КЛАССЫ И ООП

ОБЪЯВЛЕНИЕ КЛАССА

```
/* modifiers */ class Example {
        /* class content : fields and methods */
}
```

ПОЛЯ

```
class Example {
        /* modifiers */ int number;
        /* modifiers */ String text = "hello";
}
```

- Поля инициализируются значениями по умолчанию

МЕТОДЫ

```
class Example {
        int number ;
        /* modifiers */ int getNumber () {
                return number ;
        }
}
```

Возможна перегрузка методов (несколько одноименных методов с разными параметрами)

КОНСТРУКТОРЫ

```
class Example {
        int number ;
        /* modifiers */ Example (int number ) {
                this.number = number ;
        }
}
```

- Если не объявлен ни один конструктор, автоматически создается конструктор по умолчанию (без параметров)

ДЕСТРУКТОР

- В Java нет деструкторов, сбор мусора автоматический
- Есть метод void finalize(), но пользоваться им не рекомендуется.
- При необходимости освободить ресурсы заводят обычный метод void close() или void dispose()

finalize()

- Метод finalize() вызывается один раз перед тем, как сборщик мусора удалит данный объект.
- А что будет, если в методе finalize() добавить ссылку на себя, в какой-нибудь глобальный список?
- Будет плохо. Именно поэтому пользоваться этим методом не рекомендуется.

СОЗДАНИЕ ЭКЗЕМПЛЯРА

```
Example e = null ;
// e. getNumber () -> NullPointerException

e = new Example (3);
// e. getNumber () -> 3

e. number = 10;
// e. getNumber () -> 10
```

НАСЛЕДОВАНИЕ

```
class Derived extends Example {
        /* derived class content */
}
```

- Java не поддерживает множественное наследование, но есть интерфейсы
- Все классы наследуют java.lang.Object

КОНСТРУКТОР В НАСЛЕДНИКЕ

```
class Derived extends Example {
        Derived () {
                this(10);
        }
        Derived (int number) {
                super (number);
        }
}
```

ИНТЕРФЕЙСЫ

Интерфейс определяет возможные сообщения,
но не их реализацию

```
interface ExampleInterface {
        int getNumber ();
}
```

Класс может реализовывать несколько интерфейсов

```
class Example implements ExampleInterface, AutoCloseable {
        int getNumber () {
                // implementation
        }
}
```

МОДИФИКАТОР abstract

abstract class Example {...}

нельзя создать экземпляр класса

abstract int getNumber();

метод без реализации (класс должен быть абстрактным!)

java.lang.Object

- String toString()

- boolean equals(Object obj)
- int hashCode()

- Class<?> getClass()

- void wait() — три варианта
- void notify()
- void notifyAll()

- void finalize()
- void clone()

ПРИМЕРЫ ИЕРАРХИИ ИЗ JDK

java.lang.Object

- java.lang.Number
  - java.lang.Integer
  - java.lang.Double

- java.lang.Boolean
- java.lang.Character
- java.lang.String

- java.lang.AbstractStringBuilder
  - java.lang.StringBuilder
  - java.lang.StringBuffer

МОДИФИКАТОРЫ

МОДИФИКАТОРЫ ДОСТУПА

- public
  - доступ для всех
- protected
  - доступ в пределах пакета и дочерних классов
- private
  - доступ в пределах класса
- по умолчанию (нет ключевого слова)
  - доступ в пределах пакета

- Могут использоваться перед классами, методами, полями

МОДИФИКАТОР final

- «Это нельзя изменить»
- Можно использовать для данных, методов, классов

- Для классов – нельзя иметь наследников
- Для методов – нельзя изменить в производных классах (private => final)

final поля

Либо:

- Константа времени компиляции
- Значение инициализируемое в процессе работы, но которое нельзя изменить. (Аккуратнее со ссылками!)

```
public class BlankFinal {
        private final int i = 0;
        public static final int N = 10;
        private final int j;

        public BlankFinal(int x) {
                j = x;
        }
}
```

ИНИЦИАЛИЗАЦИЯ

ИНИЦИАЛИЗАЦИЯ

```
....
void f() {
        int i;
        i++; //не компилируется
}
...
```

Обязательно инициализировать переменные при объявлении внутри методов!

ИНИЦИАЛИЗАЦИЯ ПОЛЕЙ

```
class Test {
        int i;
        public Test() {
                i++;
                System.out.println(i);
        }
}
```

Поля инициализируются значениями по умолчанию

ИНИЦИАЛИЗАЦИЯ ПОЛЕЙ

```
class Test {
        int n = 3;
        int i = foo(n);
        int foo(int num) {
                return 11*num;
        }
}
```

СТАТИЧЕСКИЕ СЕКЦИИ

```
public class StaticTest {
        static int i;  int j, h;
        static {
                i = 25;
                System.out.println("Hello1");
        }
        {
                j = 8;
                h = 3;
                System.out.println("Hello2");
        }
        public static void main(String[] args) {
                System.out.println("Hello3");
                StaticTest t = new StaticTest();
        }
}
```

ИСКЛЮЧЕНИЯ

ЧТО ТАКОЕ «ИСКЛЮЧЕНИЕ»

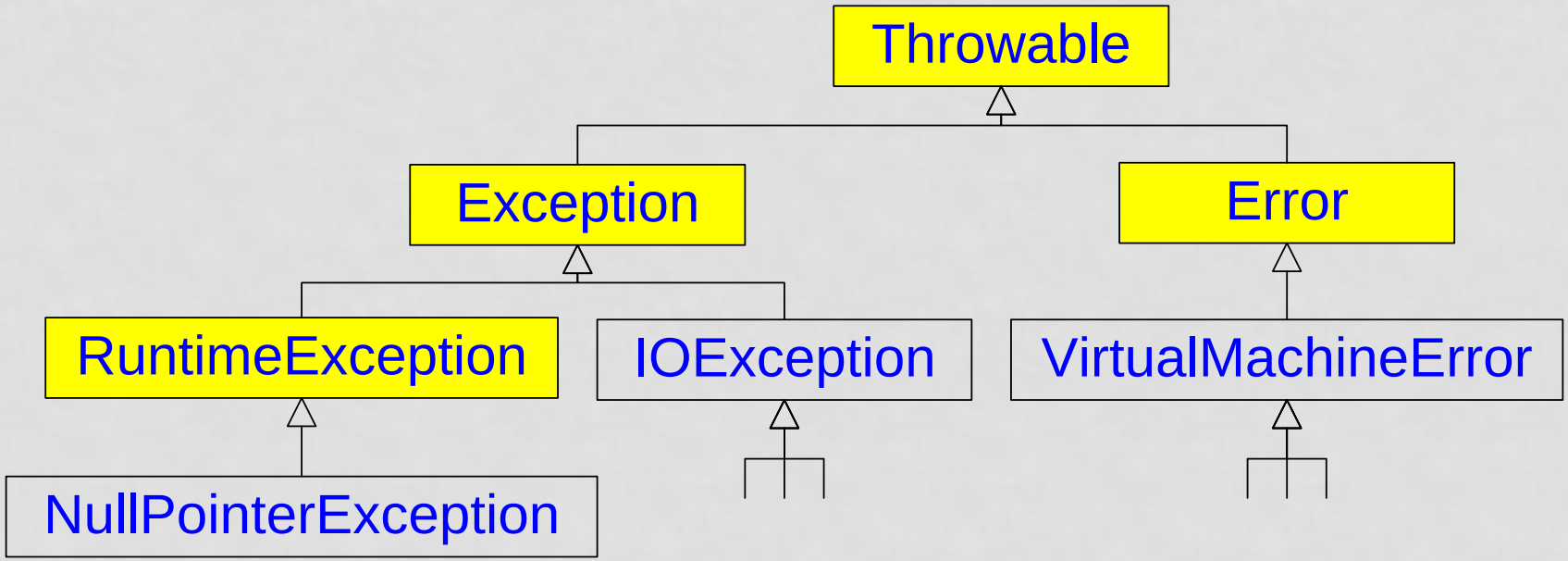
- Исключение (exception) — событие, возникающее в процессе работы программы и прерывающее её нормальное исполнение

- Примеры:
- java.lang.NullPointerException
- java.lang.ArrayIndexOutOfBoundsException
- java.lang.ClassCastException
- java.lang.ArithmeticException
- java.lang.OutOfMemoryError
- java.io.IOException

ПРИЧИНЫ ОШИБОК

- Ошибки программирования – **непроверяемые исключения**
  - NullPointerException
- Неверное использование API –**непроверяемые (чаще) или проверяемые исключения**
  - InvalidArgumentException
- Доступ к внешним ресурсам – **проверяемые исключения**
  - IOException
- Системные сбои
  - VirtualMachineError

ИЕРАРХИЯ ИСКЛЮЧЕНИЙ



java.lang.Throwable

- Исключение в Java — полноценный объект
- Все исключения в Java наследуются от класса Throwable

- String getMessage() - сообщение об ошибке
- Throwable getCause() - причина исключения
- StackTraceElement[] getStackTrace()
- void printStackTrace() - печать стека исполнения

ГЕНЕРАЦИЯ ИСКЛЮЧЕНИЯ

- Либо генерируется используемым кодом
- Либо генерируем сами

```
public static int parseInt (String s, int radix)
    throws NumberFormatException {
        if (s == null ) {
                throw new NumberFormatException (" null ");
        }
        // code skipped
}
```

- Оператор throw прерывает нормальное исполнение программы и запускает поиск обработчика исключения
- Если исключение проверяемое, метод должен содержать его в списке throws

РАБОТА С ИСКЛЮЧЕНИЯМИ

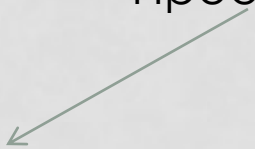
При вызове метода, который бросает проверяемое исключение необходимо

- Либо обработать его (перехватить)
- Либо пробросить дальше (написать throws … у текущего метода)

```
public void foo() throws IOException {
      . . .
}

public void bar() throws IOException {
      foo();
}
```

Проброс исключения



ПЕРЕХВАТ ИСКЛЮЧЕНИЙ

```
System.out.print("Please enter number: ");
int n = 0;
while (true) {
	String s = readUserInput();
	try {
		n = Integer.parseInt(s);
		break;
	} catch (NumberFormatException e) {
		System.out.print("Bad number, try again: ");
	}
}
```

ПЕРЕХВАТ НЕСКОЛЬКИХ ИСКЛЮЧЕНИЙ

```
try {
        // ...
} catch (IOException e) {
        e.printStackTrace();
} catch (NumberFormatException e) {
        e.printStackTrace();
}

// this works since Java 7:
try {
        // ...
} catch (IOException | NumberFormatException e) {
        e.printStackTrace();
}
```

ОБРАБОТКА ИСКЛЮЧЕНИЙ

Если в коде вызываются методы, бросающие проверяемые исключения, эти исключения надо либо поймать и обработать (catch), либо добавить в список throws

Стратегии обработки:

- Игнорирование (пустой catch) - ПЛОХО
- Запись в лог – ТОЖЕ ПЛОХО
- Проброс дальше того же или нового исключения
- Содержательная обработка (например, повтор операции)

ИСКЛЮЧЕНИЕ И РЕСУРСЫ

```
InputStream is = new FileInputStream("a.txt");
try {
      readFromInputStream(is);
} finally {
      is.close();
}
```

Блок finally будет выполнен в любом случае

В нем обычно освобождают использованные ресурсы

try with resources

```
try (InputStream is=new FileInputStream("a.in")) {
      readFromInputSteam(is);
}
```

- Добавлен в Java 7
- Метод close() будет вызван автоматически, как в finally
- Можно перечислить несколько ресурсов через ;
- Ресурсы должны реализовать интерфейс java.lang.AutoCloseable

try – catch – finally

```
try {
    // Действия, способные вызвать исключение
} catch (*Exception e) {
    // Обработка исключений первого типа
} catch (*Exception e) {
    // Обработка исключений второго типа
} finally {
    // Действия, выполняемые при выходе из блока
}
```

ПРИМЕНЕНИЕ ИСКЛЮЧЕНИЙ В КОДЕ

ВЫДЕЛЕНИЕ КОДА ОБРАБОТКИ ОШИБОК

- Ошибки обрабатываются там, где для этого достаточно информации

```
try {
    f();
} catch (*Exception e) {
    …
}


f() { g() }
g{ … throw new *Exception(…) }
```

УПРАВЛЕНИЕ РЕСУРСАМИ

```
// Получение ресурса
try {
   // Действия с ресурсом
} finally {
   // Освобождение ресурса
}
```

УПРАВЛЕНИЕ НА ИСКЛЮЧЕНИЯХ

```
try {
    int index = 0;
    while (true) {
        System.out.println(a[index++]);
    }
} catch (IndexOutOfBoundsException e) {
}
```

ИГНОРИРОВАНИЕ ИСКЛЮЧЕНИЙ

- Полное игнорирование

```
try {

} catch (Exception e) { }
```

- Запись в лог

```
try {

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

ПЕРЕХВАТ БАЗОВЫХ ИСКЛЮЧЕНИЙ

```
try {

} catch (Exception e) {
    // Что-то полезное
}
```

ПОЛЬЗОВАТЕЛЬСКИЕ ИСКЛЮЧЕНИЯ

```
public class BadMoveException extends Exception {
	public BadMoveException(String message) {
		super(message);
	}
}
```

РАЗРАБОТКА СОБСТВЕННЫХ ИСКЛЮЧЕНИЙ

- Что пользователь может сделать с исключением?
  - Ничего – непроверяемое исключение
  - Что-то осмысленное – проверяемое исключение
- Пользовательский код не должен знать об устройстве класса
  - Обертывание исключений
    - В непроверяемые
    - В проверяемые
  - Игнорирование исключений

ОБЕРТЫВАНИЕ ИСКЛЮЧЕНИЙ

- Правильно

```
try {
} catch (IOException e) {
    throw new APISpecificException(e);
}
```

- Неправильно

```
try {
} catch (IOException e) {
    e.printStackTrace()
    throw new APISpecificException();
}
```