

Лекция 6.

Исключения

Исключительные ситуации

- При исполнении кода могут возникать такие ситуации, когда исполнение нашего кода не пойдет по предполагаемому сценарию
- Например, при попытке прочитать данные из файла, во время исполнения программы, обнаружим что файла не существует
- Или, при чтении данных из сети, обрывается соединение
- Или кончается свободное место на диске
- Такие ситуации называют **исключительными ситуациями** (или просто **исключениями**)

Реакция на исключения

- По умолчанию, при возникновении исключения, программа падает
- Но можно зарегистрировать свой код-перехватчик исключений, и тогда программа падать не будет

Как реагировать на исключения

- На исключительные ситуации можно реагировать разным образом:
 - Можно завершить программу, выдав сообщение об ошибке
 - Можно просто продолжить исполнение со следующей команды
 - Можно попытаться выполнить действие ещё раз
 - И т.д.
- В каждом конкретном случае реагировать на исключительные ситуации нужно по-своему

Исключения

- В Java и многих других языках, реализован механизм обработки исключительных ситуаций
- Для этого используется конструкция `try-catch-finally`
- А **бросить исключение** (т.е. инициировать его) можно при помощи оператора `throw`
- <http://docs.oracle.com/javase/tutorial/essential/exceptions/index.html>

Терминология

- Информация об исключительной ситуации передается при помощи специальных классов-**исключений (exceptions)**
- Они содержат в себе сообщение об ошибке и стек вызовов, который очень полезен при отладке
- Если в функции может возникнуть исключительная ситуация, то говорят, что функция **бросает исключение (throws exception)**
- **Поймать исключение** – сделать так, чтобы выполнялся ваш обработчик исключения
- **Обработать исключение** – обработчик исключения успешно отработал до конца, не вызвав новое исключение

Выброс исключения

- ```
public static void f() {
 // этот код выполнится
 throw new RuntimeException("Message");
 // этот код никогда не выполнится
}
```
- Можно самим выбросить исключение, например, если мы сами проверили аргументы метода, и видим, что передали недопустимое значение
- Для этого нужно бросить экземпляр исключения при помощи слова **throw** (с англ. - бросить)
- В данном случае мы создаем новый объект исключения **RuntimeException**

# Что происходит при бросании исключения

1. Делается проверка, находимся ли мы внутри конструкции `try-catch`, при этом там есть `catch`, который соответствует типу нашего исключения
  2. Если да, то попадаем в `catch`. Распространение исключения прекращается
  3. Если нет, то метод немедленно завершается, мы вместе с брошенным исключением попадаем в то место, откуда метод был вызван
  4. В этом месте выполняются действия из п.1 и т.д.
- Так исключение может дойти до **main**. Если `try-catch` нет и там, то программа падает и завершается



# Раскрутка стека вызовов

- При возникновении исключения происходит **раскрутка стека вызовов**
- Т.е. стек «**разматывается**» в обратном порядке, исполнение возвращается назад, в вызвавшую функцию и т.д.
- ```
public static void f() {  
    // ЭТОТ КОД ВЫПОЛНИТСЯ  
    throw new RuntimeException("Message");  
    // ЭТОТ КОД НИКОГДА НЕ ВЫПОЛНИТСЯ  
}  
  
public static void main(String[] args) {  
    f();  
    System.out.println("OK");  
}
```

Проверка входных данных

- Часто нужно кидать исключения при проверке входных данных – в конструкторах, сеттерах и публичных методах
- Нужно проверять все аргументы на корректность, если они некорректные, то кидать подходящее исключение с понятным сообщением

Проверка входных данных

```
public class Person {  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        if (age <= 0) {  
            throw new IllegalArgumentException("age must be > 0");  
        }  
        if (name == null) {  
            throw new IllegalArgumentException("name can't be null");  
        }  
        this.name = name;  
        this.age = age;  
    }  
}
```

Обработка исключения

- Рассмотрим вариант, когда мы хотим обработать исключение
- Есть два блока – `try` и `catch`
- `try` {
 // тут идет код для благополучного хода событий
 // например, когда файл есть, нет никаких ошибок
} `catch` (`IOException` e) {
 // этот код выполнится, при возникновении исключения
 // `IOException` в любой из команд, которая выполняется внутри
 // блока `try`
}
- Блок `catch` ловит исключение указанного типа. Если исключение поймано, то оно считается обработанным и не распространяется дальше по стеку вызовов

Вариант – исключения не было

- `try {`
 // тут идет код для благополучного хода событий
 // например, когда файл есть, нет никаких ошибок
`} catch (IOException e) {`
 // этот код выполнится, при возникновении исключения
 // IOException в любой из команд, которая выполняется внутри
 // блока `try`
`}`
- Просто выполняется блок `try`, а затем выполняется код, который идет ниже, чем `try-catch`

Вариант – произошло IOException

- ```
try {
 // тут идет код для благополучного хода событий
 // например, когда файл есть, нет никаких ошибок
} catch (IOException e) {
 // этот код выполнится, при возникновении исключения
 // IOException в любой из команд, которая выполняется внутри
 // блока try
}
```
- Выполняется блок `try`, в какой-то его команде возникает исключение `IOException`
- Тогда произойдет размотка стека до этой команды. Остальные команды блока `try` не выполнятся
- Исполнение переходит в блок `catch`, он исполняется. Если в нем возникло новое исключение, то тогда стек разматывается дальше
- Если `catch` успешно отработал, то исполняется код после `try-catch`

# Вариант – произошло другое искл-е

- `try {`  
    // тут идет код для благополучного хода событий  
    // например, когда файл есть, нет никаких ошибок  
`} catch (IOException e) {`  
    // этот код выполнится, при возникновении исключения  
    // IOException в любой из команд, которая выполняется внутри  
    // блока `try`  
`}`
- Блок `catch` не поймает исключение, поэтому стек вызовов будет разматываться дальше, туда, откуда вызван метод, содержащий этот `try-catch`

# Несколько блоков catch

- ```
try {  
    // код  
} catch (FileNotFoundException e) {  
    // обработка исключения FileNotFoundException  
} catch (IOException e) {  
    // обработка исключения IOException  
}
```
- Блоков `catch` может быть много
- Классы исключений могут наследоваться друг от друга, например, `FileNotFoundException` наследуется от `IOException`
- И вот тут интересный момент – как выбирается нужный блок `catch` для выброшенного исключения

Несколько блоков catch

- ```
try {
 // код
} catch (FileNotFoundException e) {
 // обработка исключения FileNotFoundException
} catch (IOException e) {
 // обработка исключения IOException
}
```
- Допустим, брошено исключение
- Смотрятся блоки `catch` в порядке их объявления
- Если указанный тип исключения совместим с типом брошенного исключения, т.е. проверка `instanceof` дает `true`, то выбирается этот блок `catch`
- Компилятор следит чтобы `catch` с наследником шел раньше

# Несколько блоков catch с Java 1.7

- ```
try {  
    // код  
} catch (SQLException | IOException e) {  
    // обработка исключений этих типов  
}
```
- Если код обработчиков одинаковый, то с версии Java 1.7 можно использовать следующий синтаксис
- Имена типов исключений должны разделяться вертикальной чертой |

Повторный выброс исключения

- ```
try {
 // код
} catch (IOException e) {
 System.out.println("Error occurred: " + e.getMessage());
 e.printStackTrace(); // печатает стек вызовов
 throw e; // обратим внимание, что бросаем тот же
 // объект исключения
}
```
- Поймав исключение блоком `catch`, его можно выбросить повторно
- Это полезно, если мы при помощи блока `catch` хотим либо частично обработать исключение, либо просто записать в лог информацию о том, что произошла ошибка

# Стек вызовов

```
java.lang.Exception: text
 at ebt.s7.vm.Main.f(Main.java:16)
 at ebt.s7.vm.Main.main(Main.java:21)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
 at java.lang.reflect.Method.invoke(Method.java:483)
 at com.intellij.rt.execution.application.AppMain.main(AppMain.java:134)
```

e.printStackTrace();

# Блок finally

- ```
try {  
    // код  
} catch (IOException e) {  
    // обработка исключения IOException  
} finally {  
    // обычно, освобождение ресурсов  
}
```
- **finally** – это блок кода, который выполняется всегда после блока **try** или блока **catch**
- Обычно он используется для освобождения ресурсов
- Например, мы хотим поработать с файлом, после работы с ним, нужно его закрыть в любом случае – если все прошло хорошо, и если произошло исключение

Блок `finally` – нет исключения

- ```
try {
 // код
} catch (IOException e) {
 // обработка исключения IOException
} finally {
 // обычно, освобождение ресурсов
}
```
- Сначала выполняется блок `try`, потом – блок `finally`
- Причем блок `finally` исполнится даже если внутри `try` будет `return`, `break` или `continue`

# Блок `finally` – брошен `IOException`

- ```
try {  
    // код  
} catch (IOException e) {  
    // обработка исключения IOException  
} finally {  
    // обычно, освобождение ресурсов  
}
```
- Частично выполняется блок `try`, потом `catch`, потом блок `finally`
- Причем блок `finally` исполнится даже если внутри `catch` будет `return`, `break` или `continue`, или будет выброшено новое исключение

Блок `finally` – брошен не `IOException`

- ```
try {
 // код
} catch (IOException e) {
 // обработка исключения IOException
} finally {
 // обычно, освобождение ресурсов
}
```
- Частично выполняется блок `try`, потом блок `finally`, потом исключение распространяется дальше по стеку вызовов
- В общем, блок `finally` выполняется реально **всегда**
- И это как раз очень полезно для освобождения ресурсов – то что мы не забудем их освободить



# Return из finally

- ```
public static int f() {  
    try {  
        // код  
        return 1;  
    } catch (IOException e) {  
        // обработка исключения IOException  
    } finally {  
        return 2;  
    }  
}
```
- Если в **finally** стоит **return** , то его значение «перебивает» все остальные **return** в функции
- То есть результатом вызова функции будет 2
- Этот вопрос могут спросить на собеседовании

Блока catch может не быть

- ```
try {
 // код
} finally {
 // обычно, освобождение ресурсов
}
```
- Блока `catch` может не быть, но тогда должен быть `finally`
- Этот вариант полезен, если при возникновении исключения мы хотим выполнить код, обычно очистку ресурсов, но само исключение ловить и обрабатывать не хотим

# Обработка исключения до Java 1.7

- Рассмотрим вариант, когда мы хотим обработать исключение

```
public static void main(String[] args) {
 FileReader reader = null;
 try {
 reader = new FileReader("input.txt");
 // ... много строк кода, чтение из файла
 } catch (IOException e) {
 System.out.println("Ошибка записи в файл: " + e.getMessage());
 } finally {
 if (reader != null) {
 try {
 reader.close();
 } catch (IOException e) {}
 }
 }
}
```

Приходится объявлять reader  
снаружи try-catch, чтобы  
переменная была видна во всех  
этих блоках

Проверка на null нужна, потому что  
если исключение будет в  
конструкторе, то reader останется null

Сам метод close бросает исключение,  
приходится его ловить

# Обработка исключения с Java 1.7

- ```
public static void main(String[] args) {  
    try (FileReader reader = new FileReader("input.txt")) {  
        // ... много строк кода, чтение из файла  
    }  
}
```
- Эта конструкция автоматически вызывает метод `close` своего аргумента после завершения блока любым образом
- По сути, это и есть реализация, которая указана слайдом ранее
- Все классы, которые реализуют интерфейс `java.lang.AutoCloseable` (в том числе, кто реализует `java.io.Closeable`), могут использовать ЭТОТ синтаксис

Обработка исключения с Java 1.7

- ```
public static void main(String[] args) {
 try (
 PrintWriter writer1 = new PrintWriter("output1.txt");
 PrintWriter writer2 = new PrintWriter("output2.txt")
) {
 // ... много строк кода, запись в файл
 }
}
```
- В одном блоке `try` можно иметь несколько ресурсов
- Они отделяются точкой с запятой
- Методы `close` потом вызываются в порядке создания ресурсов

# Обработка исключения с Java 1.7


- ```
public static void main(String[] args) {  
    try (PrintWriter writer = new PrintWriter("output.txt")) {  
        // ... много строк кода, запись в файл  
    } catch (IOException e) {  
        // обработка исключения  
    } finally {  
        // дополнительный блок finally  
    }  
}
```
- Можно добавлять `catch` и `finally`
- Они будут вызываться уже после освобождения ресурсов

Подавленные исключения

- ```
public static void main(String[] args) {
 try (PrintWriter writer = new PrintWriter("output.txt")) {
 // ... много строк кода, запись в файл
 } catch (IOException e) {
 // обработка исключения
 }
}
```
- Исключение может возникнуть в блоке try, либо при попытке закрыть ресурс
- Если исключение в блоке try, то будет выброшено именно оно
- Если выброшено только исключение при освобождении ресурса, то будет выброшено именно оно
- Если выбросятся оба исключения, то выбросится исключение из try, но можно получить доступ и к подавленному исключению при помощи статического метода `Throwable.getSuppressed()`

# Проверяемые исключения

- Пусть, у нас есть код записи в файл. Здесь может возникать ошибка, например, когда нет прав
- Хотим написать следующий код, но будет ошибка компиляции:
- ```
public static void main(String[] args) {  
    PrintWriter writer = new PrintWriter("output.txt");  
    // ... много строк кода, запись в файл  
    writer.close();  
}
```
- В Java некоторые виды исключений нужно обязательно либо обработать, либо явно указать, что метод выбрасывает исключение (т.е. что мы не хотим обрабатывать исключение)
- Такие исключения называют **проверяемыми (checked)**, т.к. компилятор проверяет, что мы сделали действия из предыдущего абзаца



Компилятор будет ругаться,
что не обработано
возможное исключение

Спецификация исключения

- Рассмотрим вариант, когда мы хотим сказать, что метод должен бросить исключение

- ```
public static void main(String[] args) throws IOException {
 PrintWriter writer = new PrintWriter("output.txt");
 // ... много строк кода, запись в файл
 writer.close();
}
```

IOException – стандартный  
класс исключений  
ввода/вывода

- К сигнатуре метода добавляется конструкция **throws**:  
**throws Exception1, Exception2**

Классы исключений, которые может  
выбросить метод

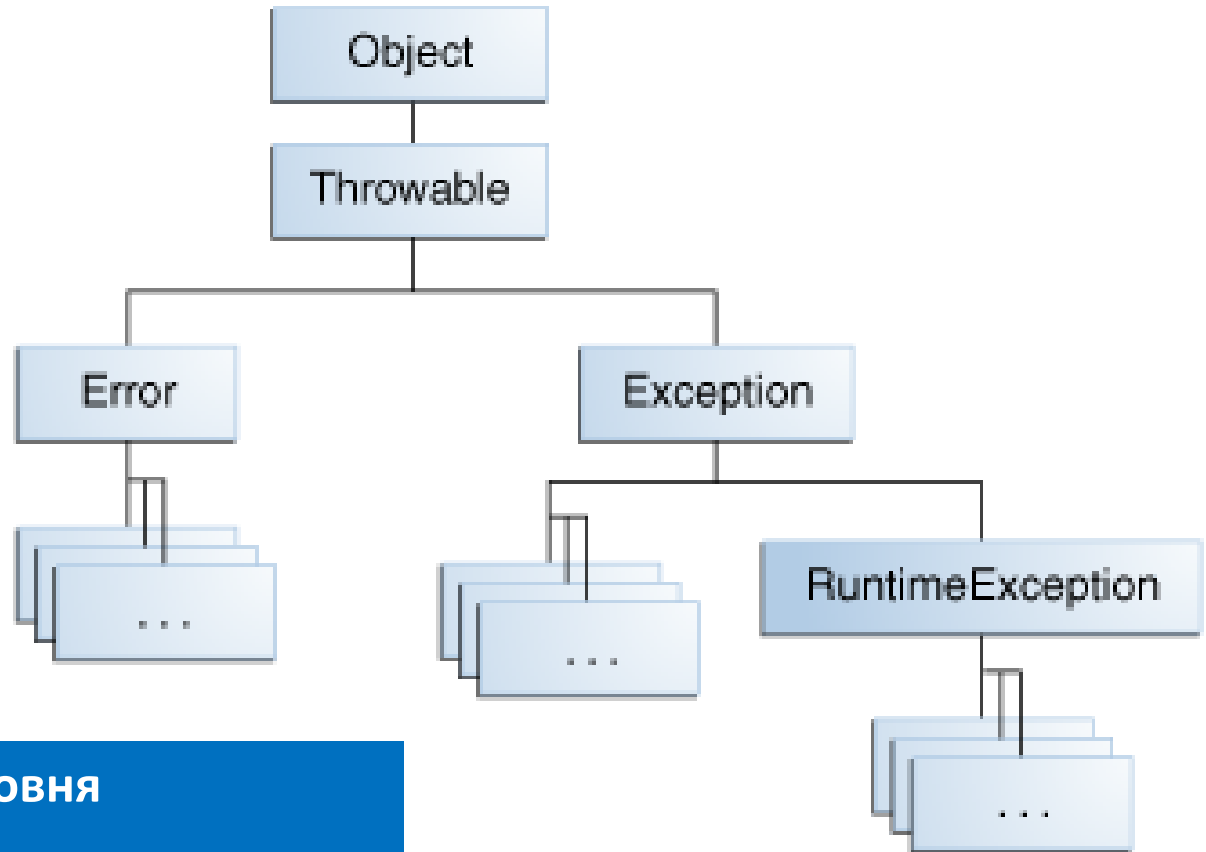
- Теперь компилятор тоже не ругается

# Иерархия исключений

1. Все, что можно бросить как исключение, наследуется от класса Throwable

2. От Throwable наследуются Error и Exception

3. Error – это ошибки уровня виртуальной машины. Например, нехватка памяти. Их не обязательно перехватывать, в отличие, например, от IOException

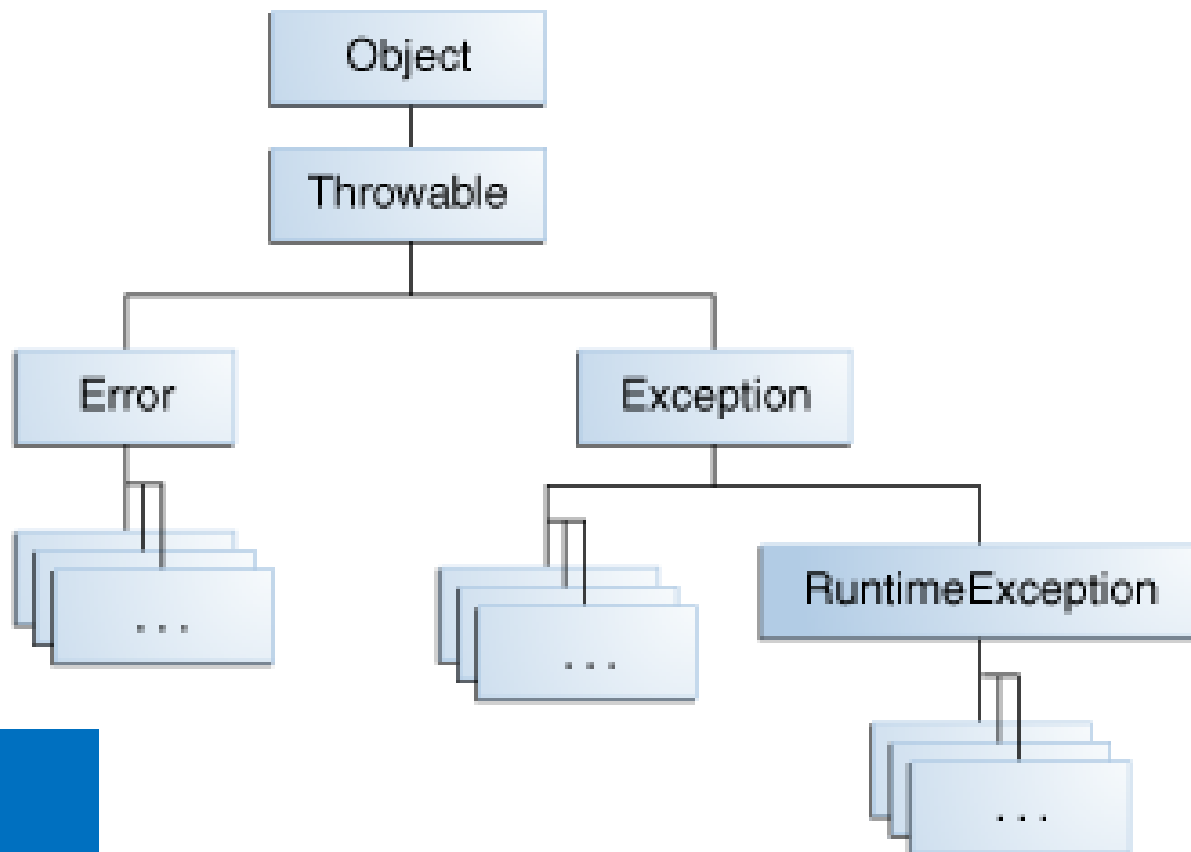


# Иерархия исключений

1. Exception – это исключения, как мы их рассматривали

2. Компилятор следит, чтобы мы их обязательно обрабатывали, либо указывали в сигнатуре методов

3. Есть специальный подкласс исключений - RuntimeException



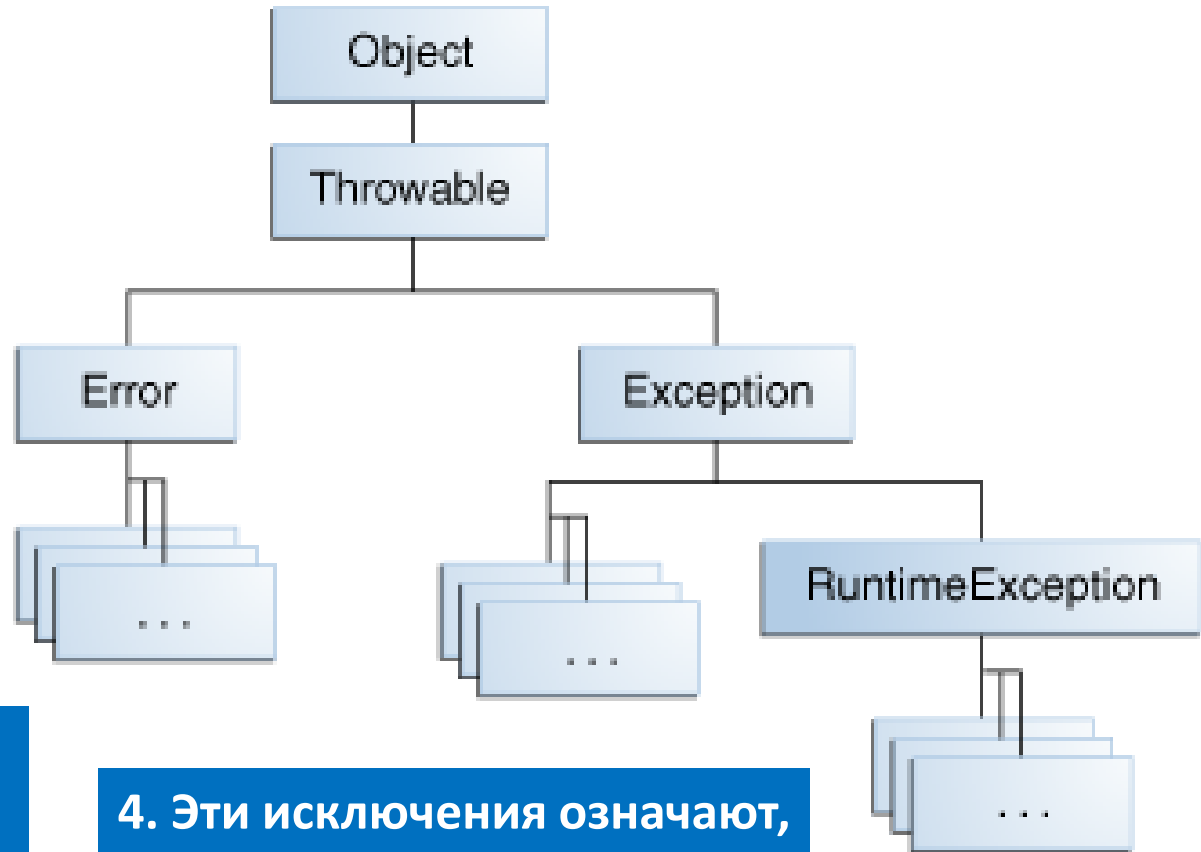
# Иерархия исключений

1. RuntimeExceptions –  
исключения времени  
исполнения

2. Примеры:  
NullPointerException –  
попытка вызвать  
метод у null

3. Эти исключения не  
обязательны для  
обработки, обычно их  
даже не надо  
обрабатывать

4. Эти исключения означают,  
что мы ошиблись в коде.  
Например, забыли  
проверить на null



# Как поймать все исключения

- ```
try {  
    // код  
} catch (Throwable e) {  
    // обработка исключения  
}
```
- Такой блок catch будет ловить все, что можно бросить, в том числе **Error**'ы
- Делать так не рекомендуется, потому что не надо перехватывать то, что мы не можем правильно обработать

Создание своих исключений

- Лучше создавать свои исключения если:
 - В Java нет такого стандартного исключения
 - Свой тип исключения поможет отличить их от других исключений
- При создании своих классов исключений рекомендуется заканчивать их имена на Exception
- Пример:
- ```
public class MyException extends Exception {
 public MyException(String message) {
 super(message);
 }
}
```

# Альтернатива исключениям

- Возможная альтернатива исключениям – возвращение специальных значений из метода
- Например, в некоторых случаях, если произошла ошибка, можно возвращать из метода `null`
- Либо, если метод должен возвращать неотрицательное целое число, можно возвращать отрицательные числа в качестве кодов ошибок, а затем сравнивать результат с этими кодами
- Достоинством данного подхода является более высокая производительность
- Но читаемость кода при этом существенно страдает

# Язык С – работа с файлами

```
FILE* file = fopen("input.txt", "r"); // открытие файла для чтения
int a;
float b;
if (file != NULL) { // проверка что нет ошибки
 fscanf(file, "%d", &a); // считываем из файла целое число в переменную a
 if (ferror(file)) {
 printf("Ошибка при чтении файла");
 fclose(file);
 } else {
 fscanf(file, "%f", &b); // считываем вещ. число в переменную b
 if (ferror(file)) {
 printf("Ошибка при чтении файла");
 fclose(file);
 }
 }
} else {
 printf("Ошибка при чтении файла");
}
```



# Язык С – если бы были исключения

```
try (FILE* file = fopen("input.txt", "r")) {
 int a;
 float b;
 fscanf(file, "%d", &a); // считываем из файла целое число в переменную a
 fscanf(file, "%f", &b); // считываем вещ. число в переменную b
} catch (FileNotFoundException e) {
 printf("Ошибка при чтении файла");
} catch (IOException e) {
 printf("Ошибка при чтении файла");
}
```

## Достоинства:

- Четко виден основной вариант алгоритма, когда нет ошибок
- Обработка ошибок, которые хотим обрабатывать одинаково, происходит централизованно
- Если мы не можем обработать ошибку здесь, то брошенное исключение может быть обработано «выше» по стеку вызовов

# Задачи

- Используйте исключения в курсовых и домашних задачах там, где они требуются
- Например, в конструкторах, сеттерах и методах, если передают некорректные данные