**Приложение к встрече 16 котября 2015.**

Темы, обсуждавшиеся на встрече (с комментариями):

1. **Из чего состоит класс?**

Важно помнить несколько следующих вещей:

* По умолчанию классы имеют конструктор без параметров
* Класс представляет из себя не только шаблон для создания объектов. Его статические члены (инициализаторы, поля, методы) представляют собой автономные сущности, доступные по полному имени класса.

**package** java.lang;

*// ...*

**public final class** System {

*// ...*

**public final static** PrintStream ***out*** = **null**;

*// ...*

}

*// Anywhere in the code*

System.***out***.println(**"Hello, World!"**);

1. **Тип String**

Метод intern() повышает производительность сравнения через equals() и делает легитимными сравнения по ссылке между объектами типа String. Но использовать этот метод придётся **крайне редко**, потому что современные компиляторы всё делают за программистов.

System.***out***.println(**"a"** + **"b"** == **"ab"**); *// true*

1. **Видимость переменных**

<http://stackoverflow.com/questions/18417285/variable-visibility-in-java>

В оригинальном вопросе был 2 кусочек кода с спользованием локальной переменной внутри анонимного класса. Почему всё-таки правильно определять переменную final int i?

**public class** Main {  
 **public static void** main(String[] args) {  
 **final int** i = 0;  
 **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 System.***out***.println(i);  
 }  
 }).start();

*// i++?*

}  
}

При создании объекта анонимного класса, все используемые *значения* переменных копируются в конструкторе (который за нас пишет компилятор). Если значение переменной можно изменить во внешнем классе, то его значение на момент исполнения (создание треда, доступ к системе вывода операционной системе занимают большое время) может быть **неактуально**.

Однако, следующий код компилируется. Почему?

**public class** Main {  
 **static int** *i* = 0;  
 **public static void** main(String[] args) {  
 **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 System.***out***.println(*i*);  
 }  
 }).start();  
 }  
}

(Нестатическое поведение аналогично) Здесь шутка заключается в том, что компилятор при создании объекта анонимного класса понимает, что переменная i принадлежит классу **Main** (нестатический контекст – объекту класса **Main**) и обращается к переменной **i**, используя ссылку на внешний класс, которая тоже автоматически копируется в конструкторе анонимного класса. Имея *ссылку*, а не просто *копию* переменной, анонимный класс всегда может получить актуальное значение переменной по ссылке.

*// Static*System.***out***.println(*i*); -> System.***out***.println(Main.*i*);

*// non-static*System.***out***.println(**i**); -> System.***out***.println(Main.**this**.**i**);

1. Циклы

Обещанный код класса, который может быть использован в for each цикле.

**import** java.util.Iterator;  
  
**public class** MyIterableClass<T> **implements** Iterable<T> {  
 T[] **array**;  
  
 **public** MyIterableClass(T[] array) {  
 **if** (array == **null**) {  
 **throw new** IllegalArgumentException(**"This class can't hold a null instance"**);  
 }  
  
 **this**.**array** = array;  
 }  
  
 @Override  
 **public** Iterator<T> iterator() {  
 **return new** MyIterator();  
 }  
  
 **private class** MyIterator **implements** Iterator<T> {  
 **int cursor** = 0;  
  
 @Override  
 **public boolean** hasNext() {  
 **return cursor** < **array**.**length**;  
 }  
  
 @Override  
 **public** T next() {  
 **return array**[++**cursor**];  
 }  
 }  
}

***PECS – будет рассмотрен вместе с Generics***

**public class** Main {  
 **public static void** main(String[] args) {  
 **for** (String arg : **new** MyIterableClass<String>(args)) {  
 System.***out***.println(arg);  
 }  
 }  
}

1. **Главный принцип программирования на Java: безопасность до момента компиляции**

Особо сказать нечего. Вы поймёте со временем, что все ограничения языка – благо. Хотя и новая функциональность – ещё большее благо)

1. **API best practices. Naming conventions, code style + OOP: инкапсуляция.**

Это стоит выучить и знать назубок. Или хотя бы знать, как не нужно писать :)

<http://www.oracle.com/technetwork/java/codeconventions-135099.html>

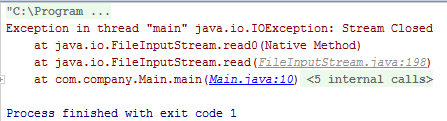
Улучшить свой стиль программирования можно, всего лишь нажимая почаще Ctrl+Alt+L. Если вы видите сообщение: исправлено 0 строк, то поздравляю и так держать.

API = ***a****pplication****p****rogramming****i****nterface.* Хороший апи не предоставляет доступа к лишним методам и переменным, является дуракоустойчивым и интуитивно понятным. В идеале, каждый пакет в java-приложении представляет собой отдельный API. Если код сломается от неправильного порядка вызова методов, от изменения переменных, на состояние которых полагается класс – это плохой дизайн и вина на том, кто разрабатывал API. И да, все методы, вроде close() в пакете java.io, представляют вопиющий пример плохого дизайна.

FileInputStream fis = **new** FileInputStream(**"input.txt"**);  
fis.close();  
fis.read(); *// Exception in thread "main" java.io.IOException: Stream Closed*

Но что поделать, мы не можем (пока что) полагаться на метод finalize(), все ограничения API должны быть хорошо описаны. Конкретно это ограничение связано с необходимостью OS освобождать файлы: <https://en.wikipedia.org/wiki/File_locking>

1. **Исключения**

В идеальном мире ваш код всегда вызван другим java-кодом и это необходимо использовать. Я предлагаю вам полагаться на внешние обработчики исключений. Наример, если ваш код выполняется в IDE:

Здесь ключевым понятием является чейнинг исключений, появившийся в Java 1.4

В классе Throwable появились конструкторы, которые принимают другие Throwable. Таким образом формируется вертикальная иерархия исключений, и именно оборачивание сейчас является рекомендуемой стратегией обработки исключений.

Suppressed-исключения используются крайне редко, могут быть полезны на стадии отладки, при желании отправляйтесь читать гугл или исходник класса Throwable.

1. **Javadoc.**

Javadoc + API => красивая html справка. И ничего больше. Если в требованиях проекта не стоит задачи организовать красивую справочку по публичным интерфейсам, методам, полям и т.д., вам не нужно его писать. Ради интереса вы можете убедиться в том, что содержание всех javadoc в исходных кодах Java **дословно** совпадает с содержимым <http://docs.oracle.com/javase/7/docs/api/>

Если вы хотите объяснить что-то в коде, то использовать обычные комментарии абсолютно допустимо. В идеале, при чтении ваших кодов вопросов вроде: «а что это делает?» возникать не будет. Код должен быть словесным. Существительные представляются классами, прилагательные – интерфейсами, а глаголы – методами. И именно поэтому один код намного лучше другого:

String c = **new** Scanner(**"input.txt"**).next();  
  
**switch** (c) {  
 **case "+"**:**case "-"**:**case "\*"**:**case "/"**:  
 **return true**;  
 **default**:  
 **return false**;  
}

*// vs*

Collection<String> operations = Arrays.*asList*(**"+"**, **"-"**, **"\*"**, **"/"**);  
**return** operations.contains(c);

1. **IDE (Intellij IDEA, Eclipse, NetBeans)**

Резюме: все эти IDE хороши, умеют примерно одно и то же. Пусть у них и разная популярность, у каждой свои минорные плюсы. Рекомендую использовать Intellij потому, что я на ней пишу, люди, которые меня учили и с которыми я работаю, тоже используют Intellij.

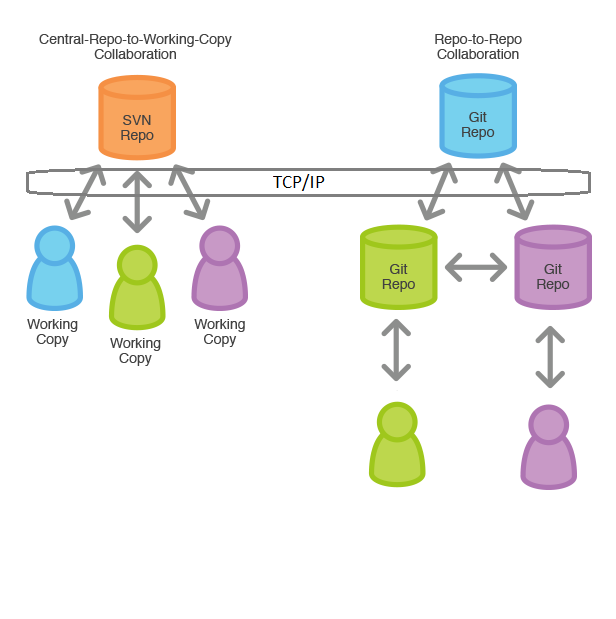
Возможно, NetBeans – лучшая для разработки веб-приложений (так как разрабатывается Oracle). Ну и что? Год назад все использовали Eclipse для разработки на андроид, но недавно гугл объявила, что теперь андроид девелопер студия будет выходить на базе Intellij. В этом мире ничего не постоянно и не надо считать, что есть единый ответ на некоторые вопросы.

1. **CVS (Git vs SVN)**

CVS = система контроля версий.

Основная функциональность: доступ к общему серверу (**checkout**, **pull**, …) и отправка своего кода на сервер (**check-in**, **commit**, **push**, …). Обе системы контроля версий отлично справляются с данной возможностью. Введение разных веток и т.д. являются минорными функциями и сделаны (на мой взгляд) идентично.

Разница проявляется в том, что гит поддерживает офлайн работу с репозиторием, так как всегда создаёт его локальную копию. Но у этого есть и обратная сторона. С репозиторием SVN можно выкачать всего 1 файл и работать с ним :)

  
*That’s all, folks!*