

Прикладное программирование

Лекция №4.

1. Классы (повторение)
2. Основы Git (повторение)
3. Обработка исключений
4. Работа с базовыми объектами

Студенты плакали и кололись,



но продолжали учить Java.

Java. Class

Что такое класс?

Класс — это описание того, как **будет устроен объект**, являющийся экземпляром данного класса, и какие **методы этот объект сможет вызывать**.

Какую задачу решают классы?

Классы используются для того, чтобы создавать **объекты**, которые обладают четко заданными **свойствами** и **методами**.

```
1  public class Student {
2      private String faculty;
3      private String speciality;
4      private String name;
5      private int group;
6
7      public Student(String faculty, String speciality, String name, int group) {
8          this.faculty = faculty;
9          this.speciality = speciality;
10         this.name = name;
11         this.group = group;
12     }
13
14     // your getter/setter should be here
15 }
16
17 public class Basic {
18     public static void main(String[] args) {
19         Student student = new Student("RFCT", "CS", "Igor", 8);
20         System.out.println(student);
21     }
22 }
23
```

Java. Interface

Что такое интерфейс?

Интерфейсы — это специальная разновидность полностью абстрактных классов, которые:

- **Не имеют** реализованных методов;
- **Не могут** включать поля данных
- **Могут** содержать константы.

Какую задачу решает интерфейс?

Интерфейсы используются для **описания функциональности**, которую **должен реализовать каждый класс**, который реализует этот интерфейс.

```
1
2 public interface Education {
3     public void getDiploma();
4     public void takeExam();
5 }
6
7 public class Student implements Education {
8     public void getDiploma() {
9         System.out.println("Студент получает диплом");
10    }
11
12    public void takeExam() {
13        System.out.println("Студент сдает экзамен");
14    }
15 }
16
17 public class UndergraduateStudent implements Education {
18     public void getDiploma() {
19         System.out.println("Магистрант получает диплом");
20    }
21
22    public void takeExam() {
23        System.out.println("Магистрант сдает экзамен");
24    }
25 }
```

Java. Встроенные классы

- **Внутренние (Inner) классы** – non-static.
- **Вложенные (Nested) классы** – static.
- **Локальные классы** – only inside your methods.
- **Анонимные классы** – «on the run» classes.

Inner classes

Какую задачу помогают решить?

Внутренние классы реализуют **более развернутое описание** каких-либо компонентов внешнего класса.

Если нет необходимости каждый компонент родительского класса описывать в отдельном внешнем классе, то более рациональным решением будет расположить всю структуру данных в одном классе.

Inner classes

И создал Бог **Человека**.

И наделил Бог **Человека Сердцем**, которое может стучать.

Или нет 😊

```
1
2  public class Main {
3      public static void main(String[] args) {
4          Human testHuman = new Human();
5          testHuman.die();
6      }
7  }
8
```

```
1
2 public class Human {
3     private int health = 100;
4     private Heart heart;
5
6     public Human() {
7         this.heart = new Heart();
8         heart.beat();
9     }
10
11     public void die() {
12         heart.stop();
13     }
14
15     private class Heart {
16         public void beat() {
17             System.out.println("I'm heart and i'm beating!");
18             health = health;
19         }
20
21         public void stop() {
22             System.out.println("Упс, моя остановочка!");
23             health = 0;
24         }
25     }
26 }
27
```

Inner (non-static) class

Inner class

Свойства внутренних классов:

- Inner классы существуют **только у объектов**, поэтому для их создания нужен объект. *Сердце – важная часть человека. Если человека уже нет – зачем ему сердце?*
- Внутри Java класса **не может быть статических переменных**. Если нужны константы или прочие статические поля, их следует выносить во внешний класс.
- **У класса полный доступ ко всем приватным полям внешнего класса**. Данная особенность работает в две стороны.

Nested classes

Вложенные статические классы отличаются от внешнего класса только одним свойством:

```
Building.House house = new Building.House();
```

Какую задачу решают?

Статические классы используются для того, чтобы **упорядочить связанные классы в одном месте**, так как с логической структурой было работать проще.

Nested classes

Как пользоваться вложенными классами?

Давайте создадим программиста **Игоря** с весьма задорной и насыщенной жизнью:

```
1
2 public class Main {
3
4     public static void main(String[] args) {
5         Person person = new Person("Igor", 26);
6         person.findJob(new Person.Job("Programmer", 164));
7         person.work();
8         person.work();
9         person.work();
10    }
11 }
12
```

Nested classes

Как это работает?

Вы создаете внешний класс **Person**, в котором будет содержаться конкретный список классов.

Эти классы будут представлять из себя «**процессы**» человека:

- **Работа;**
- **Хобби;**
- **Путешествия.**

В противном случае, Вам понадобится создавать несколько отдельных классов.

```
1
2 public class Person {
3     private String name;
4     private int age;
5     private Job currentJob;
6
7     public Person(String name, int age) {
8         this.name = name;
9         this.age = age;
10    }
11
12    public void findJob(Job job) {
13        this.currentJob = job;
14    }
15
16    public void work() {
17        if (currentJob == null) {
18            System.out.println("House work");
19        } else {
20            currentJob.visit();
21        }
22    }
23
24    public static class Job {
25        private String name;
26        private int weeklyWorkingHours;
27
28        public Job(String name, int weeklyWorkingHours) {
29            this.name = name;
30            this.weeklyWorkingHours = weeklyWorkingHours;
31        }
32
33        public void visit() {
34            System.out.println("Working " + name + " for " + weeklyWorkingHours / 5 + " a day");
35        }
36    }
37 }
38
```



Nested classes

Преимущества:

- Количество классов **значительно меньше**.
- Все классы **размещены внутри класса-родителя**, при этом легко можно проследить всю иерархию классов без открытия каждого из них по-отдельности.
- Если обратиться напрямую к классу **Person**, IDE будет отображать список всех подклассов данного класса, что **сильно упрощает поиск необходимых классов**.

Anonymous classes

Зачем нужны анонимные классы?

Анонимные классы используются для реализации **нескольких методов** и **создания собственных методов объекта**. Эффективно применяется только тогда, когда необходимо **переопределение метода**, но **создавать новый класс нет необходимости**.

Ключевая особенность анонимных классов:

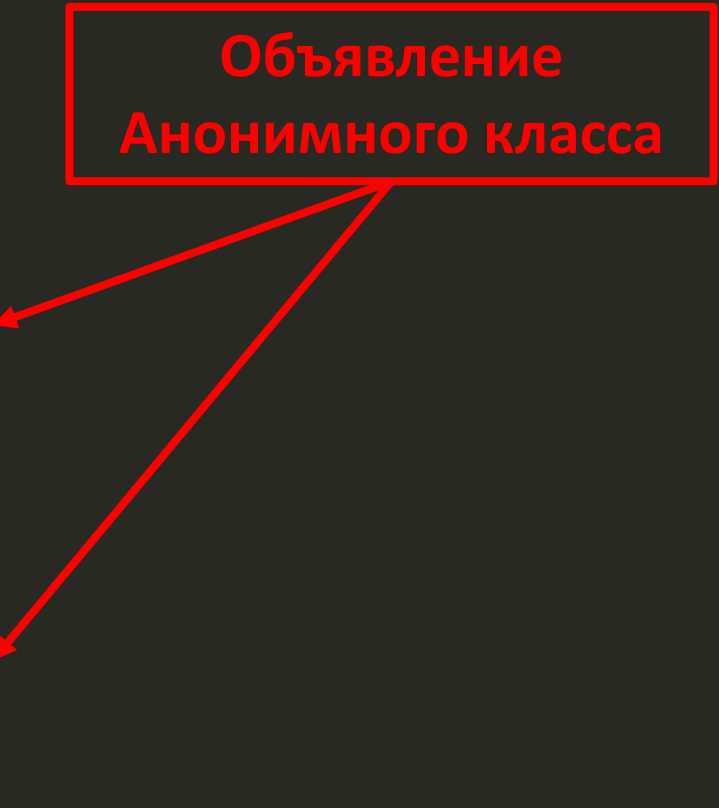
Ключевая особенность анонимных классов заключается **в удобстве их использования**. Это позволяет Вам написать свой класс прямо при создании экземпляра другого класса.

```
1
2 public interface MusicInstrument {
3     void play();
4 }
5
6 class Musician {
7     public void play(MusicInstrument instrument) {
8         instrument.play();
9     }
10 }
```

```
12 Musician maestro = new Musician();
13 maestro.play(new MusicInstrument() {
14     public void play() {
15         System.out.println("Piano");
16     }
17 });
```

```
19 maestro.play(new MusicInstrument() {
20     public void play() {
21         System.out.println("Guitar");
22     }
23 });
```

**Объявление
Анонимного класса**



Anonymous classes

Когда стоит использовать анонимные классы?

- Когда тело класса **короткое**;
- Необходим только **один экземпляр** класса;
- Класс используется **в месте его создания** или **сразу после него**;
- Имя класса никак **не влияет на понимание** кода в целом.

Anonymous classes

Как на практике применяются анонимные классы?

Как правило, анонимные классы часто используются **для создания обработчиков событий.**

Например, для **создания кнопки и обработки события нажатия:**

```
1
2  Button rightButton = new Button();
3  rightButton.addActionListener(new ActionListener() {
4      public void actionPerformed(ActionEvent e) {
5          System.out.println("Right button was pressed!");
6      }
7  });
8
```

Local classes

Локальные классы объявляются **внутри других методов**.

По сути, это те же **нестатические вложенные классы**, только создавать их экземпляры **можно только в методе!**, при этом сам метод **не может быть статическим**.

Как применяются на практике?

На практике локальные классы можно встретить довольно редко, т.к. часто они **затрудняют прочтение кода** и **не дают особых преимуществ**, кроме **обеспечения доступа к переменным метода**.

```
1 public class Outer {
2     void outerMethod() {
3         System.out.println("Метод внешнего класса");
4         // Внутренний класс является локальным для метода outerMethod()
5
6         class Inner {
7             public void innerMethod() {
8                 System.out.println("Метод внутреннего класса");
9             }
10        }
11
12        Inner inner = new Inner();
13        inner.innerMethod();
14    }
15
16    public static void main(String[] args) {
17        Outer outer = new Outer();
18        outer.outerMethod();
19    }
20 }
21
```

Метод внешнего класса

Метод внутреннего класса

Local classes

Особенности:

- Локальные классы способны работать только **с final переменными метода**. Если же переменная объявлена **final**, то компилятор может сохранить **копию переменной** для дальнейшего использования объектом.
- Локальные классы **нельзя объявлять с модификаторами доступа**.
- Локальные классы **обладают доступом к переменным метода**.

Git Basics

- Что такое **Git**?
- Как использовать **Git**?
- Что такое **репозиторий**?
- Что такое **Fork**?
- Что такое **ветка**?
- Что такое **коммит**?
- Как посмотреть **изменения**?
- **Команды Git**

Базовый класс Object

Класс **Object** является базовым для всех классов Java, поэтому все его поля и методы наследуются и содержатся во всех классах.

Object. Methods

- public Boolean **equals(Object obj)**
- public int **hashCode()**
- protected Object **clone()** throws **CloneNotSupportedException**
- public final Class **getClass()**
- protected void **finalize()** throws **Throwable**
- public String **toString()**
- void **notify()**
- void **notifyAll()**
- void **wait()**

Object. Сравнение объектов

public Boolean equals(Object obj) – возвращает **true** в случае, когда равны значения объекта, из которого вызывается метод, и объекта, передаваемого через ссылку **obj** в списке параметров.

Если объекты не равны, возвращается **false**.

Версия **equals()**, реализованная в классе **Object**, делает то же самое, что и «==», т.е. сравнивает ссылки.

В потомках этот метод может быть переопределен, и может сравнивать объекты по их содержимому.

Object. hashCode

public int hashCode() – возвращает хэш-код объекта

(уникальный числовой идентификатор), сопоставляемый объекту.

Данный идентификатор используется при организации коллекций элементов (списков, множеств, хэшей) для быстрой адресации элементов коллекций.

Рекомендуется переопределять данный метод каждый раз, когда переопределяется метод сравнения объектов **equals()**.

Object. Клонирование объектов

protected Object `clone()` throws `CloneNotSupportedException`:

данный метод осуществляет копирование объекта и возвращает ссылку на созданный дубликат объекта.

В наследниках класса **Object** этот метод следует обязательно переопределять, а также указывать, что класс реализует интерфейс **Cloneable**.

Попытка вызова метода из объекта, который не поддерживает клонирование, вызовет создание исключительной ситуации **`CloneNotSupportedException`**.

Object. Клонирование объектов

Различают два вида клонирования:

1. **мелкое (shallow)** – в клон один к одному копируются значения полей оригинального объекта, а ссылочные поля будут указывать на те же самые экземпляры объектов, что и поля оригинала;
2. **глубокое (deep)** – для полей ссылочного типа создаются новые объекты, клонирующие объекты, на которые ссылаются поля оригинала.

Object. Клонирование объектов

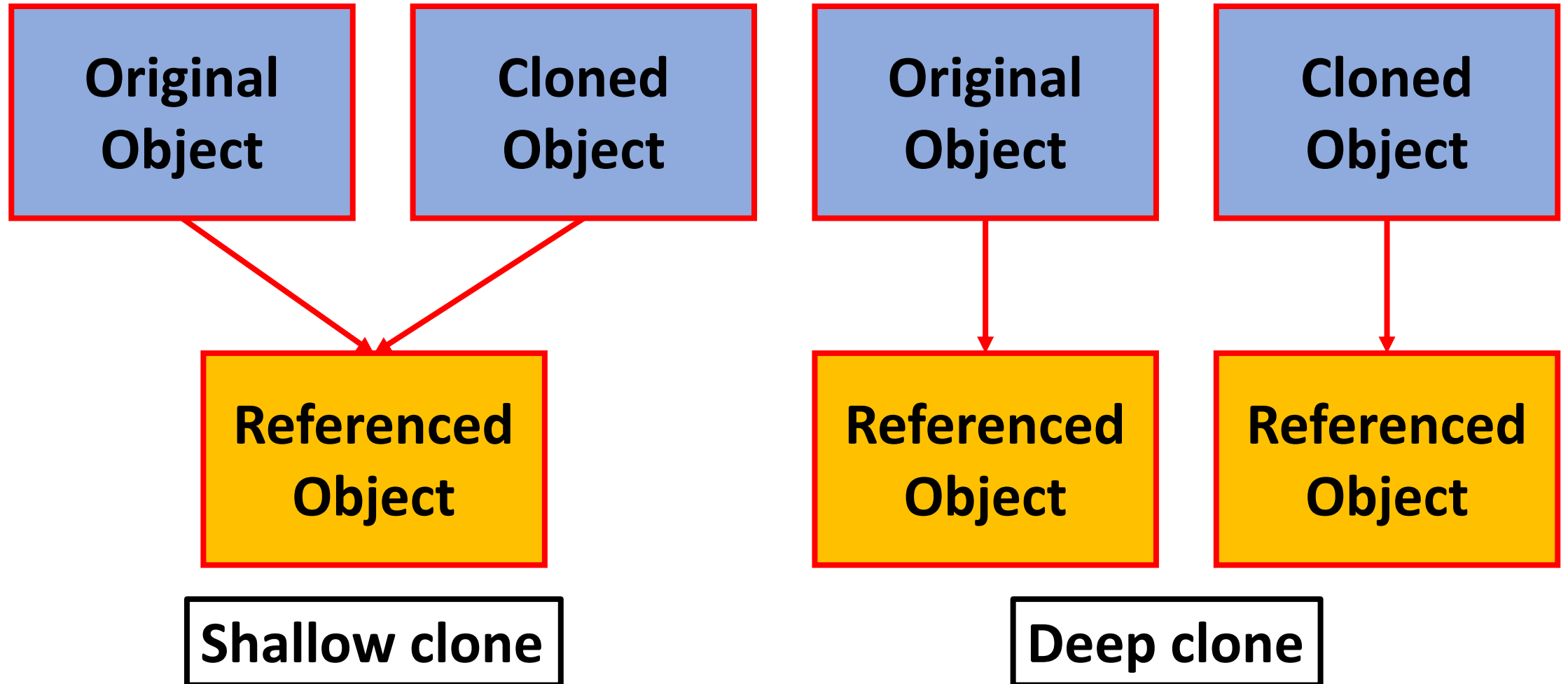
Какой тип клонирования лучше использовать?

Если объект имеет поля только примитивных типов, различий между мелким и глубоким клонированием нет.

Реализацией клонирования занимается разработчик класса, т.к. автоматического механизма клонирования нет, и именно на этапе проектирования класса следует решить, какой вариант клонирования стоит выбирать.

В большинстве случаев требуется глубокое клонирование.

Shallow and Deep Clone



Object. Описание объектов

public final Class getClass()

– возвращает ссылку на метаобъект типа Class.

```
Class c = System.console().getClass();
```

С его помощью можно получать информацию о классе, к которому принадлежит объект, и обращаться к его полям и методам (библиотека **Reflection API**).

Object. Описание объектов

Метод **getClass()** возвращает значение класса, соответствующее java.io.Console.

```
// enum – перечисление логически связанных объектов  
enum E { A, B }  
Class c = A.getClass();
```

Если **A** – это экземпляр **E**, следовательно метод **getClass()** вернет класс, соответствующий типу перечисления **E**.

Object. Разрушение объектов

protected void `finalize()` throws `Throwable`:

Метод `finalize()` вызывается перед уничтожением объекта.

Он строго должен быть переопределен в тех потомках **Object**, в которых требуется совершать такие **вспомогательные действия** перед уничтожением объекта, как:

- Заккрытие файла;
- Вывод сообщения;
- Т.д.

Object. Разрушение объектов

```
protected void finalize() throws Throwable {  
    try {  
        close(); // close files for example  
    } catch (IOException e) {  
        // do smth  
    } finally {  
        super.finalize();  
    }  
}
```

Object. Строковое представление объектов

public String toString() – возвращает строковое представление объекта.

В классе **Object** этот метод реализует выдачу полного имени объекта (вместе с именем пакета), после которого следует символ «@», а затем в шестнадцатеричном виде хэш-код объекта.

В большинстве стандартных классов этот метод переопределен.

Для числовых классов – строковое представление числа.

Для строковых классов – содержимое строки.

Для символьного – сам символ.

```
1
2 public class Student {
3     private String faculty;
4     private String speciality;
5     private String name;
6     private int group;
7
8     public Student(String faculty, String speciality, String name, int group) {
9         this.faculty = faculty;
10        this.speciality = speciality;
11        this.name = name;
12        this.group = group;
13    }
14
15    // your getter/setter should be here
16 }
17
18 public class Basic {
19     public static void main(String[] args) {
20         Student student = new Student("RFCT", "CS", "Igor", 8);
21         System.out.println(student); // Compiler writes here student.toString()
22     }
23 }
```

Output:Student@1fee6fc

Object. Синхронизация объектов

Иногда при взаимодействии нескольких потоков в приложении возникает вопрос **об извещении одних потоков о действиях других.**

Например, когда действия одного потока зависят от результата действий другого, при этом нужно известить первый поток о том, что второй совершил некоторое действие.

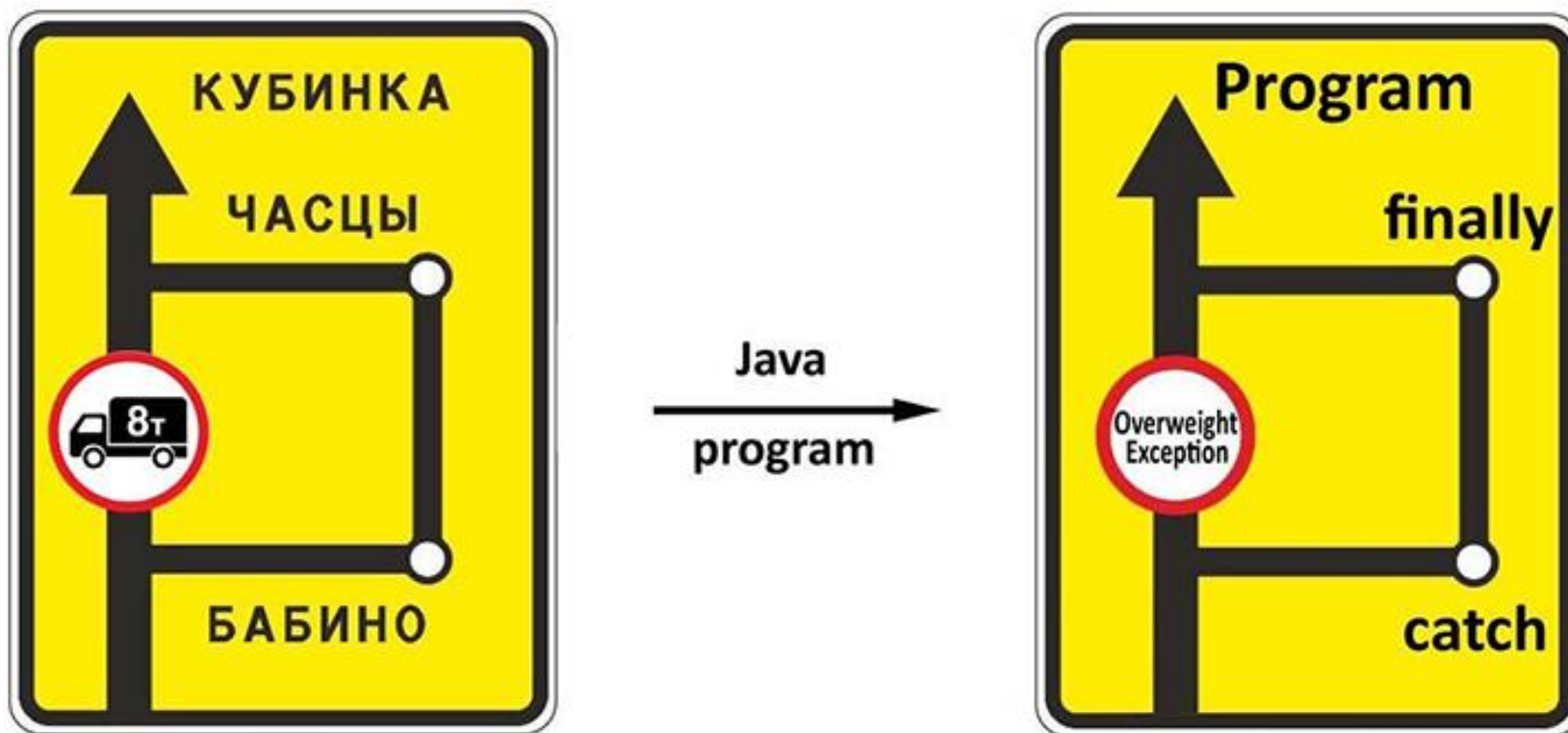
Object. Синхронизация объектов

Для подобных ситуаций у класса **Object** определен ряд методов:

- **wait()**: освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет **метод notify()**
- **notify()**: продолжает работу потока, у которого ранее был вызван **метод wait()**
- **notifyAll()**: возобновляет работу всех потоков, у которых ранее был вызван **метод wait()**

Все эти методы вызываются только из синхронизированного контекста - синхронизированного блока или метода.

Обработка исключительных ситуаций



Обработка исключительных ситуаций

В штатном режиме работы приложения выполнение операторов обычно идет в рамках основной ветви блок-схемы реализуемого им алгоритма.

Но время от времени по непредсказуемым, но потенциально возможным обстоятельствам (деление на 0, недостаточные права при обращении к файлу, отсутствие места на диске, ввод ошибочного символа), возникают исключительные ситуации (исключения), требующие обработки и приводящие к отклонению от основной ветви.

Обработка исключительных ситуаций

В отличие от **катастрофических ситуаций (ошибок)** такие исключения в большинстве случаев могут быть учтены в программе, и, в частности, **не должны приводить к ее аварийному завершению.**

Решение указанных проблем с помощью различных проверок на допустимость присваиваний и математических операций замедляет работу приложения и не гарантирует его надежной работы.

Вместо этого в Java применяется механизм обработки исключительных ситуаций на основе конструкции **try-catch-finally.**

Конструкция try-catch-finally

Общий случай использования защищенного блока программного кода и перехвата исключительных ситуаций выглядит так:

```
try { // операторы }  
catch (<ТипИсключения1> <переменная>) { // операторы }  
...  
catch (<ТипИсключенияN> <переменная>) { // операторы }  
finally{ // операторы; }
```

Конструкция try-catch-finally

Операторы в блоке **try** выполняются в обычном порядке до возникновения исключительной ситуации, после чего их **выполнение прерывается и происходит анализ типов обрабатываемых ситуаций**, представленных блоками **catch**.

В качестве параметра передается **ссылочная переменная**, имеющая **тип исключения**, которое должен перехватывать данный блок.

Если тип возникшего исключения совместим с типом, указанным в блоке **catch**, то выполняются соответствующие операторы, и проверки в следующих блоках **catch** не выполняется.

После перехвата исключений (или если исключений не произошло) выполняются операторы блока **finally**. **Они выполняются всегда**, даже в том случае, если в блоке **catch** с помощью операторов **break**, **continue**, **return** или **System.exit()** выполняется прерывание работы блока программного кода.

Пример обработки исключений с помощью try-catch

```
try {  
    account.open();  
    double balance=Double.parseDouble(account.getBalance());  
    account.setBalance(balance+100.0);  
    account.save();  
    account.close();  
}  
catch (NullPointerException e) {  
    LOG.fatal("Поле balance объекта Account равно null");  
    account.close();  
    System.exit(2);  
}  
catch (NumberFormatException e) {  
    LOG.error("Значение " + account.getBalance() +  
        " не является записью вещественного числа");  
    account.close();  
}
```

Пример обработки исключений с помощью try-catch-finally

```
try {
    account.open();
    double balance=Double.parseDouble(account.getBalance());
    account.setBalance(balance+100.0);
    account.save();
}
catch (NullPointerException e) {
    LOG.fatal("Поле balance объекта Account равно null");
    System.exit(2);
}
catch (NumberFormatException e) {
    LOG.error("Значение " + account.getBalance() +
        " не является записью вещественного числа");
}
finally {
    account.close();
}
```


Иерархия исключительных ситуаций

Типы исключительных ситуаций в Java определяются потомками **Throwable** – классами **Error** и **Exception**.

- Экземплярами класса **Error** являются непроверяемые исключительные ситуации, представляющие катастрофические ошибки, после которых невозможна нормальная работа приложения, поэтому их невозможно перехватить в блоках **catch**.
- Экземплярами класса **Exception** и его потомков являются проверяемые исключительные ситуации (за исключением класса **RuntimeException** и его потомков).

Иерархия исключительных ситуаций

Классы исключительных ситуаций либо predetermined в стандартных пакетах (например, **ArithmeticException** – в пакете **java.lang**, **IOException** – в пакете **java.io** и т.д.), либо описываются самостоятельно как наследники класса **Exception** или его потомков.

В Java типы-исключения принято именовать, оканчивая имя класса на **Exception** для проверяемых исключений или на **Error** для непроверяемых.

По правилу совместимости типов исключительная ситуация типа-потомка всегда может быть обработана как исключение родительского типа, поэтому порядок следования блоков **catch** имеет большое значение.

ArithmeticException	Арифметическая ошибка: деление на нуль и др.
ArrayIndexOutOfBoundsException	Индекс массива находится вне границ
ArrayStoreException	Назначение элементу массива несовместимого типа
ClassCastException	Недопустимое приведение типов
ConcurrentModificationException	Некорректная модификация коллекции
IllegalArgumentException	При вызове метода использован незаконный аргумент
IllegalMonitorStateException	Незаконная операция монитора на разблокированном экземпляре
IllegalStateException	Среда или приложение находятся в некорректном состоянии

IllegalThreadStateException	Требуемая операция не совместима с текущим состоянием потока
IndexOutOfBoundsException	Некоторый тип индекса находится вне границ
NegativeArraySizeException	Массив создавался с отрицательным размером
NullPointerException	Недопустимое использование нулевой ссылки
NumberFormatException	Недопустимое преобразование строки в числовой формат
StringIndexOutOfBoundsException	Попытка индексации вне границ строки
UnsupportedOperationException	Встретилась неподдерживаемая операция

Объявление собственных исключительных ситуаций

Для того чтобы задать собственный тип исключительной ситуации, требуется задать соответствующий класс. Он должен быть наследником от какого-либо класса исключительной ситуации, например:

```
class MyOwnException extends Exception {  
    int param1;  
    String param2;  
    MyOwnException(int param1, String param2) {  
        this.param1 = param1;  
        this.param2 = param2;  
    }  
    int getParam1() { return param1; }  
    String getParam2() { return param2; }  
}
```

Создание объекта-исключения может проводиться в произвольном месте программы обычным образом, как для всех объектов, при этом возбуждения исключения не происходит.

Возбуждение исключительных ситуаций

Программное возбуждение исключительной ситуации производится с помощью оператора **throw**, после которого указывается оператор создания объекта-исключения:

```
throw new <Тип_исключения> (<Параметры_конструктора>) ;
```

Если после частичной обработки требуется повторно возбудить исключительную ситуацию **exception**, используется вызов **throw exception**;

```
1
2 public static int getFactorial(int num) throws Exception {
3
4     if (num < 1) throw new Exception("The number is less than 1");
5
6     int result = 1;
7     for(int i = 1; i <= num; i++){
8
9         result *= i;
10    }
11
12    return result;
13 }
14
```

Возбуждение исключительных ситуаций

Для проверяемых исключений всегда **требуется явное возбуждение**. При возбуждении исключения во время выполнения какого-либо метода **прерывается основной ход программы**, и идет **процесс обработки исключения**.

Если в методе исключение данного типа **не перехватывается**, то выполняется соответствующий блок **finally** (если он есть), и происходит выход из текущего метода в метод более высокого уровня.

Если оно не перехватывается и там, то происходит выход еще на уровень выше и т.д. (этот процесс называют **«всплыванием» исключения**).

Если в методе исключение данного типа **перехватывается**, то всплывание **прекращается**. Если исключение **не перехвачено** ни на одном из уровней, то оно **обрабатывается исполняющей средой**.

Объявление методов, способных возбудить исключения

Формат объявления функции, которая может возбуждать проверяемые исключительные ситуации, следующий:

```
<Модификаторы> <Тип> <Имя> (<Список_параметров>)  
throws <Тип_исключения1>, ..., <Тип_исключенияN> {  
    // Тело функции  
}
```

Непроверяемые исключения генерируются и обрабатываются системой автоматически, и, как правило, приводят к завершению приложения (их типы нигде не указываются, и **throws** в заголовке указывать не надо).

```
1
2 public static int getFactorial(int num) throws Exception {
3
4     if (num < 1) throw new Exception("The number is less than 1");
5
6     int result = 1;
7     for(int i = 1; i <= num; i++){
8
9         result *= i;
10    }
11
12    return result;
13 }
14
```

Объявление методов, способных возбудить исключения

Если в теле реализуемого метода используется метод, который может возбуждать исключительную ситуацию, и это исключение не перехватывается, в заголовке реализуемого метода **требуется указывать соответствующий тип возбуждаемого исключения.**

Если же это исключение порождается внутри защищенного блока программного кода, и в каком-либо блоке **catch** перехватывается этот тип исключения или более общий (родительский), то указывать в заголовке тип исключения **не следует.**

Замечание: оператор **throw** может быть использован для порождения исключения другого типа после обработки перехваченного исключения в блоке **catch**.

```
1
2 public static int getFactorial(int num) {
3
4     int result = 1;
5     try {
6         if (num < 1) throw new Exception("The number is less than 1");
7
8         for (int i = 1; i <= num; i++) {
9             result *= i;
10        }
11    }
12    catch(Exception ex){
13        System.out.println(ex.getMessage());
14        result = num;
15    }
16
17    return result;
18 }
19
20
```

Переопределение методов, порождающих исключения

Если в родительском классе задан метод, в заголовке которого указан **тип какой-либо исключительной ситуации**, а в классе-потомке этот **метод переопределяется**, то в переопределяемом методе **также требуется указывать совместимый тип исключительной ситуации**.

При этом может быть указан либо **тот же тип**, либо **тип исключительной ситуации – потомка от данного типа**.

В противном случае на этапе компиляции выдается **диагностика ошибки**.

Java StackOverflow. The most used exceptions

- **Casting**: `ClassCastException`
- **Arrays**: `ArrayIndexOutOfBoundsException`, `NullPointerException`
- **Collections**: `NullPointerException`, `ClassCastException`
- **IO**: `java.io.IOException`, `java.io.FileNotFoundException`, `java.io.EOFException`
- **Serialization**: `java.io.ObjectStreamException` (and its all Subclasses)
- **Threads**: `InterruptedException`, `SecurityException`, `IllegalThreadStateException`
- **Potentially common**: `NullPointerException`, `IllegalArgumentException`

Неизменяемые строки в Java – класс String

Класс **String** инкапсулирует действия со строками, а его экземплярами являются строки-константы, состоящие из произвольного числа символов, **от 0 до 2^{109}** .

Литерные константы типа **String** представляют собой последовательности символов, заключенные в двойные кавычки.

Внутри литерной строковой константы не разрешается использовать ряд символов - вместо них применяются управляющие последовательности (например, перенос на новую строку – “\n”).

Также строка может быть пустой, не содержащей ни одного символа.

Неизменяемые строки в Java – класс String

В языке Java строковый и символьный тип несовместимы!

Поэтому «A» – строка из одного символа,

а 'A' – число с ASCII кодом символа A.

Строки можно складывать:

если **s1** и **s2** – строковые литерные константы или переменные, то результатом операции **s1+s2** будет строка, являющаяся сцеплением строк, хранящихся в **s1** и **s2**.

```
String text = "hello " + "world";    // "hello world"
```


Неизменяемые строки в Java – класс String

Любая строка является объектом – экземпляром класса **String**, а переменные этого класса являются ссылками на объекты, что следует учитывать при передаче параметров в подпрограммы и изменениях строк.

Так как **String** представляет собой строку-константу, то при каждом изменении строки **в динамической области памяти создается новый объект**, а прежний попадает в сборщик мусора.

Поэтому при многократных изменениях строк в цикле занимается слишком много памяти.

Наиболее востребованные методы класса String

charAt(i)	возвращает символ с индексом i в строке (0 – начало).
endsWith(str)	возвращает true в случае, когда строка заканчивается последовательностью символов, содержащихся в строке str .
equals(str)	возвращает true в случае, когда последовательностью символов в конце строки совпадает с последовательностью символов, содержащихся в строке str .
equalsIgnoreCase(str)	аналогично equals(str) , но без учета регистра.
indexOf(str)	индекс позиции, где в строке первый раз встретилась последовательность символов str .
length()	длина строки (для пустой строки – нуль).
replaceFirst(str1,str2)	возвращает строку, в которой первое вхождение последовательности символов str1 заменено на символы строки str2 .
replaceAll(str1,str2)	возвращает строку, в которой все вхождения символов str1 заменены на символы str2 .

Наиболее востребованные методы класса String

split(str)	возвращает массив строк String[] , полученный путем разделением строки на независимые строки по местам вхождения разделителя, задаваемого строкой str . При этом символы, содержащиеся в str , в получившиеся строки не входят. Пустые строки из конца получившегося массива удаляются.
startsWith(str)	возвращает true , если строка начинается с символов строки str .
substring(index1)	возвращает строку с символами, скопированными из строки начиная с позиции index1 .
substring(index1,index2)	возвращает строку с символами, скопированными из строки начиная с позиции index1 и заканчивая позицией index2 .
toCharArray()	возвращает массив символов строки.
toLowerCase()	возвращает копию строки с преобразованными к нижнему регистру символами.
toUpperCase()	возвращает копию строки с преобразованными к верхнему регистру символами.
trim()	возвращает копию строки, из которой убраны ведущие и завершающие пробелы.

Изменяемые строки в Java

класс StringBuffer

Класс **String** предназначен для представления **неизменяемых строк**.

В случае модификации содержания какой-либо строки старый объект выбрасывается для переработки сборщиком мусора, а вместо него выделяется память и происходит инициализация нового объекта, на который и указывает теперь ссылочная переменная типа **String**.

Это приводит к **чрезмерному расходу памяти**, и в том случае, если ожидаются частые изменения какой-либо строки, имеет смысл объявить ее как изменяемую с помощью класса **StringBuffer**.

А после того, как преобразования строки будут завершены, экземпляр класса **String** можно будет получить из объекта класса **StringBuffer** с помощью метода **toString()**.

Методы класса StringBuffer:

append(str)	добавляет к текущему содержимому буфера строку str .
insert(i, str)	вставляет строку str в текущее содержимое буфера начиная с символа i .
reverse()	выстраивает содержимое буфера в обратном порядке.
setCharAt(i, ch)	копирует символ ch в текущее содержимое буфера в позицию i .
charAt(i)	возвращает символ, находящийся в текущем содержимом буфера в позиции i .

Массивы в Java

В Java переменная типа **массив** является ссылочной – в ней содержится адрес объекта, но не сам объект, как и для всех других объектных переменных в Java.

В качестве элементов (ячеек) массива могут выступать значения как примитивных, так и ссылочных типов, в том числе – переменные типа массив.

Тип ячейки массива называется базовым типом для массива.

Вместо имени класса при объявлении переменной используется имя базового типа, после которого идут пустые квадратные скобки, например:

```
int[] a1;
```

При этом размер массива заранее не задается и не является частью типа.

Массивы в Java

Для того, чтобы создать объект типа массив, следует воспользоваться ключевым словом **new**, после чего указать ИМЯ базового типа, а за ним в квадратных скобках число ячеек в создаваемом массиве:

```
int[] array = new int[10];
```

После создания массивы Java всегда инициализированы — в ячейках содержатся нули, поэтому если базовый тип массива примитивный, элементы массива будут нулями соответствующего типа.

Если базовый тип ссылочный — в ячейках будут значения **null**.

Работа с массивами

Ячейки в массиве имеют индексы, которые всегда начинаются с **нуля**.

Длина массива хранится в поле **length**, которое доступно только для чтения – изменять его путем присваивания нового значения невозможно.

Для одновременного задания массива и инициализации значений его элементов применяют следующую запись:

```
int[] arr = new int[] {1, 2, 3, 4, 5};
```

Для копирования всего массива (получения его дубликата) следует применять метод **clone()**, т.к. присваивание имен массивов приведет лишь к присваиванию ссылок на объект-массива:

```
int[] arr2 = arr.clone();
```


Работа с массивами

Служебный класс [Arrays](#) предлагает ряд удобных вспомогательных методов для работы с массивами:

- **copyOf(arr, i)** – получение нового массива, являющегося копией **arr** начиная с позиции **i**.
- **fill(arr, value)** – заполнение массива **arr** значением **value**.
- **equals(arr1, arr2)** – поэлементное сравнение массивов **arr1** и **arr2**.
- Различные вариации метода **sort(arr)** – сортировка массива.

Преимущества и недостатки использования массивов

Преимущества:

- эффективность, т.е. высокая скорость обработки;
- тип элементов определяется во время компиляции;
- могут хранить значения примитивных типов.

Недостатки:

- размер фиксирован и не может изменяться.

«Prod» — не самолет, но его можно уронить.

Бывший коллега (с)



Увидимся на следующей лекции!