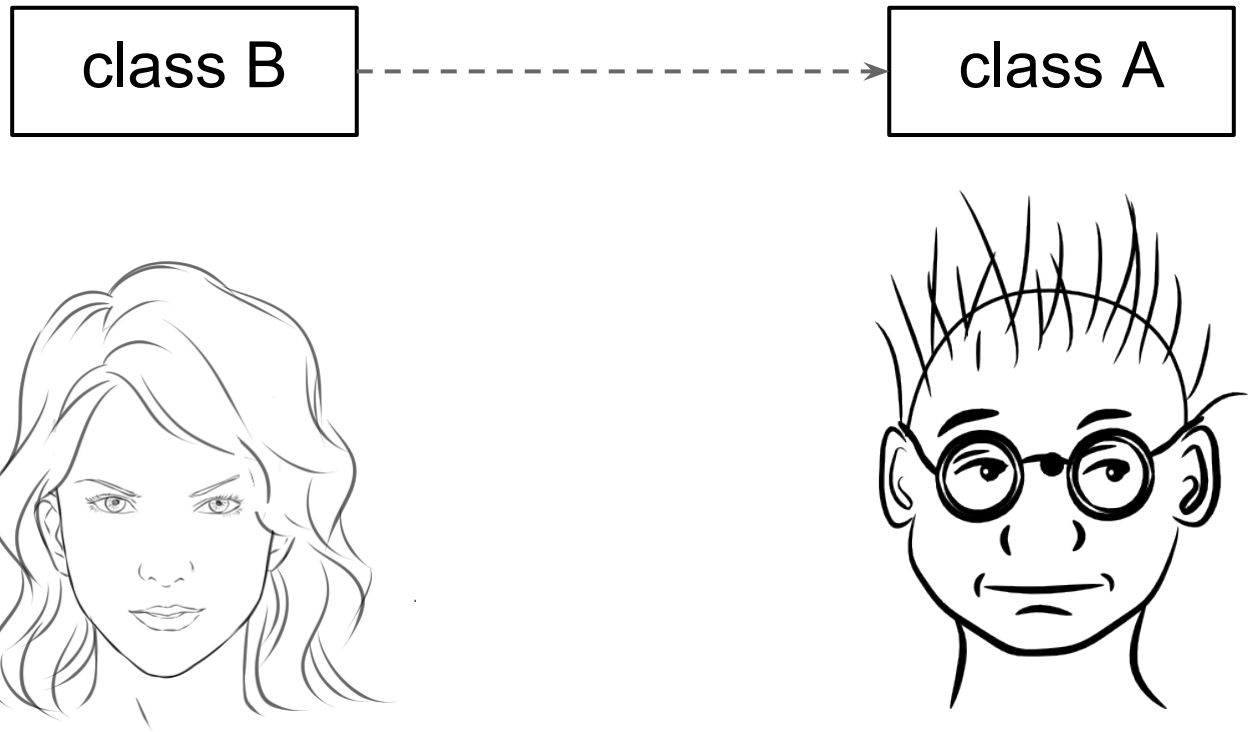
Разработка API в Java-проекте: как оказывать влияние на людей и не приобрести врагов

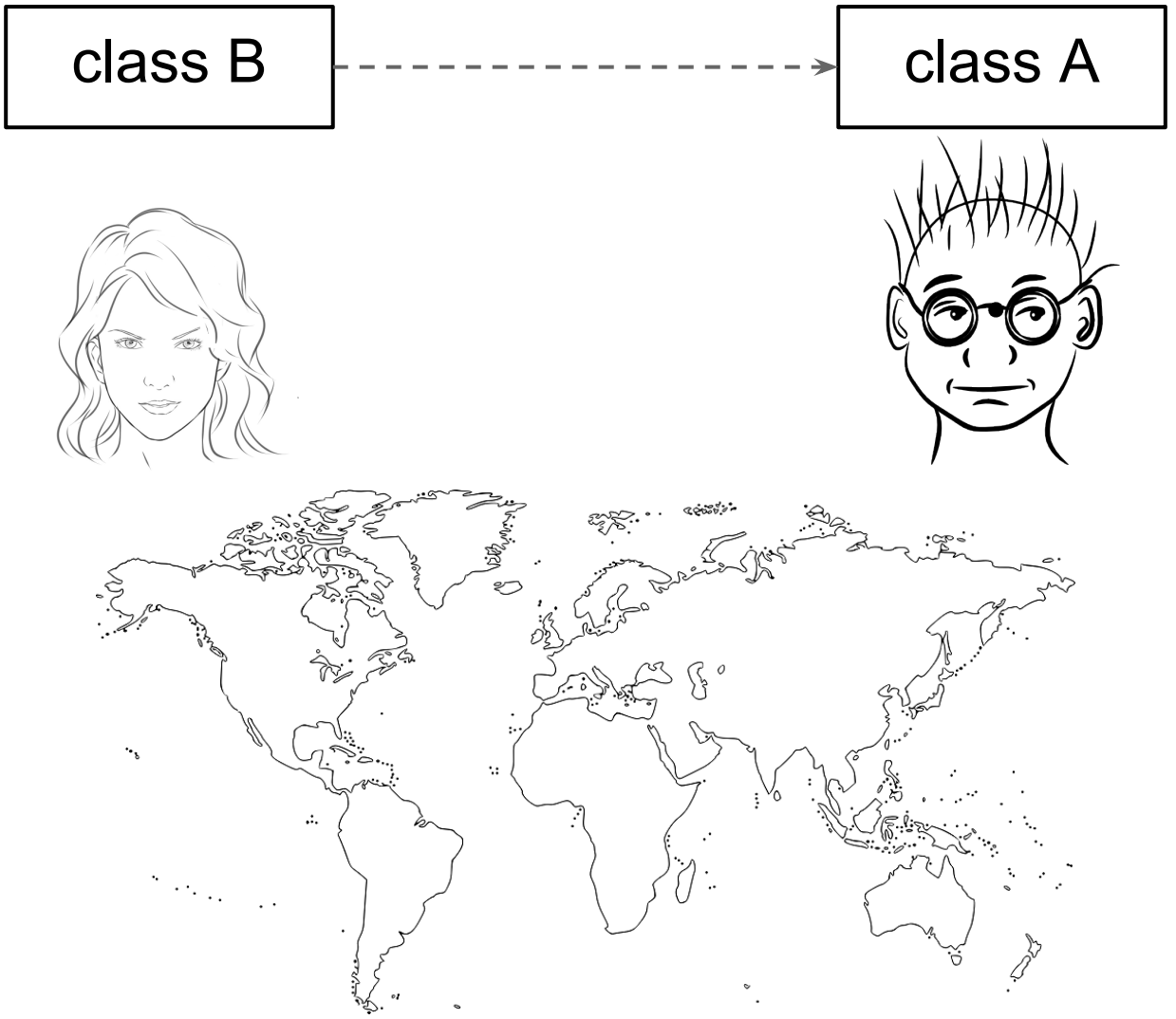
Чашников Николай
программист
JetBrains
Nikolay.Chashnikov@jetbrains.com

Как возникает API

Как возникает API



Как возникает API



К чему приводят недостатки API

API неудачное, если

- его трудно понимать
- его неудобно использовать
- с ним легко допустить ошибку
- его проблематично менять
- оно недостаточно выразительно

Свойства хорошего API

- его легко понимать
- его удобно использовать
- оно защищает от ошибок
- его можно безболезненно менять
- оно достаточно мощное

В API — только необходимое

Добавить в API просто, убрать что-то из API очень трудно.

Нельзя выделять что-то в API просто “чтобы было”, “а вдруг понадобится”.

API не должно содержать реализацию

Все методы классов в API должны быть

- либо abstract
- либо возвращать значение по умолчанию
- либо делегироваться к другим методам API

API не должно содержать реализацию

- отсутствие тел методов облегчает читаемость
- отсутствие тел методов упрощает наследование
- код, вынесенный в API, становится частью контракта

Наследование от классов API

Для каждого интерфейса в API должно быть чётко установлено, предполагается ли его наследование пользователями API.

Разрешать наследование только при необходимости.

Наследование от классов API

```
public abstract class Car {
  private final String manufacturer;
  private final String model;
  public Car(String manufacturer, String model){
    this.manufacturer = manufacturer;
    this.model = model;
  }
  public abstract void playEngineRoar();
  public String getManufacturer()
    { return manufacturer; }
  public String getModel() { return model; }
}
```

Наследование от классов API

```
public class AudiQ7Diesel extends Car {
  public AudiQ7Diesel() {
    super("Audi", "Q7");
  }
  public void playEngineRoar() {
    ...
  }
}
```

Наследование от классов API

```
public interface Car {
  String getManufacturer();
  String getModel();
  CarEngine getEngine();
}
public interface CarEngine {
  void playEngineRoar();
}
public interface CarFactory {
  Car createCar(String model, String
              manufacturer, CarEngine engine);
}
```

Запрещать всё, что не разрешено

- использовать наиболее закрытые модификаторы доступа
- делать final всё, что не предполагается переопределять

Ешьте своё API сами. Дважды

Параллельно с созданием API надо писать минимум два примера его использования.

Мощь и простота использования

API должно обладать достаточной мощью для решения поставленных задач.

Но решение типичных задач должно быть простым.

Как проще всего прочитать в String текст файла (в Java 6)?

```
FileInputStream input =
        new FileInputStream(file);
byte[] bytes =
     new byte[(int) file.length()];
input.read(bytes);
input.close();
return new String(bytes, "UTF-8");
```

В Java 7 это стало проще

```
return new String(
    Files.readAllBytes(file.toPath()),
    StandardCharsets.UTF_8);
```

Упрощайте доступ к нужным методам

```
public interface Form {
  /** use {@link Validators} */
  void check();
}

public class Validators {
  public static boolean isValidEmail(String s)
{…}
  public static boolean isValidPhoneNumber
                              (String s) { … }
}
```

Упрощайте доступ к нужным методам

```
public interface Form {
  void check(Validators validators);
}

public interface Validators {
  boolean isValidEmail(String s);
  boolean isValidPhoneNumber(String s);
}
```

Защита от ошибок

- в случае ошибки падать как можно раньше
  - в идеале — при компиляции
- использовать аннотации @NotNull, @Nullable.
- не использовать String, если есть более подходящий тип
- использовать generics вместо casts и Object

Слабая типизация

```
public interface AssortedData {
   void putData(String key,
                Object data);
   Object getData(String key);
}
```

Сильная типизация

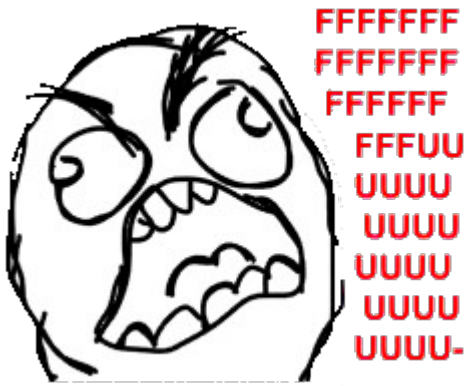
```
final class Key<T> {}

public interface AssortedData {
   <T> void putData(Key<T> key,
                    T data);
   <T> T getData(Key<T> key);
}
```

Как при использовании этого класса можно ошибиться?

```
public class Shop {
  private List<Order> orders
                              = new ArrayList<Order>();
  private Order[] ordersArray;
  public void addOrder(Order order)
    { orders.add(order); ordersArray = null; }
  public Order[] getOrders() {
    if (ordersArray == null)
      ordersArray =
           orders.toArray(new Order[orders.size()]);
    return ordersArray;
  }
}
```

Например, вот так

```
Order[] orders = shop.getOrders();
…
…
Arrays.sort(orders);
```



А вот как от этого защититься

```
public class Shop {
  private List<Order> orders =
                              new ArrayList<Order>();
  private List<Order> unmodifiableOrders =
            Collections.unmodifiableList(orders);
  public void addOrder(Order order) {
    orders.add(order);
  }
  public List<Order> getOrders() {
    return unmodifiableOrders;
  }
}
```

Тот номер больше не пройдёт

```
List<Order> orders = shop.getOrders();
…
…
Collections.sort(orders);
//вылетит UnsupportedOperationException
```

Защита от ошибок

- избегать неявных ограничений
- использовать immutable объекты

Навязывать нужное поведение

```
public class DoNotOverrideEquals {
  public final boolean equals(Object obj) {
    return super.equals(obj);
  }

  public final int hashCode() {
    return super.hashCode();
  }
}
```

Навязывать нужное поведение

```
public abstract class MustOverrideEquals {
  public abstract boolean equals(
                                Object obj);

  public abstract int hashCode();
}
```

Изменение API

При изменении методов интерфейсов, от которых не наследуется пользователь, оставлять старые варианты с пометкой deprecated.

Изменение API: наследование

```
public interface Cat {
   void stroke();
}
```

Как добавить параметр в метод, не ломая совместимости?

Изменение API: наследование

```
/** @deprecated
              implement Cat2 instead */
public interface Cat {
   void stroke();
}

public interface Cat2 extends Cat {
   void stroke(double force);
}
```

Изменение API: наследование

```
if (cat instanceof Cat2) {
   ((Cat2)cat).stroke(0.5);
}
else {
   cat.stroke();
}
```

Изменение API: наследование

```
public abtract class AbstractCat
                          implements Cat {
/** @deprecated override
            stroke(double) instead */
  public void stroke() {}
  public void stroke(double force){
     stroke();
   }
}
```

Изменение API: наследование

Безопасное расширение интерфейсов, от которых наследуется пользователь, возможно только в случае, если у них есть базовый класс, от которого все наследуются.

Ждём Java 8 и default methods в интерфейсах.

API в виде Internal DSL

Internal DSL (Domain Specific Language) — код на Java, который не похож на обычный Java-код

Обычные сеттеры

```
SearchQuery query =
                    new SearchQuery("java");
query.setSite("http://oracle.com");
query.setSortingCriteria(
                  SortingCriteria.RELEVANCE);
query.setMaximumResults(10);
SearchResult result = runSearch(query);
```

DSL на билдерах

```
SearchResult result =
    search("java")
      .onSite("http://oracle.com")
      .sortByRelevance()
      .returnFirst(10)
      .run();
```

DSL на билдерах: реализация

```
public class SearchQuery {
  private final String text;
  private String site;

  private SearchQuery(String text) { this.text = text; }
  public static SearchQuery search(String text) {
    return new SearchQuery(text);
  }
  public SearchQuery onSite(String site) {
    this.site = site;
    return this;
  }
  ...
}
```

DSL на билдерах: сложный случай

Указание подходящего места в синтаксическом дереве в IntelliJ IDEA: http://tinyurl.com/n2ophbu

DSL для описания XML

```
<web-app version="2.5">
  <servlet>
    <servlet-name>Sample</servlet-name>
    <servlet-class>...</servlet-class>
  </servlet>
</web-app>
```

DSL на интерфейсах

```
public interface WebApp {
   Servlet addServlet();
   List<Servlet> getServlets();
}
public interface Servlet {
   String getServletName();
   void setServletName(String name);
}
```

DSL на интерфейсах: реализация

```
java.lang.reflect.Proxy.newProxyInstance
   (classLoader, interfaces, handler)

public interface InvocationHandler {
    Object invoke(Object proxy, Method
                  method, Object[] args)
        throws Throwable;
}
```

DSL на интерфейсах: реализация

AdvancedProxy в IntelliJ IDEA platform:

http://tinyurl.com/ohnto9q

Вопросы?

Если возникнут позже, пишите

Nikolay.Chashnikov@jetbrains.com