08.
Miscellaneous

Информатика, ИТИС, 2 курс
М.М.Абрамский
2016

Вспоминаем. Структура проекта на Servlets (как должно быть)

```
MyHelloProject
    css
        style.css
    WEB-INF
        classes
      …иерархия пакетов…
      HelloServlet.class
        web.xml
```

Кто формирует эту структуру, чтобы все заработало?

Сравните с исходниками

MyHelloProject
- **src**
  - *…иерархия пакетов…*
  - HelloServlet.java
- **web**
  - **css**
    - style.css
  - **WEB-INF**
    - web.xml

Можно вручную

1. Вручную компилируем все java-файлы,
2. В папке webapps сервера Tomcat создаем описанную структуру, копируя туда .class-файлы,
3. ???
4. Profit!!!

Можно вручную

1. Вручную компилируем все java-файлы,
2. В папке webapps сервера Tomcat создаем описанную структуру, копируя туда .class-файлы,
3. ???
4. Profit!!!

Какие минусы такого подхода?

Автоматизация сборки

Автоматизация описанных процессов (компиляция, тестирование, развертывание и т.п.) ускоряет работу, избавляет от человеческого фактора, и т.д.

Сборщики:

- Ant
- Maven
- Gradle
- ! ...

Сборка. Избирательная терминология

- **Artifact** – конкретная библиотека / созданный экземпляр проекта.
- **WAR-файл** – упакованное веб-приложение, готовое к деплою (по аналогии с jar),
- **Деплой (deploy)** – развертывание свежей версии рабочих файлов приложения на сервере.
- **Зависимость (dependency)** – использование сторонней библиотеки определенной версии в Java-приложении

Apache Ant

- “Another Neat Tool”,
- Аналог *make*,
  - !google **make**
- Императивный подход,
  - ? Что это такое?
- Скрипт пишется на XML.

Apache Ant

- Targets – цели (какой именно процесс сборки выполняется),

  Примеры:
  - *build – компиляция и создание jar/war,*
  - *clean – удаление временных файлов,*
  - *deploy – развертывание,*
  - *и т.п.*

- Tasks – задания, выполняемые в рамках целей

  Примеры:
  - *javac – компиляция java-файлов,*
  - *copy – копирование файлов,*
  - *exec – выполнение внешней команды,*
  - *и т.п.*

Apache Ant. Отрывки

```
<target name="compile" depends="prepare"
        description="Compile the servlet">
    <echo message="Compiling the Java file "/>
    <echo message="${compiled.servlet}.java..."/>
    <javac srcdir="${src}" destdir="${build}">
        <include name="${compiled.servlet}.java" />
        <classpath refid="servlet-classpath"/>
    </javac>
</target>
        ….
<target name="deploy-servlet" depends="compile">
<echo message="Copying the servlet to Tomcat web app"/>
<copy todir="${tomcat.webapps}/WEB-INF/classes">
    <fileset dir="${build}" />
</copy>
</target>
```

Apache Maven

- Maven - “Собиратель знания” (идиш),
- Декларативный подход,
  - ? Что это такое?
- Сборка на основе описания структуры проекта на языке XML.

Apache Maven. Project Object Model. *pom.xml (Wikipedia example)*

```
<project><modelVersion>4.0.0</modelVersion>
    <!-- набор значений, позволяет идентифицировать этот проект
(координаты проекта) -->
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1.0</version>
    <dependencies> <!-- зависимости от библиотек -->
        <dependency>
            <!-- координаты необходимой библиотеки -->
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <!-- только для запуска и компилирования тестов -->
            <scope>test</scope>
        </dependency>
    </dependencies></project>
```

Apache Maven. Плагины

Непосредственно выполняют необходимые задачи

mvn имя_плагина:имя_цели

mvn compiler:compile
mvn archetype:generate

Apache Maven. Жизненный цикл

1. Создание по образцу (archetype),
2. Компиляция (compile),
3. Тестирование (test),
4. Упаковка (package),
5. Локальное развертывание (install),
6. Удаленное развертывание (deploy).

Apache Maven. Архетипы

Позволяют создавать проект с нужной структурой и заголовками конфигурационных файлов

Пример вызова плагина для создания проекта по архетипу:

```
mvn archetype:create -DgroupId=com.mycompany.app
-DartifactId=my-webapp
-DarchetypeArtifactId=maven-archetype-webapp
```

Примерный pom.xml для нашего приложения

```
<project xmlns="http://maven.apache.org/POM/4.0.0" … >
      <modelVersion>4.0.0</modelVersion>
      <groupId>servlet-hello</groupId>
      <artifactId>servlet-hello</artifactId>
      <packaging>war</packaging>
      <version>1.0</version>
      <dependencies>
          <dependency>
              <groupId>javax.servlet</groupId>
              <artifactId>servlet-api</artifactId>
              <version>2.5</version>
          </dependency>
      </dependencies>
      <build>
          <finalName>servlet-hello</finalName>
      </build>
</project>
```

Maven-структура нашего приложения

```
src
    main
        java
            …иерархия пакетов…
            HelloServlet.java
        …
        webapp
            WEB-INF
                web.xml
    …
    pom.xml
```

“mvn package” создаст war-файл, с которым можно делать deploy или explode.

ХАРДКОД!

Хардкод (Hardcode). Случай с числами

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    int[] a = new int[42];
    for (int i = 0; i < 42; i++) {
        a[i] = scanner.nextInt();
    }
    for (int i = 0; i < 42 / 2; i++) {
        a[i] = a[42 - i - 1];
    }
}
```

Хардкод. Еще хуже

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    int[] a = new int[42];
    for (int i = 0; i <= 41; i++) {
        a[i] = scanner.nextInt();
    }
    for (int i = 0; i < 21; i++) {
        a[i] = a[41 - i];
    }
}
```

Константы как частный способ решения проблемы

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    final int SIZE = 42;
    int[] a = new int[SIZE];
    for (int i = 0; i < SIZE; i++) {
        a[i] = scanner.nextInt();
    }
    for (int i = 0; i < SIZE / 2; i++) {
        a[i] = a[SIZE - i - 1];
    }
}
```

Более частый случай – строковый хардкод

```
if (person.getGender().equals("Мужской")) {
    ...
}
...
if (direction.getName().equals("Вверх")) {
    ...
}
...
if (season.getName().equals("Лето")) {
    ...
}
```

Одно из решений – строковые константы

```
final String SUMMER = "Summer";
final String MALE_GENDER = "Male";
```

Проблемы:

- название и значение дублируют друг друга,
- где хранить, чтобы обращаться?
- как быть с Summer и SUMMER?
- ...

Другое решение

```
class Season {
    final static int WINTER = 0;
    final static int SPRING = 1;
    final static int SUMMER = 2;
    final static int FALL = 3;
}
```

Проблемы:

- Откуда знать весь диапазон значений и как его перебрать?
- Если x == 0, то Season.WINTER == x, но действительно ли корректно считать левую переменную x хранящей значение «Зима»

Итак

Нужен тип данных:

- Чтобы у переменных этого типа явно было видно значение,
- Чтобы можно было легко перебрать все его значения,
- *Чтобы не хардкодить,*

Он есть! И это...

Перечисления (Enumerations)

Объявление:

```
enum Season {
    WINTER, SPRING, SUMMER, FALL
}
```

Использование:

```
Season s = Season.SPRING;
```

Решаем проблемы.
Перебираем с помощью `values()`

- `values()` возвращает массив из всех значений перечисления

```
for (Season season: Season.values()) {
    System.out.println(season);
}
```

Решаем проблемы. Сравнивать можно только с другими значениями перечисления

```
Season season = Season.SUMMER;
...
if (season == Season.WINTER)
    System.out.println("NEW YEAR!");

// у каждого есть свой порядковый номер
// выведет 3
System.out.print(Season.FALL.ordinal());
// но вот такое сделать не получится
if (season == 3) {
    ...
}
```

Решаем проблемы. Ввод

- Значение можно восстановить по строке
  - Надо вводить строку с точностью до регистра!

```
// Прокатит
Season season = Season.valueOf("WINTER");
...
// Не прокатит
Season season = Season.valueOf("Winter");
```

Все гораздо интереснее

- Вы думаете, эти WINTER, SUMMER – просто константы?

- А вот и нет! Это объекты!

Другой enum. Цвет

```
enum Color {
    RED, GREEN, BLUE, WHITE, BLACK
}
```

У каждого цвета есть значения RGB.

Наша потребность:

- Чтобы каждый цвет знал свои значения,
- Чтобы каждый цвет мог возвращать строку-представление RGB

Для этого изменим enum.

«В новом цвете»

```
enum Color {
    RED(255, 0, 0), GREEN(0, 255, 0),
    BLUE(0, 0, 255), WHITE(255, 255, 255),
    BLACK(0, 0, 0);

    private int r, g, b;
    Color(int r, int g, int b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
    public String getRGBValues(){
        return "(" + r + "," + g + "," + b +")";
    }
}
```

«В новом цвете». Использование

```
Color color = Color.BLACK;
System.out.println(color.getRGBValues());
```

META

Слово «Мета»

Греческое слово
μετά
«между, через, после, за, следующее»

В: Что такое метаданные?

Слово «Мета»

Греческое слово
μετά
«между, через, после, за, следующее»

В: Что такое метаданные?
О: Данные о данных.

Метаданные в программах

- Не влияют на непосредственную работу программы,
- Но могут быть выявлены другими программами на этапе компилирования или разработки, которые при этом скорректируют свою работу.

```
class MyThread extends Thread {

    public void run(boolean alive) {
        System.out.println("THREAD IS COMING! ");
    }
    public static void main(String[] args) {
        (new MyThread()).start();
    }
}
```

```
class MyThread2 extends Thread {

    public void run() {
        System.out.println("THREAD IS COMING! ");
    }
    public static void main(String[] args) {
        (new MyThread2()).start();
    }
}
```

Чем различаются эти случаи? В чем возможная ошибка?

С MyThread все было бы в порядке, если бы применили..

```
class MyThread extends Thread {

    @Override
    public void run(boolean alive) {
        System.out.println("THREAD IS COMING! ");
    }
    public static void main(String[] args) {
        (new MyThread()).start();
    }
}
```

- Компилятор бы просто не скомпилировал эту программу, т.к. метод, над которым написано @Override, не является переопределением.
- Увидев ошибку компилятора, мы бы исправили сигнатуру

Заметка про Override

- Нужда для программиста, а не для программы
- Запрещает компилирование, но при этом никак не влияет на выполнение метода (при правильном случае что она есть, что ее нет)
- Override – аннотация.
  - А аннотации – это и есть метаданные.

Про аннотации

- Не влияют напрямую на работу кода, но могут быть обнаружены другими средствами
- Могут быть аннотированы класс, метод, параметр, атрибут и т.д.
- Другие примеры аннотаций?
  - @Deprecated
  - @SuppressWarnings

Создание собственных аннотаций

Самая простая

```
@interface MyAnno {}
```

Использование:

```
@MyAnno
class MyClass {
    // ...
}
```

Методы-члены аннотации

- Объявляются как методы:

```
@interface Author {
    String name();
    int year();
}
```

- Но используются как поля:

```
@Author(name="Tony Stark", year=1996)
class MyClass {
    // ...
}
```

Значения по умолчанию

- *Внимание на year:*

```
@interface Author {
     String name();
     int year() default 2000;
}
```

- Теперь можно делать и так,

```
@Author(name="Tony Stark", year=1996)
class MyClass { …
```

- И так:

```
@Author(name="Tony Stark")
class MyClass { …
```

Аннотации, аннотирующие аннотации (лежат в java.lang.annotation)

@Retention – политика удержания аннотации (*по-деревенски: до какого этапа компилирования или выполнения аннотация видна)*

Значения лежат в перечислении RetentionPolicy:

- SOURCE – отбрасываются при компиляции
- CLASS – сохраняются в байт-коде, но недоступны во время работы
- RUNTIME – сохраняются в байт-коде и доступны во время выполнения

? Какой Retention у Override?

Аннотации, аннотирующие аннотации (лежат в java.lang.annotation)

@Target – к чему может быть применена аннотация? Значения – из перечисления **ElementType** (из того же пакета)**:**

- FIELD – поле
- METHOD – метод
- TYPE – класс, интерфейс, перечисление
- …

- Может применяться к нескольким:

```
@Target({ElementType.TYPE, ElementType.METHOD})
```

Аннотации, аннотирующие аннотации

Чтобы наш Author был доступен во время работы и применялся к объявлениям класса, интерфейса:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface Author {
    String name();
    int year() default 2000;
}
```

ВСПОМНИМ ООП

Вспомним ООП.
Что есть у каждого класса

- Название класса
- Название пакета
- Атрибуты
- Методы
- ?...

Еще раз

Класс:

- Имя
- Имя пакета
- Набор атрибутов
- Набор методов
- ...

In English, please

Class:

- name
- package name
- List of attributes
- List of methods
- ...

Со шрифтом “Courier New” выглядит «по-программистски»

```
Class:
  name
  packageName
  List attributes
  List methods
  …
```

Wait, what?

```
class Class {
   String name;
   String packageName;
   List<Attribute> attributes;
   List<Method> methods;
   …
}
```

- Получается, Класс (**Class**) – тоже сущность (а сущность – это **класс**);

- А **все** конкретные **реализованные классы** *(String, User, ComplexNumber – ДА ВСЕ)* – **экземпляры класса Class**.

- Значит, все инструменты ООП мы можем применить к самим классам как к сущностям.
    - Это и называется рефлексией!

Класс Class

- Служебный класс, экземпляры которого хранят конкретную информацию о конкретном классе.
  - Объект класса Class для String, объект класса Class для Thread и т.п.
- Уже реализован в Java (Reflection API)

Как узнать свой класс?

- Объекту

(*пусть **obj** – экземпляр класса **MyClass***):

```
Class c = obj.getClass();
```

- Классу

(*пусть это **MyClass***):

```
Class c = MyClass.class;
```

- Названию класса

(*пусть полное имя класса: **org.kpfu.UseClass***):

```
Class c = Class.forName("org.kpfu.UseClass");
```

О-па!

- Экземпляры класса, представимого объектом класса Class, можно создавать с помощью getInstance

- `String type = scanner.next();`
- `Class c = Class.forName(type);`
- `Object o = c.newInstance();`

Параметризация

- Вообще говоря, Class параметризован
  - Не Class, а Class<T>
- Но если знать тип заранее, весь кайф от зависимости типа данных от входа пропадает.

Параметризация

```
Class<String> c =
Class.forName(интересно_какой_же_сюда_
мы_можем_вставить_класс_неужели_String
_вот_это_неожиданность);
String s = c.newInstance();
```

– бред, чего сразу String не использовал?

Параметризация

А вот так – больше возможностей:

```
String type = scanner.next();
Class c = Class.forName(type);
Object o = c.newInstance();
//тип неизвестен заранее
```

Да, экземпляры с будут Object, но мы можем в принципе вызвать instanceof – и все будет ОК.

```java
@Author(name="Smart Programmer", year=2015)
class Vector2D {
    private double x, y;

    public double getX() { return x; }
    public void setX(double x) { this.x = x; }
    public double getY() { return y; }
    public void setY(double y) { this.y = y; }

    public Vector2D(){
        x = 0;
        y = 0;
    }
    public Vector2D(double x, double y){
        this.x = x;
        this.y = y;
    }

    public Vector2D add(Vector2D v) {
        return new Vector2D(x + v.getX(), y + v.getY());
    }
}
```

Это тоже классы!

- Method
- Field
- Constructor
- Annotation
- Type
- Package

если Class – сущность (класс), то почему они не могут быть классами?

? Какие атрибуты Field? Method?

Получить все методы

```
Class cv = Vector2D.class;
Method[] methods = cv.getMethods();
for (Method method : methods) {
    System.out.println(method.getName());
    System.out.println(method.getReturnType());
    System.out.println(
     Arrays.toString(
        method.getParameterTypes()
     )
    );
}
```

Получить все методы

add
class Vector2D
[class Vector2D]
setY
void
[double]
setX
void
[double]
getX
double
[]
getY
double
[]
hashCode
int
[]
getClass
class java.lang.Class
[]
equals
boolean
[class java.lang.Object]
toString
class java.lang.String
[]
wait
void
[long, int]
wait
void
[long]
wait
void
[]
notify
void
[]
notifyAll
void
[]

Получить все поля

```
Class cv = Vector2D.class;
Field[] fields = cv.getFields();
for (Field field : fields) {
    System.out.println(field.getName());
    System.out.println(field.getType());
}
```

Получить все поля

```
Class cv = Vector2D.class;
Field[] fields = cv.getFields();
for (Field field : fields) {
    System.out.println(field.getName());
    System.out.println(field.getType());
}
```

Кстати, тут ничего не выведется.

Declared

- Рефлексия учитывает инкапсуляцию, хотя может и игнорировать ее
  - getDeclaredMethod(), getDeclaredMethods(), getDeclaredFields() и др. методы с Declared в названии возвращают все соответствующие сущности, вне зависимости от модификатора,
  - Аналогичными методами без Declared будут возвращаться только public-сущности.

Получить все поля

```
Class cv = Vector2D.class;
Field[] fields = cv.getDeclaredFields();
for (Field field : fields) {
    System.out.println(field.getName());
    System.out.println(field.getType());
}
```

Вывод:

x
double
y
double

Да, кстати, проверка Аннотаций

```
Class cv = Vector2D.class;
Annotation[] annotations = cv.getAnnotations();
for(Annotation annotation : annotations){
    if(annotation instanceof Author){
        ...
    }
}
```

Проверяем, что Vector2D аннотирован @Author

Самый экшн

у Класса:

- getMethod(…) – возврат метода по **сигнатуре**;
- getConstructor(…) – возврат конструктора по **сигнатуре**;

у Метода:

- invoke() – вызов метода

Сигнатура в терминах рефлексии

- “Имя и набор типов параметров”
- String и массив объектов класса Class

```
Class cs = String.class;
Method m = cs.getMethod(
   "indexOf",
   new Class[]{String.class, int.class}
);
```

! Java varargs

Reflection in action!

```
Scanner scanner = new Scanner(System.in);

Class cv = Class.forName(scanner.next());
Class cv2 = Class.forName(scanner.next());

String methodName = scanner.next();
Method m = cv.getMethod(methodName, cv2);

Object o1 = cv.newInstance();
Object o2 = cv2.newInstance();

// вызываю у o1 метод m (с именем methodName)
// на объекте o2
System.out.println(m.invoke(o1, o2));
```

```
Scanner scanner = new Scanner(System.in);
Class cv = Class.forName(scanner.next());
Class cv2 = Class.forName(scanner.next());
String methodName = scanner.next();
Method m = cv.getMethod(methodName, cv2);
Object o1 = cv.newInstance();
Object o2 = cv.newInstance();
System.out.println(m.invoke(o1, o2));
```

- Работает, если я подам на вход:
  - **Vector2D Vector2D add**

    т.к. в Vector2D есть add(Vector2D)
  - **java.util.HashSet int add**

    т.к. в HashSet есть add(Object)
  - **java.lang.Thread java.lang.String setName**

    т.к. в java.lang.Thread есть setName(String)

IMPORTANT!

Я могу управлять работой программ гибко, на разных классах, не переписывая их и не компилируя каждый раз заново!”

Это легло в основу многих
java-фреймворков,
в частности ***Spring, Hibernate и др.***

Рефлексия в других языках

- В Java обычный класс и объект класса Class, соответствующий обычному классу – разные сущности
- В Python, например, это одно и то же:

```
class Pet:
   pass
```

Объявил одновременно и класс Pet, и экземпляр класса Class, соответствующий Pet.

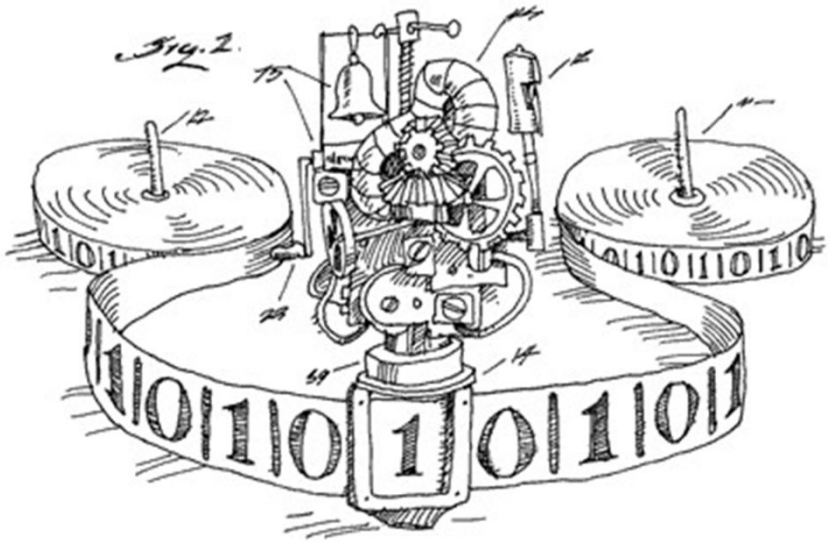
Могу внутри него писать методы для Pet как обычного класса,

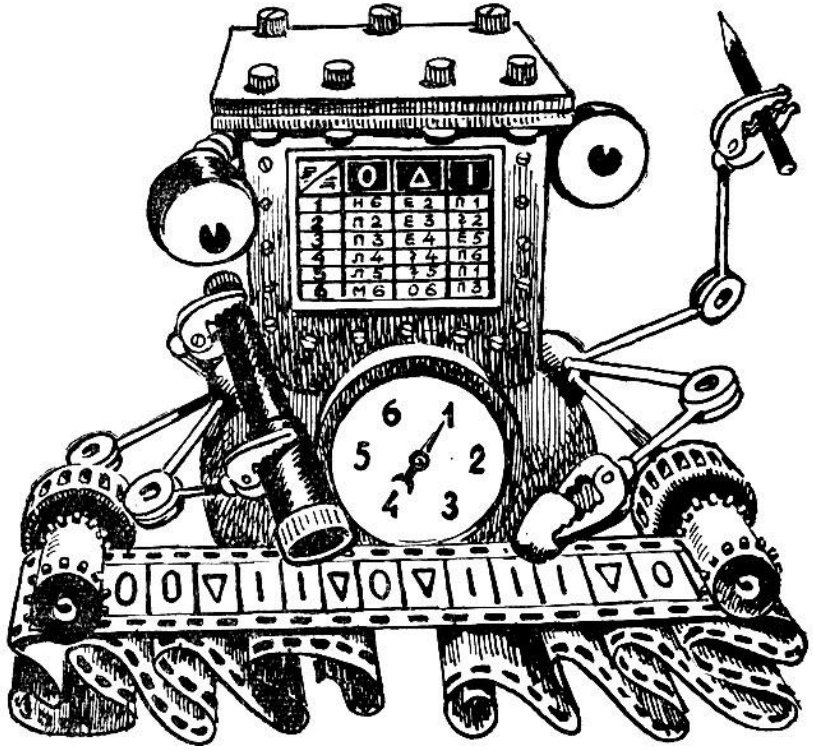
Могу для Pet как для объекта класса Class (class methods)

СУЩЕСТВОВАНИЕ ПРОГРАММНОЙ ИНЖЕНЕРИИ (ИЗ ЛЕКЦИЙ ДЛЯ 1 КУРСА)

Машина Тьюринга (МТ)

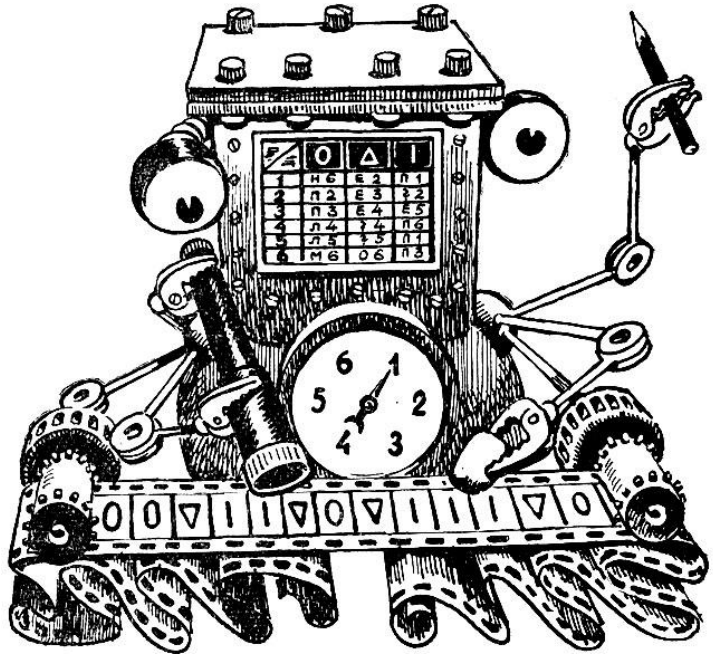
- Алан Тьюринг, 1936
- *Абстрактная модель вычислительного устройства – вычисление любой функции*





Устройство МТ

- Алфавит
- Состояния (память)
- Лента (*бесконечная*)
- Считывающая головка
- Программа



	s1	s2
0	s1 →	1 stop
1	s1 →	0 s2 ←
^	s2 ←	1 stop

Это функция f(x) = x + 1

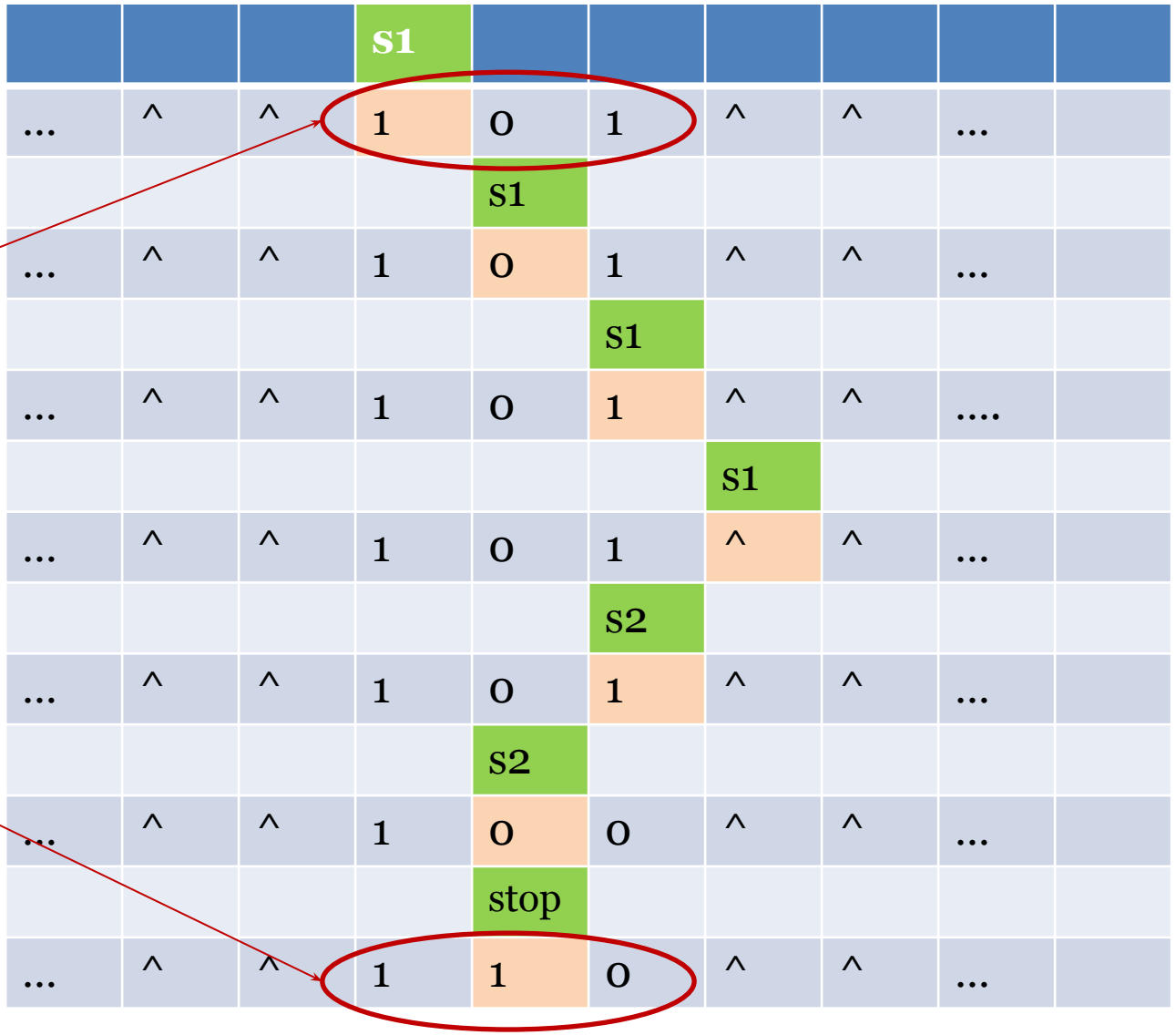
Работа Машины Тьюринга

	s1	s2
0	s1 →	1 stop
1	s1 →	0 s2 ←
^	s2 ←	1 stop

Это функция f(x) = x + 1

Вход (аргумент функции, которую реализует МТ)
101 – двоичный код числа 5.

Выход, результат работы, значение функции f(x) = x + 1
110 – двоичный код числа 6.



Тезис Чёрча-Тьюринга

- Любой интуитивно-вычислимый алгоритм может быть реализован на машине Тьюринга.
- Другие формальные модели, удовлетворяющие этому тезису, называются **Тьюринг-полными**.
- Написание программ для машины Тьюринга – ***программирование.***

Программа МТ - данные

- Можно выписать в текст и занумеровать – превратить в цифровую информацию (код машины Тьюринга)
  - Выписываем поклеточно, # - разделитель информации о клетках:
    - 0, 1, s1, -> # 0, s2, 1, stop # …
    - Текст можно закодировать.

	s1	s2
0	s1 →	1 stop
1	s1 →	0 s2 ←
^	s2 ←	1 stop

- Этот код можно подать на вход другой машине Тьюринга

Универсальная машина Тьюринга

- Машина Тьюринга, моделирующая работу других МТ
  - На вход подают код другой МТ и входные данные, универсальная МТ выдает ответ, как если бы работала эта другая МТ
    - ! Универсальная функция – аналог

- Теорема о существовании универсальной машине Тьюринга: **универсальная машина Тьюринга *существует*!**
  - **И это то, без чего не было бы сегодняшнего цифрового мира.**

Объяснение

- МТ – модель вычислительного устройства, решающего конкретную задачу (вычисляющую конкретную функцию)

- Но если взять универсальную МТ – и ей на вход подавать код программы других машин Тьюринга – мы сможем выполнять на одном устройстве все возможные алгоритмы.
  - главное – уметь писать программы!

- Ничего не напоминает? Одно устройство, много алгоритмов, код программы…

Ура!

Теорема о существовании универсальной машины Тьюринга – обоснование наличия программирования как деятельности!

– Нам не нужно строить кучу разных устройств для каждого алгоритма!

– У нас будет один (computer), на котором мы будем выполнять программы, записанные на определенном языке (код программы)
– Язык, на котором пишут программы – **язык программирования**!

Связь универсальности, Тьюринг-полноты и рефлексии

- Рефлексия в языке – признак его тьюринг-полноты
  - «На языке можно написать его компилятор»
  - «Язык позволяет создавать свои конструкции своими же средствами»

! *Аналог теоремы об универсальной МТ – теорема об универсальной функции (для любителей серьезной алгоритмической математики)*

Прочитать

- http://www.quizful.net/post/java-reflection-api (rus)
- http://tutorials.jenkov.com/java-reflection/methods.html (eng)