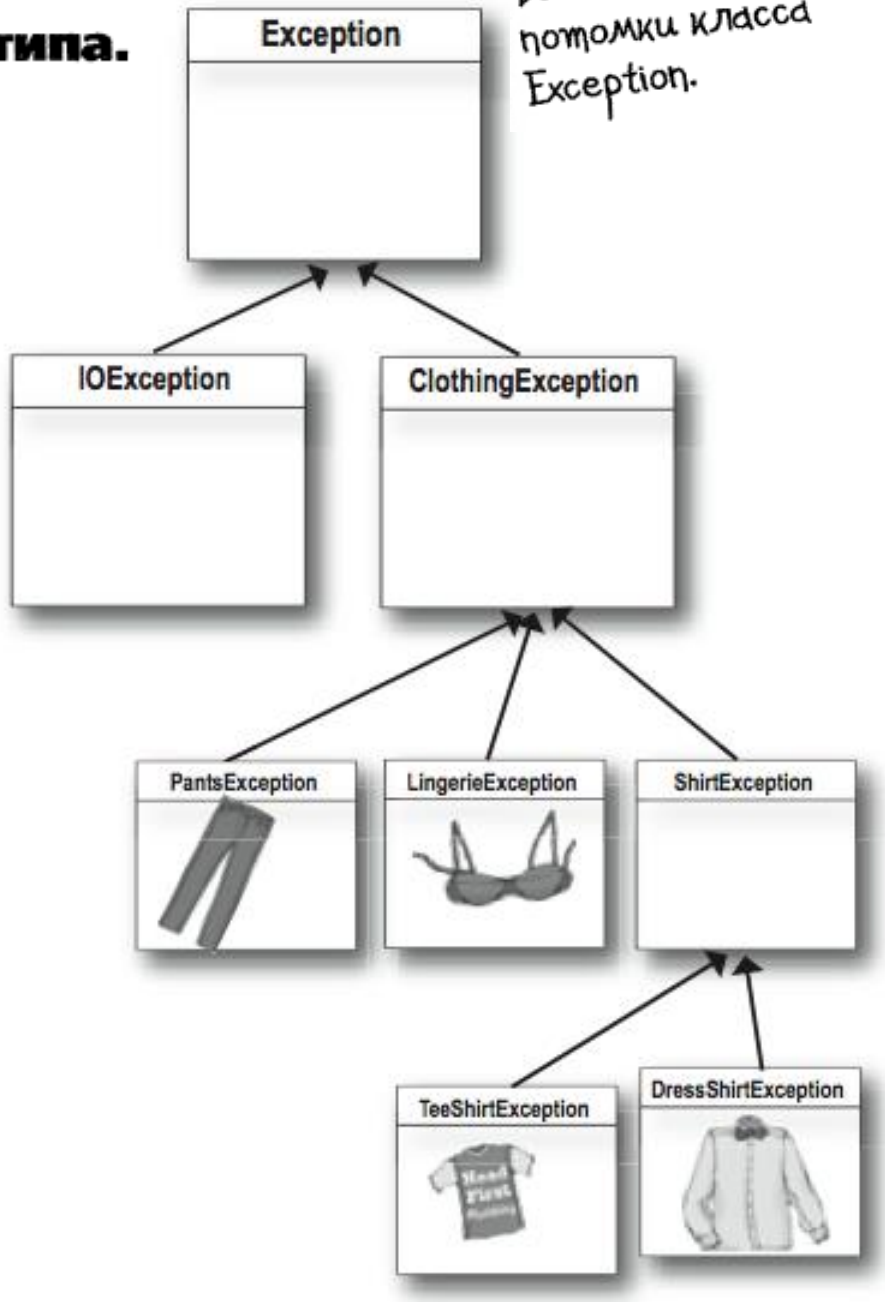
```
try {
    laundry.doLaundry();
} catch(ClothingException cex) {
    // Восстановительный код
}
```

Может поймать любой подтип ClothingException.

```
try {
    laundry.doLaundry();
} catch(ShirtException sex) {
    // Восстановительный код
}
```

Может поймать только TeeShirtException и DressShirtException.

Все исключения — потомки класса Exception.



Полиморфизм исключений

Создавайте отдельные блоки catch для каждого исключения, которое нужно обработать уникальным образом.

```
try {
    laundry.doLaundry();
} catch(Exception ex) {
    // recovery code...
}
```

Что он восстанавливает? Этот блок catch будет перехватывать все исключения, поэтому вы не сможете автоматически узнать, что именно пошло не так.

```
try {
   laundry.doLaundry();
} catch(TeeShirtException tex) {
    // Восстановление после TeeShirtException
} catch(LingerieException lex) {
    // Восстановление после LingerieException
} catch(ClothingException cex) {
    // Восстановление после all others
}
```

Исключениям TeeShirtException и LingerieException для обработки нужен свой код, поэтому вы должны использовать разные блоки catch.

Все остальные потомки ClothingException отлавливаются здесь.

Множественные блоки catch должны располагаться по возрастанию: от наименьшего к наибольшему

Сюда попадает исключение TeeShirtException, а остальные не помещаются.

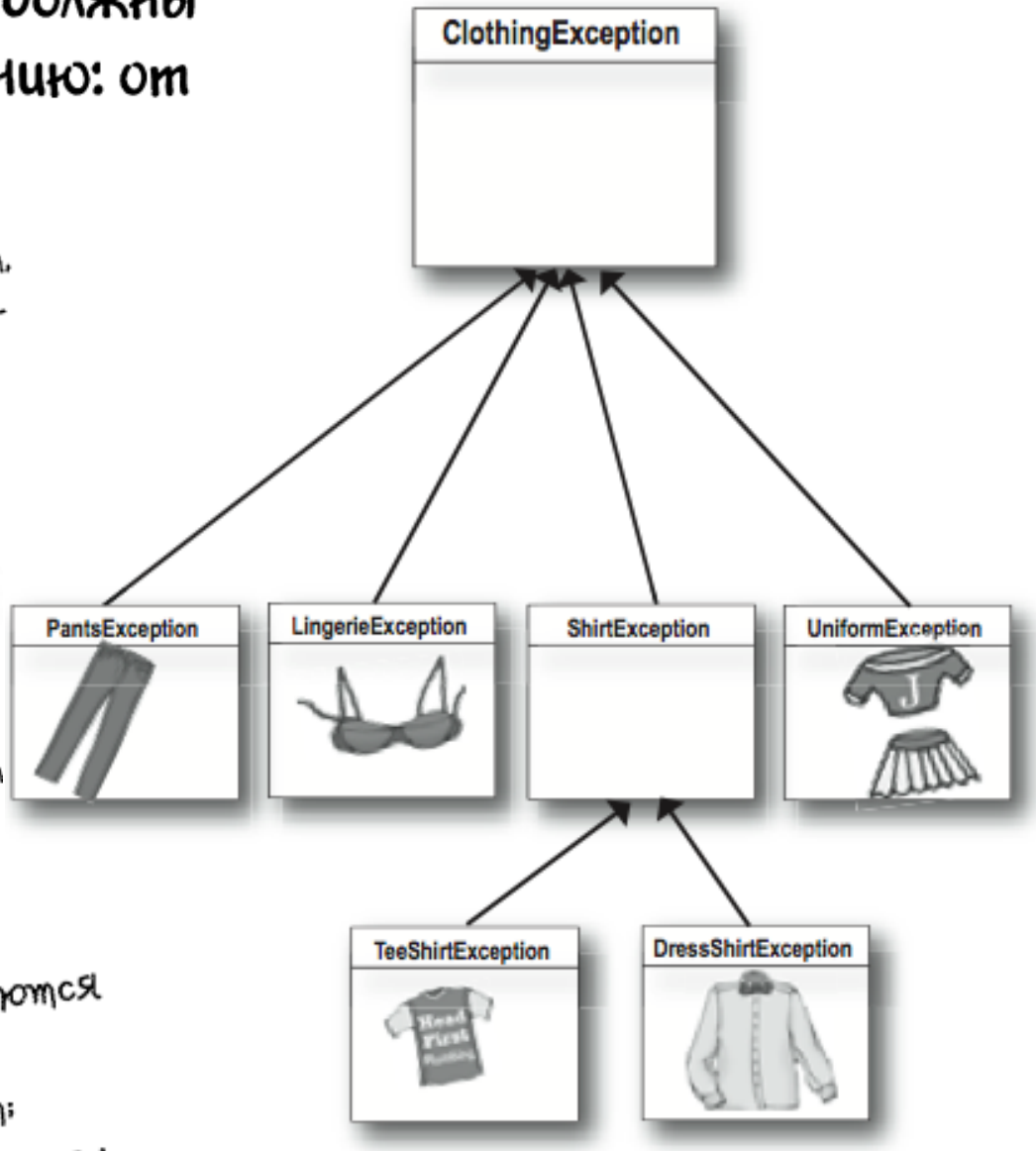
```
catch(TeeShirtException tex)
```

TeeShirtException никогда сюда не дойдет, в отличие от остальных потомков ShirtException

```
catch(ShirtException sex)
```

Здесь отлавливаются все производные ClothingException; ShirtException никогда не зайдет так далеко.

```
catch(ClothingException cex)
```



Нельзя размещать большие блоки catch над маленькими

Не делайте так!


```
try {
  laundry.doLaundry();
} catch(ClothingException cex) {
   // Восстановление после ClothingException
} catch(LingerieException lex) {
  // Восстановление после LingerieException
} catch(ShirtException sex) {
   // Восстановление после ShirtException
}
```

Блоки catch одного уровня могут
располагаться в любом порядке, так
как не способны перехватить чужое исключение.

Если вы не хотите обрабатывать исключение...

Просто пробросьте его дальше

Если вы не хотите обрабатывать исключение, то можете пробросить его на уровень ниже, используя объявление.

```
public void foo() throws ReallyBadException {
    // Вызов опасного метода без блоков try/catch
    laundry.doLaundry();
}
```

На самом деле это не вы его выбрасываете. У вас нет блока try/catch для опасного метода, который вы вызываете, поэтому вы сами становитесь «опасным методом». Теперь все, кто вызывает ваш метод, должны разбираться с этим исключением.

Пробрасывая (объявляя) исключение, вы только откладываете неизбежное.

Рано или поздно кто-то должен с этим разобраться. Но если и main() пробросит исключение?

```
public class Washer {
    Laundry laundry = new Laundry();

    public void foo() throws ClothingException {
        laundry.doLaundry();
    }

   public static void main (String[] args) throws ClothingException {
        Washer a = new Washer();
        a.foo();
    }
}
```

Оба метода пробрасывают исключение (объявляя его), поэтому его некому обрабатывать! Но этот код все равно скомпилируется.

Либо обработайте, либо объявите. Это закон.

Итак, вы уже узнали два способа удовлетворить компилятор при вызове опасного метода (выбрасывающего исключение).

① Обработка

Заключаем опасный метод в блок try/catch.

```
try {
    laundry.doLaundry();
} catch(ClothingException cex) {
      // Восстановительный код
}
```

Этот блок catch должен быть достаточно большим для обработки всех исключений, которые может выбросить метод doLaundry(). В противном случае компилятор будет жаловаться, что вы не перехватываете все исключения.

② Объявление (проброс)

Объявляем о том, что оба метода (наш и вызываемый) выбрасывают одни и те же исключения.

```
void foo() throws ClothingException {
    laundry.doLaundry();
}
```

Метод doLandry() выбрасывает ClothingException, но благодаря объявлению метод foo() пробрасывает это исключение. Никаких try/catch.

Правила (без) исключений

1 Вы не можете использовать блоки catch или finally без try.

```
void go() {
   Foo f = new Foo();
   f.foof();
   catch(FooException ex) { }
}
```

Так нельзя! Где try?

2 Вы не можете добавлять код между блоками try и catch.

```
try {
   x.doStuff();
}
int y = 43;
} catch(Exception ex) { }
```

Так нельзя! Вы не можете поместить код между try и catch.

3 За блоком try должны следовать catch или finally.

```
try {
   x.doStuff();
} finally {
   // Очистка
}
```

Несмотря на отсутствие catch, это допустимо, так как есть finally. Но вы не можете указывать блок try отдельно.

4 При использовании блока try только с finally (без catch) все равно нужно объявить исключение.

```
void go() throws FooException {
   try {
      x.doStuff();
   } finally { }
}
```

Блок try без catch не вписывается в закон «Обработка или объявление».

Работа с ресурсами: finally

```
InputStream input = null;

try{

  input = new FileInputStream(”file.txt");

  // do something with the stream

} catch(IOException e){ // first catch block
  throw new WrapperException(e);
} finally {
  try {
     if(input != null) input.close();
  } catch(IOException e) {  // second catch block
     throw new WrapperException(e);
  }
}
```

Работа с ресурсами: try-with-resources

```
private static void printFileJava7() throws IOException {

    try(FileInputStream input = new FileInputStream("file.txt")){

        int data = input.read();
        while(data != -1){
            System.out.print((char) data);
            data = input.read();
        }
    }
}
```

Работа с ресурсами: try-with-resources multiple

```
private static void printFileJava7() throws IOException {

    try(  FileInputStream input = new FileInputStream("file.txt");
          BufferedInputStream bufferedInput = new BufferedInputStream(input)
    ) {

        int data = bufferedInput.read();
        while(data != -1){
            System.out.print((char) data);
            data = bufferedInput.read();
        }
    }
}
```

Работа с ресурсами: AutoClosable

```
public interface AutoClosable {
    public void close() throws Exception;
}

public class MyAutoClosable implements AutoCloseable {
    public void doIt() {
        System.out.println("MyAutoClosable doing it!");
    }

    @Override
    public void close() throws Exception {
        System.out.println("MyAutoClosable closed!");
    }
}

private static void myAutoClosable() throws Exception {
    try(MyAutoClosable myAutoClosable = new MyAutoClosable()){
        myAutoClosable.doIt();
    }
}
```

```
MyAutoClosable doing it!
MyAutoClosable closed!
```

Ловим несколько исключений в Java 7+

До Java 7

```
try {
    // execute code that may throw 1
      // of the 3 exceptions below.

} catch(SQLException e) {
    logger.log(e);

} catch(IOException e) {
    logger.log(e);

} catch(Exception e) {
    logger.severe(e);
}
```

Java 7+

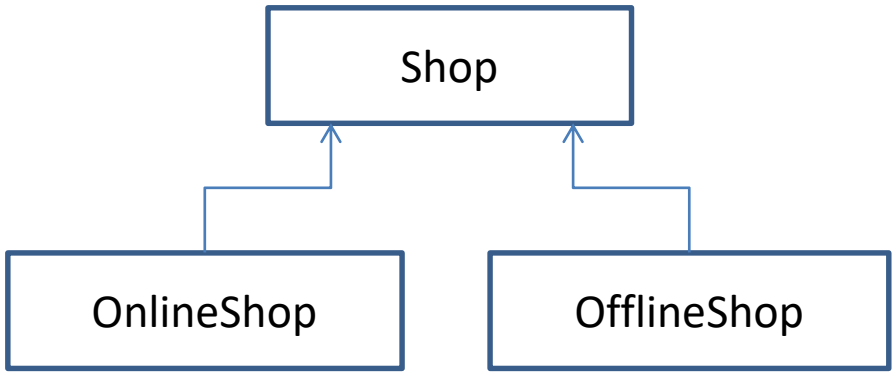
```
try {
   // execute code that may throw
   //1 of the 3 exceptions below.

} catch(SQLException |
            IOException e) {
    logger.log(e);

} catch(Exception e) {
    logger.severe(e);
}
```

Исключения и наследование

Унаследованный класс не может расширять список выбрасываемых методом исключений:




```
class Shop {
        public float getBalance();
        public static Shop createShop()
    {
                return new OnlineShop();
        }
}

class OnlineShop {
        public float getBalance()
                throws ServerException;
                // WRONG!
}
```

```
Shop shop = Shop.createShop(); // creating OnlineShop or OfflineShop
shop.getBalance(); // we rely on the safe code
```

But if shop is an instance of **OnlineShop**, then **getBalance()** is already dangerous.

Исключения и область try-catch

Что не так с этим кодом?
Assume that John Smith exists.

```
try {

        Client client1 = createClient(”John Smith”);
        Account account = client.createAccount();
        client.deposit(1000);
        client.withdraw(500);
    ----------------------------------------------
        Client client2 = createClient(”Jane Brown”);
        Account account = client.createAccount();
        client.deposit(1000);

} catch (ClientExistsException e) {
        e.printStackTrace();
}
```

Исключения и область try-catch

Assume that client has no money
Exception **NotEnoughFundsException** will be thrown.
What is wrong here?

```
try {
        Client client = createClient();
        Account account = client.createAccount();
        client.withdraw(10000);
        client.deposit(100000);
        client.withdraw(500);
} catch (NotEnoughFundsException e) {
        e.printStackTrace();
}
```

Исключения и область try-catch

What is wrong here?

```
try {
        Client client = createClient();
        Account account = client.createAccount();
        client.withdraw(10000);
        client.deposit(100000);
        client.withdraw(500);
} catch (NotEnoughFundsException e) {
        e.printStackTrace();
}
```

Remaining operations are not executed

```
try {
        client.withdraw(10000);
} catch (NotEnoughFundsException e) {
        e.printStackTrace();
}
try {
        client.deposit(1000);
} catch (NotEnoughFundsException e) {
        e.printStackTrace();
}
```

Logically related set of operations should be in its own try… catch block

Оборачивание проверяемых исключений в непроверяемые

Современные библиотеки позволяют оборачивать проверяемые исключения в непроверяемые.

Для чего?

Когда создавалась Java, считалось, что язык должен приучать программистов к **дисциплине программирования**. Но в результате код переполнен обработчиками исключений.

Оборачивание позволяет ловить исключения только там, где это необходимо.

Пример: создадим приложение для работы с БД персоналий.
У нас есть 3 уровня: доступа к БД, бизнес-логики и презентации.

Оборачивание проверяемых исключений в непроверяемые

```
class DatabaseException extends RuntimeException {
    private Exception rootException;
    private String info;
    DatabaseException(String info, Exception rootException) {
        this.rootException = rootException;
        this.info = info;
    }
}
```

Пусть мы используем только методы, возбуждающие DatabaseException:

```
class PersonDAO {
    public Person findByName(String name) throws DatabaseException {
        try {
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery(
                "SELECT xyz FROM abc WHERE inspectionTime = '"+name+"'")
            if (rs.next()) {
                Person person = new Person(name);
            }
        } catch (SQLException e) {
            throw new DatabaseException("exception in findByName", e);
        }
    }
    public void savePerson(Person person) throws DatabaseException {}
    public void deletePerson(Person person) throws DatabaseException {}
}
```

Оборачивание проверяемых исключений в непроверяемые

Создадим метод, который находит человека по имени и удаляет его:

```
class PersonManager {
    public void rename(String name, String newName) {
        PersonDAO dao = new PersonDAO();
        Person person = dao.findByName(name);
        person.setName(newName);
        dao.savePerson(person)
    }
}
```

В этом методе мы можем вообще не задумываться, как обработать исключение.
И, наконец, мы можем поймать исключение там, где это нужно:

```
class RenameGUI {
    public void renamePerson(String name) {
        String newName = showDialog("Введите новое имя для "+name+":");
        try {
            personManager.rename(name, newName);
        } catch(DatabaseException e) {
            showDialog("Переименование выполнить не удалось");
        }
    }
```