**Цент компьютерного обучения**
**“Специалист”**
**при МГТУ им. Н.Э. Баумана**

Скрынченко О.А.

# Adobe Flash CS4

## Уровень 2. Интерактивная анимация.

## Программирование на ActionScript 2.0


Учебное пособие к курсу

Москва
2009

**Скрынченко О.А.**

Учебное пособие к курсу
Adobe Flash CS4. Уровень 2. Интерактивная анимация.
Программирование на ActionScript 2.0

**Уважаемый слушатель!**

Вы начинаете изучение курса *«Adobe Flash CS4. Уровень 2. Интерактивная анимация. Программирование на ActionScript 2.0».* Данный курс поможет добавить **новые возможности** Вашим Flash-проектам. Вы научитесь использовать программную анимацию и освоите методы загрузки внешних мультимедиа ресурсов.

При изучении данного курса Вам будут полезны знания других языков программирования, особенно на таких как C++, JavaScript или JScript.

Наш курс состоит из 6 занятий по 4 академических часа каждое. В процессе обучения Вы будете не только осваивать теоретический материал, но и выполнять разнообразные практические задания. Для закрепления полученных знаний и навыков каждое занятие содержит задания для самостоятельного выполнения. Ваша самостоятельная работа – залог успешного освоения курса!

Язык ActionScript 2.0 дает базу для создания интерактивных flash-приложений.
Следующая версия ActionScript 3.0, позволяет повысить эффективность Ваших проектов.

*Желаю успеха!*

# Что находится на раздаточном диске

В учебный комплект включен компакт-диск, содержащий материалы упражнений для аудиторной и самостоятельной работы.

Все файлы разделены по занятиям. Материалы каждого занятия находятся в отдельной папке:

- `lesson_01`
- `lesson_02`
- `lesson_03`
- `lesson_04`
- `lesson_05`
- `lesson_06`

Внутри папки каждого занятия находится папка «`Practice_Work`», содержащая файлы необходимые для выполнения самостоятельных работ. А также папка «`Samples`», в которой находятся дополнительные примеры готовых flash-роликов по теме занятия.

В папке «`Documentation`» находятся официальные руководства по программированию на языке ActionScript 2.0 на английском языке. А так же файл «`Links_Books_Flash.doc`», содержащий ссылки на полезные Интернет-ресурсы и литературу, и русскоязычная справка по языку ActionScript 2.0.

# Оглавление

# Занятие 1

*Продолжительность 180 минут*

**Тема занятия:**

Первое знакомство с ActionScript 2.0

# Первое знакомство с ActionScript 2.0

## Язык ActionScript

**ActionScript** - это объектно-ориентированный язык программирования, используемый в средах выполнения flash-роликов (Flash Player и Adobe AIR). Он обеспечивает интерактивность, обработку данных и многие другие возможности при создании flash-роликов и flash-приложений.

ActionScript дословно означает «*язык действий*». Он появился в 2000-ном году с выходом версии Macromedia Flash 5. До этого в программе Flash имелось лишь несколько скриптовых команд для управления воспроизведением ролика.

Язык ActionScript основан на спецификации **ECMAScript**. Поэтому его синтаксис схож с другими скриптовыми языками программирования, например, *JavaScript* или *JScript*.

ActionScript является **компилируемым** языком. Компиляция происходит в момент создания SWF-файла. При этом программный код переводится в более компактную форму **байт-кодов**, которые исполняются **виртуальной машиной ActionScript** (AVM), являющейся частью проигрывателя Flash Player и пакета AIR. Благодаря этому flash-проекты являются кросс-платформенными.

Язык ActionScript имеет **обширную библиотеку встроенных классов**, которые позволяют быстро решать большинство типовых задач.

Для написания сценариев (программных кодов) **не обязательно** знать **все элементы** языка ActionScript. Можно начинать изучение **с самых простых** конструкций, постепенно увеличивая сложность решаемых задач.

## Версии языка ActionScript

- ***ActionScript 1.0*** – исторические первая и самая **простая** версия языка ActionScript. В настоящее время она используется лишь в некоторых версиях проигрывателя Flash Lite для мобильных телефонов и устройств.
- ***ActionScript 2.0*** – его иногда называют «языком программирования *для дизайнеров*». Для изучения его основ и практического применения не требуется глубоких знаний программирования. ActionScript 2.0 используется в основном проектах **ориентированных на оформление**.
- ***ActionScript 3.0*** – новая версия языка, основанная на объектно-ориентированном программировании. Код, скомпилированный под данную версию, выполняется **максимально быстро**. Однако для изучения ActionScript 3.0 требуются более **глубокие** знания объектно-ориентированного программирования.

Скрипты, написанные на ActionScript 3.0 **НЕ совместимы** с более ранними версиями языка и не могут смешиваться с ними! Поэтому, важно **определиться с версией** используемого языка ActionScript **до начала** работы над проектом.

## Типы сценариев. Где помещаются коды?

При создании программно управляемых роликов коды на языке ActionScript можно размещать либо **внутри FLA-файлов**, либо выносить их в **отдельные AS-файлы**.

Внутри flash-ролика, коды языка **ActionScript 2.0**, могут относиться к *кадрам* временной шкалы или *экземплярам клипов (Movie clip) и кнопок (Button)*. В языке **ActionScript 3.0** скрипты относятся ***только к кадрам***, что помогает **избежать децентрализации** кода, когда весь код проекта разбит на маленькие кусочки, принадлежащие разным экземплярам символов.

**Важно**, что добавлять скрипты можно **только** к **ключевым кадрам** (***Keyframe***), которые **не** являются частью анимации движения (Motion Tween) или обратной кинематики (Armature).

В рабочей области ролика экземпляры, имеющие коды ActionScript, никак **не выделяются**. На временной шкале кадры, имеющие скрипты, отмечаются **буквой *a*** (action):  .

## Порядок выполнения сценариев

Первым выполняется код, расположенный в **верхнем** слое **первого** кадра ролика. Далее выполняются все сценарии, относящиеся к кадрам или объектам нижележащих слоев первого кадра (по порядку следования слоев - сверху вниз). Только после этого Flash Player переходит к следующему кадру ролика и так далее, пока не будет достигнут конец ролика.

*При настройках по-умолчанию скрипты выполняются не зависимо от видимости слоя. Однако если в настройках публикации ролика* **File > Publish Setting** *на вкладке* **Flash** *отключить параметр* **Include hidden layers***, скрипты* ***скрытых*** *слоев* ***не будут исполняться****.*

## Загрузка команд ActionScript из внешнего файла

Если при создании flash-проекта требуется работать с *большим* количеством программного кода, то уместно вынести весь код или его фрагмент в отдельный файл. Внешние файлы скриптов имеют разрешение ***.AS** *(ActionScript)* и являются обычными текстовыми файлами. То есть Вы можете редактировать такие файлы простым текстовым редактором или непосредственно программами Adobe Flash или Adobe Flex. Графические элементы в AS-файлах храниться не могут.

Чтобы выполнить программный код из внешнего файла, его нужно **связать** с каким-либо элементом ролика (ключевым кадром или экземпляром символа). Для этого выделяем нужный кадр или экземпляр символа, и на палитре Actions (Действия) добавляем следующий скрипт:

```
#include «имя файла»
например:  #include «script.as»        // файл в той же директории
           #include «source/script.as»// файл внутри вложенной папки
           #include «../script.as»     // файл находиться на уровень выше
```

Директива `#include` загружает **весь** программный код из указанного файла. При этом код из внешнего файла будет выполнять точно так же, как если бы он был написан в данном месте FLA-файла.

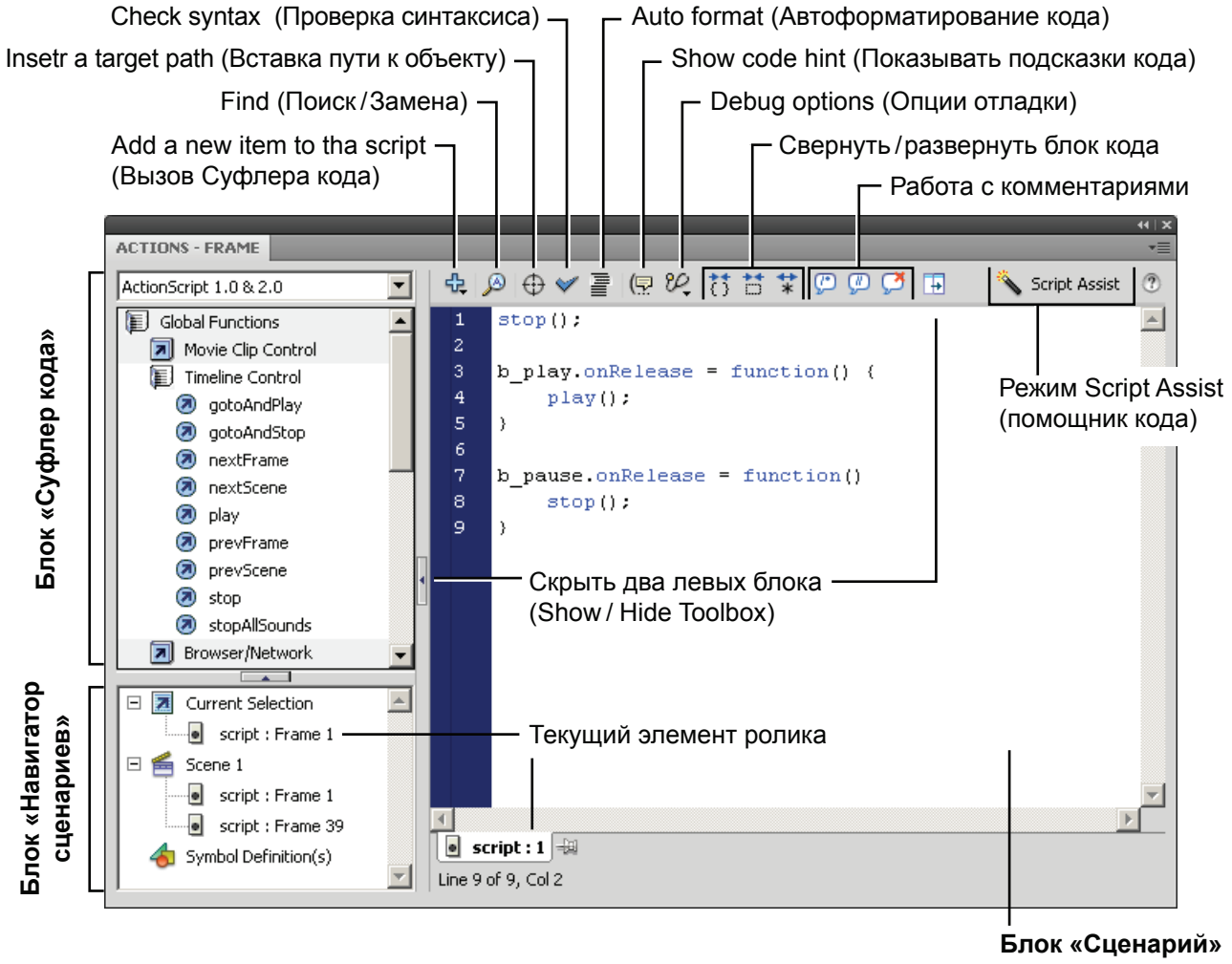
***Важно что после данной директивы НЕЛЬЗЯ ставить точку с запятой ( ; ) !***

## Палитра Actions (Действия)

Для работы со сценариями ActionScript, помещенными внутрь FLA-файла предназначена палитра **Actions** (Действия).
Данная палитра разделена на 3 части:

- Блок «***Сценарий***». Предназначен для ввода и редактирования кода ActionScript
- Блок «***Суфлер кода***». Содержит иерархический перечень всех зарезервированных конструкций языка. Двойной щелчок по любой конструкции добавляет в выделенное место блока «Сценарий»
- Блок «**Навигатор сценариев**». Указывает текущий элемент (*Current selection*), то есть тот элемент, чей сценарий в данный момент редактируется в блоке «Сценарий». А также содержит полный список всех элементов ролика, к которым добавлены сценарии ActionScript. Двойной щелчок по любому элементы вызывает его сценарий на редактирование.

Внешний вид данной палитры представлен на рисунке («горячая» клавиша ***F9***):




При работе с **внешними** файлами скриптов (*.AS) палитра **Actions не используется**. При открытии файла скриптов, *в верхней части документа* автоматически появляется палитра, содержащая все указанные функции палитры **Actions**.

## Синтаксис ActionScript и форматирование кода

**Важно:**
- Язык ActionScript является **регистрочувствительным**, то есть нельзя путать строчные (a) и прописные (A) символы.
- Команды разделяются точкой с запятой «**;**»
- Блоки кода и тело функций помещаются внутрь фигурных скобок **{**…**}**.

Для удобства чтения кода применяют *форматирование*, при котором каждый подчиненный элемент помещается с отступом вправо.
Также программа Flash автоматически включает *цветовую подсветку* кода:
- *Синий* цвет – все зарезервированные конструкции языка
- *Черный* цвет – все пользовательские конструкции
- *Зеленый* цвет – текстовые строки, заключенные в кавычки, например: `"текст"`
- *Серый* цвет – комментарии

**Комментарии** – это не исполняемые строки кода, которые служат для пояснения какого-либо фрагмента кода. Так же можно закомментировать блок кода, чтобы временно исключить его из выполнения (как правило, во время отладки ролика).
// - обозначают однострочный комментарий до конца строки (line comment)
/* - начало многострочного блока комментариев (block comment)
*/ - конец многострочного блока комментариев (block comment)

**Настроить форматирование** и подсветку кода можно с помощью команды **Edit > Preferences** (Редактирование > Настройки) вкладки *ActionScript* и *Auto Format.*

Настройки отображения кода доступны из меню дополнительных опций палитры Action:
- *Ward Warp* – перенос по словам
- *Line Numbers* – номера строк
- *Hidden Characters* – скрытые символы
- *Esc Shortcut keys* – «быстрые клавиши» Escape-ввода

Во Flash для **ускорения ввода** зарезервированных конструкций языка используется **Escape-ввод**. То есть перед вводом имени команды необходимо нажать клавишу ***Escape*** и тогда не нужно вводить имя полностью, достаточно ввести «быстрые клавиши» данной команды.

Режим **Script Assist** (*помощник кода*) позволяет создавать скрипты с помощью мастера. При этом Вы не вводите скрипт полностью, а составляете его из элементов, управляемых мастером.

Для вызовы **контекстной справки** необходимо **выделить** нужную функцию и нажать **F1**.

## Области видимости

Переменные и функции в ActionScript могут быть **локальными** и **глобальными**.
**Локальная** переменная или функция сохраняется как свойство клипа и ее видимость ограничивается им. Чтобы обратиться к переменной или функции одного клипа из другого клипа необходимо указать *«точный» адрес* того клипа, где была объявлена эта переменная или функция. ***Методы адресации будут рассмотрены на следующем занятии.***

Если необходимо, объявить **глобальную** переменную или функцию, которая будет доступна из всех клипам ролика, ее следует сохранить в особом объекте **Global**. Этот объект является носителем всех *встроенных классов, методов и функций* - именно поэтому они одинаково доступны из любой точки ролика. Чтобы объявить глобальную переменную или в последующем обратиться к ней необходимо использовать специальное свойство **_global**.

В ActionScript существует важное понятие «**цепочка областей видимости**», то есть список объектов, в которых компилятор будет искать вызываемую функцию или переменную.

**Длина** цепочки областей видимости *не постоянна* и зависит от расположения кода в ролике. Если Вы попытаетесь обратиться к не существующей переменной, поместив код в кадре главной временной шкалы ролика, то цепочка областей видимости будет следующей:
1. Вначале будет проверена область **_root** (так называется главная временная шлака ролика)
2. Затем будут просмотрены наследуемые ею, как объектом класса MovieClip, элементы. При этом будет просмотрено содержимое **прототипов** конструкторов MovieClip *(MovieClip.prototype)*, а затем Object *(Object.prototype)*.
3. В последнюю очередь будут просмотрены свойства объекта **Global**.

И только после этого будет возвращено сообщение о том, что переменная не существует.

## Типы данных

**Тип** - это устойчивое множество схожих данных. Все числа подчиняются одинаковым правилам, поэтому они входят в один тип. Текст не похож на числа - для него создан отдельный строковый тип данных. Всего в ActionScript существует **7** типов данных: ***Number, String, Boolean, null, undefined, Object, MovieClip, Void.***

Первые пять относятся к **элементарным** типам, Object, MovieClip и Void принадлежат к **сложным** типам данных. Основное различие заключается в том, что объекты элементарных типов рассматриваются как *единое целое*. Объекты же сложных типов данных содержат компоненты, то есть они сами *имеют некую структуру*.

**Типы данных:**
- **Number - Числа**. В ActionScript не существует типов real и integer, а все числа относятся к типу number, которому соответствуют 64-битные числа с плавающей точкой.
- **String - Строковые данные**.
- **Boolean - Логические величины**. К этому типу относятся две величины - *true* (истинно) и *false* (ложно). Величину true называют логической единицей, а false - логическим нулем.
- **Undefined** - К типу undefined принадлежит только одна величина, задаваемая литералом *undefined*. Она автоматически присваивается переменной при создании в том случае, если ей не было определеноконкретного значения. Так же величина *undefined* возвращается при попытке обращения к несуществующим переменным, функциям, свойствам или методам (или при ошибке в адресации).
- **Null** - Единственная относящаяся к нему величина, задаваемая литералом *null*. Она показывает, что некоторая переменная или свойство не имеет конкретного значения. Однако null, в отличие от undefined, никогда не присваивается автоматически. Эта величина введена в язык для того, чтобы было возможно различить две принципиально разные ситуации: переменная не существует или же она просто не содержит пока конкретного элемента данных.
- **Object - Объект.** В ActionScript все объекты являются экземплярами некоторого класса.

Спецификация *ECMA-262* использует следующее определение объекта: «Объект - это неупорядоченное множество свойств, каждое из которых сохраняет элементарную величину, объект или функцию. Функция, сохраненная в свойстве, называется методом»

- **MovieClip - Клипы**. Это основные символы, используемые во Flash. Формально клипы являются такими же объектами данных, как, например, массивы или строки, но имеется и ряд существенных отличий, которые будут рассмотрены на следующих занятиях.
- **Void** - Данный тип используется для обозначения функций, не возвращающих значений.

Определить к какому типу относиться объект, можно используя специальный оператор **typeof**. Например:

```
var count = 4;
trace (typeof count);     // number
trace (typeof object_1); // MovieClip
```

## Базовые конструкции языка ActionScript

### Переменные (var)

Переменная является своего рода контейнером, в который можно помещать некоторое значение, а затем изменять его. ActionScript относится к языкам программирования со слабым контролем типов данных. То есть, Вы можете присваивать переменной значения любого типа.

В ActionScript, как и в большинстве язяков программирования, различают локальные и глобальные переменные. **Глобальные** переменные доступны всем функциям и клипам ролика. Доступ к **локальным** переменным можно получить только из той функции или клипа, где она объявлена.

Объявление **локальной** переменной:
`var имяПеременной = значение;`
например: `var frameNum = 5;`

Объявление **глобальной** переменной:
`_global.имяПеременной = значение;`
напрмер: `_global.frameTotal = 15;`

### Принцип присвоения имен

Любой идентификатор (имя переменной, функции, класса и т.д.) может быть образован при помощи букв произвольного регистра, цифр, знака подчеркивания ( **_** ) или знака доллара (**$**).

Использование **всех остальных** служебных символов и символов пунктуации **недопустимо**, так как эти знаки зарезервированы в качестве операторов и разделителей ActionScript. Также в имя **не** может входить **пробельный символ**.

Имена функций (методов), переменных (свойств) и объектов принято начинать с маленькой буквы *(например, petName)*, а имена классов - с большой *(например, VirtualPet)*.
Если идентификатор образован несколькими корнями, то их начало желательно выделять большой буквой *(напрмер, gotoAndPlay)*.

В ActionScript не существует констант. Однако хороший стиль программирования требует, чтобы переменные, которые не должны изменяться, визуально отличались от изменяемых переменных. Поэтому имена констант в сценариях следует задавать буквами верхнего регистра *(например, MAXCOLOR)*, а изменяемых переменных - нижнего *(например, curentColor)*.

**Функции (function)**

Функции - это обособленные участки кода, которые могут выполняться многократно. Аналогично математике, функции в программировании имеют имена и могут принимать аргументы. Исполняемый блок кода ограничивается фигурными скобками и называется телом функции.

Функцию необходимо одни раз объявить:

```
// Пример 1 - принимает 2 параметра (аргумента)
function kineticEnergy(mass, velocity){
     // производит вычисления:
     var energy = mass * velocity * velocity / 2;
     // возвращает результат:
     return energy;
}
// Пример 2 - не принимает аргументов и не возвращает значения
function addM(){
     var summ = m1 + m2 + m3;       // производит вычисления
}
```

Использовать (вызывать) функцию можно многократно. Чтобы вызвать функцию, достаточно указать ее имя, при этом в скобках необходимо указать требуемые аргументы:

```
// вызов функции, не требующей аргументов:
addM();
// вызов функции и присвоение переменной возвращаемого значения:
var energy1 =  kineticEnergy(m1,v);
```

Если функция возвращает значение, которое необходимо присваивать переменно, то можно использовать совмещенное объявление и вызов функции:

```
var energy4 = function (mass, velocity){
     return mass * velocity * velocity / 2;
}
```

**Условные операторы**

В ActionScript используется два условных оператора **if** и **switch**. Они позволяют выполнять разные блоки кода в зависимости от значения условного выражения. Условия проверяются сверху вниз. Если условное выражение верно, то выполняется соответствующий ему блок кода, а последующие условия не проверяются.

Если блок кода содержит **одно единственное действие**, то фигурные скобки, ограничивающие данный блок кода можно не ставить. Если необходимо выполнить **два и более действия**, блок кода обязательно должен начинать и заканчиваться **фигурными скобками**.

```
// Пример 1 - оператор if:
if (условное выражение 1) {
          блок кода 1;
}
else if (условное выражение 2) {
          блок кода 2;
}
else {     // если ни одно условное выражение не было выполнено
          блок кода 3;
}

// Пример 2 - оператор switch:
switch (условное выражение 1) {
     case значение1:
          блок кода 1;
          break;
     case значение2:
          блок кода 2;
          break;
     default: // если ни одно из значений не совпало
          блок кода 3;
}
```

У конструкции `if-else` также имеется ускоренная форма записи с применением условного оператора **?:**

```
условное выражение ? блок if : блок esle
```

```
Пример:
a <= b ? trace(a + " НЕ больше " + b) : trace (a + " больше " + b)
```

При задании условных выражений используются следующие **операторы сравнения**:
< (меньше), > (больше), < = меньше или равно), > = (больше или равно),
= = (равно), ! = (не равно)

А так же логические операторы: || (ИЛИ), && (И), ! (НЕ)

**Операторы цикла**

Предназначены для множественного выполнения некоторой группы действий, заключенных в блок цикла. В ActionScript существуют четыре вида циклов: **while**, **do-while**, **for** и **for-in**.

**Цикл while**
Проверяется условное выражение и о тех пор пока оно верно *(true)*, выполняется блок кода.

```
while (условное выражение) {
     блок кода;
}
```

**Цикл do-while**

Отличие от цикла **while** заключается в том, что сначала выполняется блок кода, и только за тем проверяется условное выражение. Если оно стало ложным *(false)*, цикл завершается.

```
do {
      блок кода;
}while (условное выражение)
```

**Цикл for**

На практике данный тип цикла используется наиболее часто. Он позволяет выполнить блок кода фиксированное количество раз.

```
for (инициализация; условное выражение; корректирование){
      блок кода;
}
//Например:
var rezult = 2;
for (i=0; i<15; i++){ // блок кода выполниться 15 раз
      rezult *= 2;
      trace ("шаг " + i + " = " + rezult);
}
```

**Цикл for-in**

Данный цикл удобен в случае, когда необходимо перебрать все элементы массива или свойства объекта. Блок кода выполняется столько раз, сколько элементов в массиве или свойств у объекта, при этом порядок их перебора произволен и может измениться при повторном действии.

```
for (переменнаяИтерации in объект){
      блок кода;
}
//Например - объявили массив:
var myArray:Array = new Array("one", "two", "three");

// цикл по всем элементам массива(имя переменной произвольное):
for (var index in myArray){
      trace("myArray["+index+"] = " + myArray[index]);
}
```

**Массивы (Array)**

Массив – это упорядоченное множество однотипных элементов. В простейшем случаем массив представляет собой просто упорядоченный список элементов. Каждый элемент имеет **уникальный** порядковый номер (**индекс**). Такой массив называется **одномерным**.

В более сложном случае используются двухмерные массивы, трехмерные и так далее. *Математический аналог массива – матрица*. Все многомерные массивы в ActionScript реализованы как массивы массивов.

Массив в ActionScript - это объект встроенного класса **Array**. Для создания нового объекта класса (в данном случае массива) необходимо использовать оператор **new**:

```
var имяМассива = new Array();
```

Так же можно использовать сокращенную форму записи:

```
var имяМассива = [];
```

При создании массива можно **заполнить** его элементами:
```
var colors = ["красный", "оранжевый", "желтый" , "зеленый"];
```

Можно создать пустой массив **определенной длины**:
```
var array2 = new Array(3);    // в массиве будет 3 элемента
```

Определить **текущую длину** массива можно с помощью свойства **length**:
```
colors.length;                // длина массива = 7
```

Чтобы **обратиться** к элементу массива, необходимо указать имя массива, после него в квадратных скобках индекс нужного элемента. **Элементы массивы нумеруются от нуля!**

```
// заполнение/изменение массива:
array2[0] = 51;
array2[1] = 76;
array2[2] = 81;
```

## Отладочный вывод

Чтобы найти ошибки в программном коде или получить информацию о выполнении ролика в ходе его работы следует использовать функцию **trace()**. Данная функция может принимать **только один аргумент** - имя переменной или текстовую строку.

Чтобы узнать **текущее значение** переменной, помещаем в нужное место программного кода функцию **trace()** и передаем ей **имя переменой**:
```
var colorNum = 256;
trace(colorNum);    // будет выведено 256
```

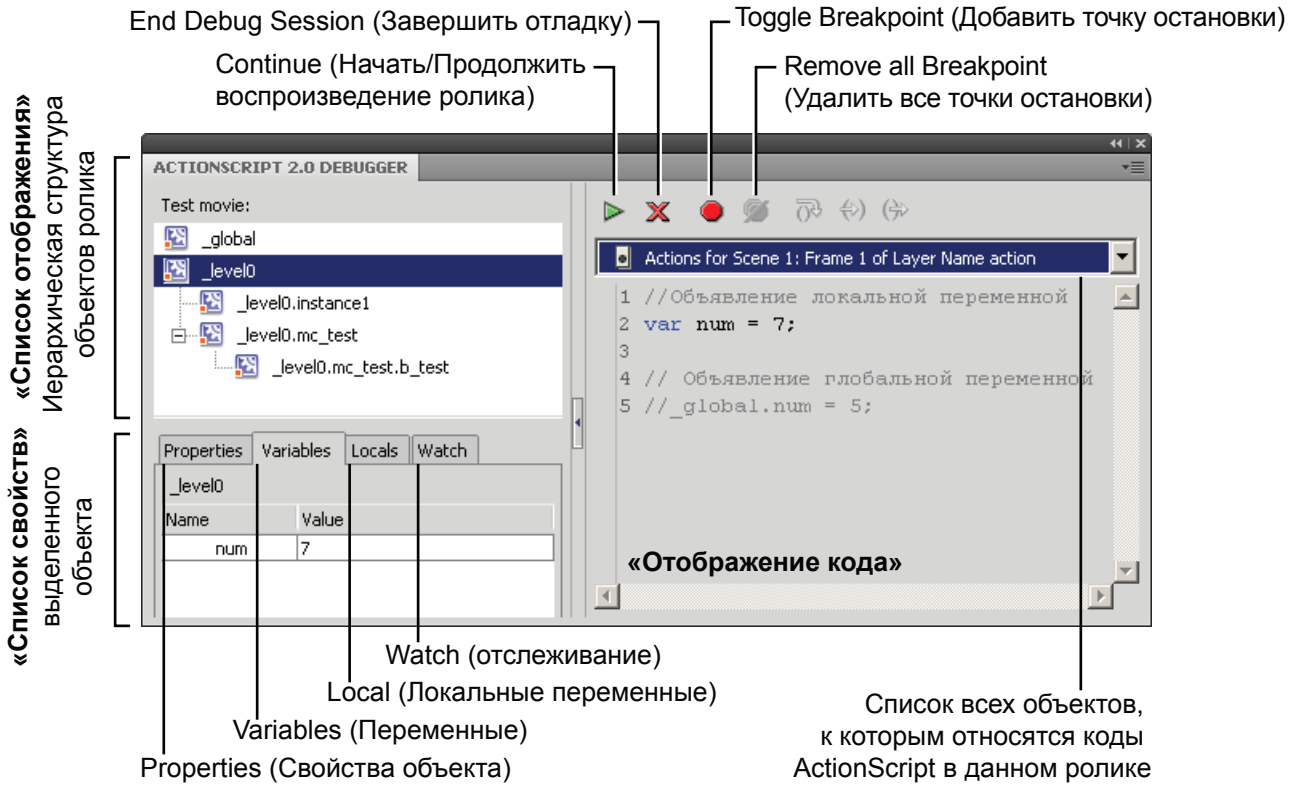
Если необходимо проследить алгоритм, можно вызывать функцию trace() с текстовой строкой:
```
if (colorNum > 16)
     trace("ok!")         // будет выведено ok!
else
     trace("bad...")      // будет выведено bad…
```

Для отображения отладочного вывода предназначена палитра **Output** (Вывод). Данная палитра открывается автоматически, если выполняется функция **trace()**.

*Отладочный вывод виден только в среде разработки Flash*. Функции trace() по умолчанию включаются в SWF-файлы, но при просмотре ролика с помощью Flash Player **пользователь не видит отладочного вывод**а. Можно отключить внедрение функции trace() в SWF-файлы. Для этого в настройках публикации ролика (**File > Publish Settings**) на вкладке **Flash** необходимо установить флажок **Omit trace actions** (Пропускать отладочный вывод).

## Отладка ролика (Debug)

Еще более широкие возможности контроля параметров ролика предоставляет отладчик ролика. Чтобы запустить ролик в режиме отладки необходимо выполнить команду главного меню **Debug > Debag Movie**. При этом откроется окно отладчика, в котором Вы можете динамически контролировать параметры ролика. Чтобы запустить воспроизведение ролика необходимо нажать кнопку **Continue** (Продолжить).
На следующем рисунке представлен внешний вид окна отладчика:




## Основы работы с классами

ActionScript является объектно-ориентированным языком. Объектно-ориентированное программирование - это концепция, основанная на трех основных принципах: **абстракция, инкапсуляция, наследование.**

Одним из главных преимуществ объектно-ориентированного подхода в программировании является инкапсуляции. **Инкапсуляция** (или скрытие реализации) позволяет создавать программу в виде отдельных, в большей или меньшей степени *изолированных блоков*.

Так как небольшой блок кода гораздо проще для восприятия, такой подход упрощает создание и отладку сложных программных продуктов. К тому же разбиение программы на блоки позволяет избежать конфликта имен.

**Основные понятия**

В объектно-ориентированном программировании **объект** - это группа переменных и функций, объединенных в одну логическую структуру. При этом *переменные* объекта принято называть **свойствами**, а *функции* - **методами**. *Свойства* определяют состояние объекта, а *методы* - его поведение. Методы, как правило, служат для изменения состояния объекта, то есть при вызове они нужным образом изменяют его свойства.

Часто в программе должно быть несколько однотипных объектов. Создавать их копированием кода не очень технично, так как это приведет к увеличению размера программы. Поэтому, объекты формируются на базе шаблона, называемого **классом**. Класс содержит описание всех методов и свойств, которые должны быть присущи данному объекту. Объекты созданные из класса называют его **экземплярами**. Таким образом, объект в ActionScript — это, с одной стороны, *объект данных* типа `object`, а с другой, *экземпляр* некоторого *класса*.

ActionScript содержит множество встроенных классов, экземпляры которых Вы может использовать в своих скриптах. Полный перечень таких объектов, а также их свойства и методы приведены в справочной системе по ActionScript (например, Move Clip, Button и тп. ).

Свойства и методы объекта могут обладать **разной степенью доступности**. Они могут быть *внутренними (private)*, в этом случае получить доступ к свойству или методу можно только из того класса, где они определены. Так же свойства и методы могут быть доступными для чтения, удаления и переопределения из других частей программы *(public)*. Разная степень доступности свойств позволяет минимизировать вероятность сбоя в работе всей программы при ошибочном удалении или переопределении одного из свойств. В ActionScript свойства и методы *всех встроенных объектов защищены* от удаления и перечисления циклом for-in, для большинства недопустимо также переопределение.

С объектами и классами объектов связано еще одно важнейшее понятие объектно-ориентированного программирования - *наследование*. **Наследование** - это передача переменных и функций между независимыми модулями программы. Зачем нужно наследование? Прежде всего, оно позволяет минимизировать размер программы, а также трудоемкость ее создания. Если есть несколько классов, у которых часть свойств и методов должна быть одинакова, то не следует объявлять их в каждом классе. Сначала создается класс, в котором объявляются одинаковые свойства и методы. Затем другие классы наследуют общие свойства и методы от этого класса и расширяют их своими индивидуальными методами и свойствами. При этом протяженность цепочки наследования может быть произвольной длинны.

**Особенности ActionScript 2.0**

*Модели объектно-ориентированного программирования в ActionScript 1.0 и ActionScript 2.0 НЕ совместимы. С целью обеспечения обратной совместимости было введено следующее ограничение:* ***в ActionScript 2.0 определение класса принципиально не может быть создано внутри fla-файла.*** *Соответствующий скрипт должен быть сохранен в отдельном*****.as-файле****.*

*При публикации для совместимости с ActionScript 1.0 определение класса будет импортировано и скомпилировано в обычную функцию-конструктор, сохраненную как* ***метод объекта _global****.*

**Создание классов в ActionScript 2.0**

**Класс** — это шаблон, на базе которого задаются собственные и наследуемые свойства и методы некоторого объекта. Объекты, как экземпляры класса создаются на основании его определения. Использование подобного шаблона помогает решить две важнейшие задачи:
- минимизировать текст программы;
- автоматизировать организацию наследования.

**Важно:**
- Все классы должны храниться во внешних as-файлах (каждый класс в своем файле)
- Имя файла должно совпадать с именем хранимого класса
- По умолчанию файлы с описание классов должны находиться в той же папке, где располагается fla-файл проекта

**При объявлении класса** указывается заголовок класса:
Первым идет ключевое слово `class`. После него при необходимости следуют указатели, задающие особенности класса (например, подклассом какого класса он является). Затем указывается имя создаваемого класса. Имя класса принято начинать с прописной буквы.

После заголовка следует **тело класса**, ограниченное блоком фигурных скобок. Тело класса содержит описание всех свойств и методов класса, а также функцию конструктор.

```
class ИмяКласса {
     Тело класса
}
```

После того как описание класса создано и сохранено в отдельном **.as-файле*, для использования этого класса необходимо создать один или несколько **экземпляров** класса.

**Для создания экземпляров класса** используется оператор `new`.
*Например,* следующий код создает новый объект `myPet1`, который является экземпляром класса `VirtualPet`:
```
var myPet1 : VirtualPet = new VirtualPet();
```

Чтобы получить **доступ к свойствам и методам класса**, необходимо указать имя экземпляра класса, затем оператор `dot` (точка), а после чего указать имя нужного свойства или метода.
*Например,* вызов метода `setPetName()` для объекта `myPet1`:
```
myPet1.setPetName ("Лион");
```

**Конструктор класса**

При создании нового экземпляр класса всегда вызывается **функция конструктор** класса, которая производит необходимую инициализацию свойств и методов данного экземпляра. То есть, разные экземпляры одного и того же класса могут иметь разные начальные значения своих свойств.

Функция-конструктор должна иметь **такое же имя, как и класс.** Задается она в теле класса.

*Пример* создания класса, описывающего животное:

1. Определение класса в файле "VirtualPet.as":

```
class VirtualPet {
     // объявление свойств класса:
     var petType: String = null;
     var petAge:Number = null;
     var petName: String = null;

     // функция конструктор класса:
     function VirtualPet(setType:String, setAge:Number,
     setName:String) {
          petType = setType;
          petAge = setAge;
          petName = setName;
     }
}
```

2. Создание 2-х экземпляров класса и инициализация свойств.
Код размещен внутри *.FLA-файла, находящегося в той же директории, что и файл "VirtualPet.as":

```
var myPet1 : VirtualPet = new VirtualPet("Лев", 7, "Лион");
var myPet2 : VirtualPet = new VirtualPet("Лев", 9, "Макс");
```

Данный код создает два объекта класса VirtualPet, описывающих двух различных животных.

# Практическая работа

*Все файлы, необходимые для выполнения данной работы, находятся на раздаточном диске в папке «***lesson_01\Practice_Work_1***» (включая примеры выполненных заданий).*

**Задание 1. Баннер с возможностью выбора ссылки**

1. Откройте файл ***start.fla***. На главной временной шкале данного ролика уже помещено шесть ключевых кадров. А также кнопки для перемещения между кадрами и перехода по выбранной ссылке. Каждой ключевой кадр соответствует одной из возможных гипер-ссылкок.

2. Чтобы управлять объектами с помощью языка ActionScript, необходимо на палитре **Properties** (Инспектор свойств) присвоить каждому объекту уникальное имя экземпляра **(Instance Name)**. Присваиваем кнопкам следующие имена:
   - Кнопка перехода по выбранной ссылке: *b_link*
   - Кнопка перехода к следующей ссылке: *b_next*
   - Кнопка перехода к предыдущей ссылке: *b_prev*

3. Коды ActionScript, разместите во внешнем файле. Для этого создайте новый файл ***action.as*** и сохраните его в туже директорию, где находиться исходный FLA-файл проекта.

4. Чтобы связать наш проект с внешним файлом скриптов, в исходном FLA-файле в первом кадре слоя «action» после имеющейся команды `stop();` добавите следующую директиву:

```
#include "action.as"
```

Теперь весь программный код мы будет размещать в файле ***action.as***. Чтобы просмотреть результат работы ролика необходимо выполнять публикацию (CTRL+Enter) FLA-документа.

5. В файле ***action.as*** создайте двумерный массив, в который поместите все доступные ссылки в формате [“имя ссылки”, “URL”].

*Например:*

```
var href = new Array
(["Spiral Galaxies", "http://antwrp.gsfc.nasa.gov/apod/ap051222.html"],
["Milky Way", "http://antwrp.gsfc.nasa.gov/apod/ap050605.html"],
["Dark Nebulae","http://antwrp.gsfc.nasa.gov/apod/ap060409.html"],
["Reflection Nebulae","http://antwrp.gsfc.nasa.gov/apod/ap000302.html"],
["Sun","http://antwrp.gsfc.nasa.gov/apod/ap051106.html"],
["Saturn","http://antwrp.gsfc.nasa.gov/apod/ap030817.html"]);
```

6. Объявите переменные, отвечающие за номер ссылки и общее количество ссылок:

```
// количество доступных ссылок:
var linkMax = href.length;    // длина массива

// номер выбранной ссылки (значения - от 0 до 5):
var linkNum = 0;
```

7. Номер и название выбранной ссылки будет вывить в панель отладочного вывода (**Output**). Для этого объявите пользовательскую функцию:

```
function linkTrace(){
     // отладочный вывод в следующей формате:
     // Номер. Название ссылки
     trace ((linkNum+1) + ". " + href[linkNum][0]);
}
```

8. Чтобы отобразить данные о первой (выбранной по умолчанию ссылке) необходимо сразу вызвать созданную функцию:

```
linkTrace();
```

9. Теперь добавьте обработчики для кнопок перехода между ссылками:

```
//переход к следующей ссылке:
b_next.onRelease = function(){
          nextFrame();   // переход на следующий кадр
          linkNum ++;    // увеличение номера выбранной ссылки

          // если переменная вышла за допустимый диапазон
          if (linkNum > (linkMax - 1))
               // максимальное значение переменной
               linkNum = linkMax - 1;
          linkTrace(); // отобразить название
}

//переход к предыдущей ссылке:
b_prev.onRelease = function(){
          prevFrame();   // переход на предыдущий кадр
          linkNum --;    // уменьшение номера выбранной ссылки

          // если переменная вышла за допустимый диапазон
          if (linkNum < 0)
               // минимальное значение переменной
               linkNum = 0;
          linkTrace(); // отобразить название
}
```

10. Чтобы осуществить переход по выбранной ссылке, добавьте обработчик для кнопки перехода:

```
b_link.onRelease = function(){
     getURL(href[linkNum][1]);
}
```

Баннер готов!
*Пример готового кода находиться в файле **action.as***
*Готовый баннер находиться в файле **sample.fla***

# Занятие 2

*Продолжительность 180 минут*

**Тема занятия:**

Программное управление объектами

# Программное управление объектами

## Обработчик событий мыши (on) для кнопок и клипов

**Событие** – в общем случае это любое изменение в состоянии системы. В программировании существуют разные типы событий. Одни из них связаны с действиями пользователя, другие появляются автоматически в ходе выполнения программы.

Чтобы некоторое действие было выполнено, как только произойдет определенное событие, необходимо использовать обработчики событий. Для разных типов событий применяются соответствующие обработчики.

Существует **два стиля** написания кода обработчиков. Синтаксис обработчика меняется в зависимости от того, **куда помещен его код** - *в кадр временной шкалы* (более новый и предпочтительный стиль) или *на экземпляр символа* (устаревший синтаксис, оставлен для совместимости с кодами Flash 5).

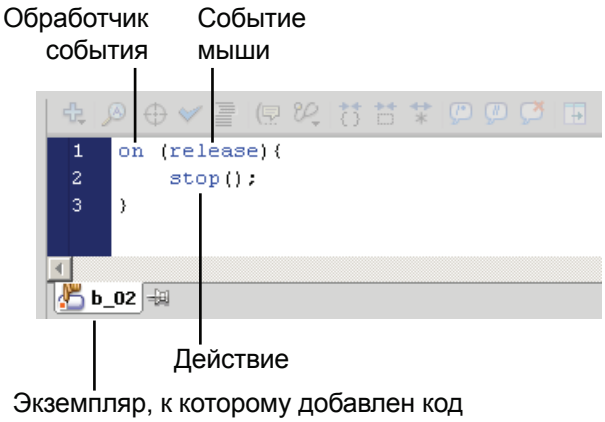
Для обработки событий мыши в **ActionScript 2.0** используется обработчик «`on`». С его помощью одинаково обрабатываются события мыши для кнопок и клипов.

**Обрабатываются следующие события мыши:**

- `press` – нажали левую кнопку мыши в зоне чувствительности объекта
- `release` - нажали и отпустили левую кнопку мыши (происходит в момент отпускания)
- `releaseOutside` – нажали левую кнопку мыши на объекте, а отпустили вне его (происходит в момент отпускания)
- `rollOver` – навели указатель мыши на объект
- `rollOut` – вывели указатель мыши из зоны чувствительности объекта
- `dragOver` - нажали левую кнопку мыши в зоне чувствительности бъекта, не отпуская кнопку мыши, вывели мышь за его пределы, а затем вернули обратно.
- `dragOut` - нажали левую кнопку мыши в зоне чувствительности объекта, не отпуская кнопку мыши, вывели мышь за пределы объекта

**Добавление обработчика события к экземпляру символа**

Чтобы добавить обработчик события, необходимо на рабочей области **выделить экземпляр** нужного символа, и с помощью палитры Actions ввести следующий код:




Обработчик событий «**on**» в качестве аргумента требует указать **событие** мыши, по которому должны происходить некоторые действия.

Событие мыши указывается в круглых скобках. Действия указываются внутри фигурных скобок и разделяются точкой с запятой.

*Приведенный сценарий при нажатии на кнопку останавливает анимацию.*

**Добавление обработчика события к кадру**

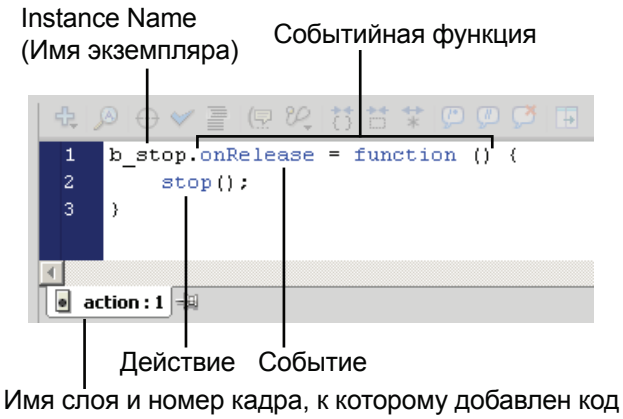
Чтобы добавить обработчик события мыши к кадру необходимо:

1. Выделить на рабочей области экземпляр нужной кнопки и на палитре **Properties** (Инспектор свойств) присвоить ему **Instance Name** (Имя экземпляра). Это позволит управлять данным экземпляром с помощью языка ActionScript. Например, присвоим кнопке, которая будет останавливать анимацию, **Instance Name**: `b_stop`.

2. Для размещения кода ActionScript обычно создают *отдельный слой*, который располагают выше всех других слоев и называют «*action*». В этот слой не размещают графические объекты, он содержит только сценарии, что упрощает доступ к кодам и облегчает работу над проектом. После создания слоя «*action*» нужно выделить его первый кадр и вызвать палитру **Actions**.

3. В блоке «Сценарий» вводим необходимый код. **Важно**, что при использовании «событийных функций» формат указания событий меняется на следующий:

- `onPress, onRelease, onReleaseOutside,`
- `onRollOver, onRollOut,`
- `onDragOver, onDragOut`




Первым указывается имя экземпляра (Instance Name), далее оператор **dot** *(точка)* и название необходимого свойства объекта. После этого ставиться оператор присвоения (=) и ключевое слово **function**. Данный синтаксис называют *событийной функцией.*

Действия указываются внутри тела событийной функции *(т.е. внутри фигурных скобок)* и разделяются точкой с запятой.

Если по одному событию должно происходить **несколько действий**, то все их нужно помещать внутрь **одного** обработчика.

Так же к одному символу может быть добавлено **несколько обработчиков**, использующихся для **разных** событий.

```
1 b_stop.onPress = function () {
2     stop();
3 }
4
5 b_stop.onRelease = function () {
6     play();
7     stopAllSounds();
8 }
```

## Управление временной шкалой ролика

Управление временной шкалой ролика или клипа осуществляется в помощью **встроенных функций** языка ActionScript. Это **самая простая** и наглядная группа функций. По умолчанию, воспроизведение анимции ролика начинается сразу и идет от первого кадра к последнему. Функции управления временной шкалой предназначены для **остановки/воспроизведения** анимации, а также для **изменения последовательности** отображения кадров ролика.

Функции управления временной шкалой относятся к глобальным функциям.
На палитре **Actions** (Действия) в блоке «*Суфлер кода*» функции управления временной находятся группе **Global Functions > Timeline Control** (Глобальные функции > Управление временной шкалой). К данной группе относятся следующие функции:

- `stop();` - остановить анимацию
- `play();` - продолжить воспроизведение анимации
- `nextFrame();` - перейти к следующему кадру временной шкалы
- `prevFrame();` - перейти к предыдущему кадру временной шкалы
- `nextScene();` - перейти к следующей сцене ролика
- `prevScene();` - перейти к следующей сцене ролика
- `stopAllSounds();` - остановить все звуки ролика

*Вышеперечисленные функции не требуют аргументов.*

- `gotoAndPlay();` - перейти к кадру и начать воспроизведение анимации
- `gotoAndStop();` - перейти к кадру и остановить воспроизведение анимации
-

*Для последних функций **требуется аргумент**, указывающий номер того кадра, куда будет осуществлен переход. Аргументы функций указываются внутри круглых скобок.*

В качестве **аргумента** можно прямо указать номер нужного кадра:

```
gotoAndPlay(5); // перейти на 5-й кадр
```

При этом данная команду будет управлять **главной временной шкалой ролика** *(для глобальных функция и свойста указание впереди ключевого слова _root не является обязательным).*

Чтобы упралять **временной шкалой** конкретного **кипа**. Ему необходимо присвоить имя эеземпляра (**Instance Name**), которое задается на палитре **Properties (Инспектор свойств).**

**Пример.** Чтобы остановить внутреннюю анимацию экземпляра клипа *"movie_mc"*, необходимо указать имя экземпляра данного клипа, замет оператор dot (точка) и после это ввести имя соответствующей функции управления временной шкалой:

```
movie_mc.stop();
```

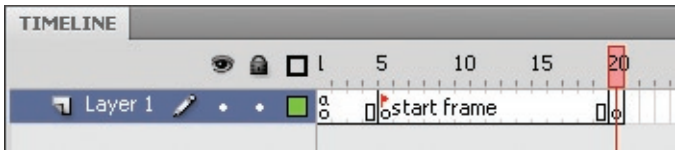
## Метки кадров

**Любому ключевому кадрам** (***Keyframe***), который **не являются частью** анимации движения (Motion Tween) или обратной кинематики (Armature) можно **назначить метку.**

Метки облегчают переход к нужному кадру. То есть, если в сценарии использованы ссылки на какой-либо кадр **по номеру**, то при изменении анимации так, что содержимое кадра переместилось на временной шкале, Вам придется **вручную изменять** номер кадра во всех ссылках. При использовании **меток** такой проблемы нет, так как **при перемещении кадра** по временной шкале **метка автоматически перемещается** вместе с кадром.

Чтобы назначить кадру метку необходимо выделить нужный ключевой кадр и на палитре **Properties** (Инспектор свойств) в секции **Lebel** в поле **Name** ввести имя метки. При этом в поле **Type** (Тип) должно быть выбрано значение **Name** (Метка кадра).

На временной шкале кадры, имеющие метки обозначаются **красным треугольником**, рядом с которым указывается имя метки:




На данном рисунке кадру 5 присвоена метка: **start frame**

**Важно**, что когда метки кадров указываются в качестве аргументов функций, писать их нужно в двойных кавычках. То есть перейти на **5-й кадр** и начать воспроизведение анимации можно с помощью следующей функции:

```
gotoAndPlay("start frame");
```

## Клип, как основной объект программной анимации

**Клип** - это основное понятие программы Flash. В подавляющем большинстве случаев ActionScript используется именно как инструмент управления клипами.

### Типы символов

Кроме клипов в программе Flash существуют еще два типа символов – это *графические символы* (Graphic) и *кнопки* (Button). Разница между символами разных типов не очень велика, так у них был **единый предшественник**. Программа Flash развилась из простого редактора для создания анимации **FutureSplash**, в которой имелся всего один тип символов.

**Клипы** являются *наиболее универсальными символами* и подходят для решения любых задач.

**Кнопки** – это по сути *специализированные клипы*, предназначенные для упрощения создания *интерактивных* элементов ролика. Однако, внутрь кнопок нельзя помещать команды ActionScript, что делает их не пригодными для решения сложных программных задач.

**Графические символы** предназначены для хранения объектов, которыми не нужно управлять программно. Для них не производиться проверок разного рода программных событий, которые всегда происходят для клипов и кнопок. Это позволяет снизить затраты процессорных ресурсов и оперативной памяти при работе ролика.

### Особенности клипов. Отличие от объектов

Клипы относятся к особому классу **MovieClip**.

Прототипом клипа, как уже сказано, является символ программы FutureSplash, в которой уже имелись некоторые скриптовые команды для управления символами. Таким образом, *понятие «клипа» появилось* за долго *до выхода языка ActionScript*.

Когда разработчики компании Macromedia создавали объектно-ориентированный язык ActionScript, за основу была взята спецификация *ECMAScript*, в которой **не было** даже близкого **аналога клипо**в. Поэтому многие понятия *ECMAScript* оказались **не применимыми** к клипам, что вызвало определенную путаницу. Например, экземпляры объектов создаются при помощи оператора **new** и конструктора класса, но это не применимо к клипам:

```
var clip:MovieClip = new MovieClip(); // НЕ ДОПУСТИМО
```

Так же нельзя удалить клип с помощью оператора **delete**.

*Методы создания и удаления клипов будет рассмотрены на следующих занятиях.*

Так образом, **клип – это особая, специфическая структура данных**. Клипам присуще такое понятие как временная шкала, они имеют собственную систему координат, внутрь их могут помещаться другие экземпляры символов, а также программный код. Клипы по своей сути являются *«маленькими» swf-фильмами*. Исходя из описанных особенностей, клипы относятся к **особому типу данных MovieClip**, *не предусмотренному спецификацией ECMA-262*.

**Все остальные объекты** ActionScript принадлежат к типу данных **Object** и являются экземплярами некоторых классов.

В ActionScript экземпляры **объектов не имеют собственных идентификаторов**. Их именами считаются имена указывающих на них переменных. **Экземпляры клипов** наоборот **обладают индивидуальными идентификаторами**, присваиваемыми им при создании.

Несмотря на принципиальные отличия, при написании сценариев клипы и объекты имеют **определенное сходство**. Клипам, как и объектам, *могут быть присвоены свойства и методы*. *Ссылки на них* могут хранить переменные и элементы массивов. У клипов имеются свойства *__proto__* и *constructor*, что означает, что принципы наследования для них совпадают с наследованием в случае объектов.

## Методы адресации в Adobe Flash

В Adobe Flash используется два метода адресации – *абсолютная* и *относительная*.

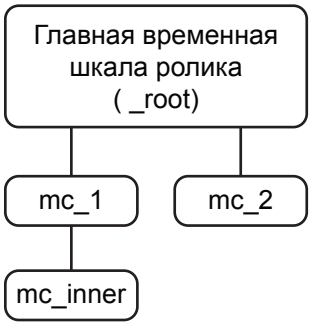
**Абсолютаная адресация**

В данном случае указывается полный путь к целевому клипу. При этом используются ключевые слова **_root** и **_level*N***.

Ключевое слово **_root** указывает на **главную временную шкалу** ролика.
Так как внутрь одного ролика могут динамически загружаться другие ролики и у каждого из них есть своя главная временная шкала, то введено еще одно понятие **_level*N*** – **программный уровень**. Каждый динамически загружаемый объект размещается на отдельном уровне. ***N*** – порядковый номер программного уровня, задается неотрицательным целым числом. **_level0** – является синонимом **_root** и ссылается на главную временную шкалу **данного** ролика. *Более подробна работа с программными уровнями будет рассмотрена на занятии 4.*

Если **клипы вложены один внутрь другого**, то чтобы обратиться к свойствам внутреннего клипа, необходимо указать цепочку имен всех клипов, в которые он вложен. При этом указываются *имена экземпляров* (**Instance Name**). Имена клипов разделяются оператором **dot** *(точка)*. Первым указывается имя клипа верхнего уровня.

**Например**, на главной временной шкале находятся два клипа *«mc_1»* и *«mc_2»*. Внутри клипа *«mc_1»* находиться еще один клип *«mc_inner»*.




Чтобы обратиться **с главной временной шкалы** к свойствам клипа «mc_inner» (например, координате X), необходимо поместить в ключевой кадр главной временной шкалы следующий код:

```
trace(_root.mc_1.mc_inner._x);
```

**Важно**, что при этом первое ключевое слово **_root не** является обязательным:
```
trace(mc_1.mc_inner._x);
```

Чтобы обратиться к свойствам **одного клипа из другого**, необходимо указать полный путь к клипу. При этом путь **обязательно** должен начинаться с ключевого слова **_root** и **_level*N***.

**Например**, чтобы обратиться из клипа *«mc_2»* к свойствам клипа *«mc_inner»* (координата X), необходимо на временную шкалу клипа *«mc_2»* добавить следующий код:
```
trace(_root.mc_1.mc_inner._x);
```

Однако абсолютная адресация является *не гибкой* и не позволяет создавать легко «переносимый» код. Поэтому более предпочтительной является относительная адресация.

**Относительная адресация**

При относительной адресации используются ключевые слова **_parent** и **this**.

Ключевое слово **_parent** ссылается на **родительский объект**. То есть на временную шкалу, где находиться данный объект (подъем на одну ступеньку вверх в иерархии вложенности клипов).

**В нашем примере**, чтобы получить доступ из клипа *«mc_inner»* к свойствам клипа *«mc_2»* (например, координате Y), необходимо поместить на временную шкалу клипа *«mc_inner»* следующий код:
```
trace(_parent._parent.mc_2._y);
```

Ключевое слово **this** *(записывается без переднего подчеркивания)* в ActionScript имеет огромное значение. Оно возвращает ссылку непосредственно **на сам объект.** Будучи использовано в коде метода, оно позволяет ссылаться на тот объект, который вызывает данный метод.

**Например,** чтобы обратиться к свойству клипа (например, координате Y) **из самого клипа**, нужно на его временную шкалу добавить следующий код:
```
trace(this._y);
```

## Программное изменение свойств экземпляров

В большинстве случаев программная анимация представляет собой изменение свойств клипов в зависимости от некоторых событий.

**Важно**, что **все** экземпляры, которыми необходимо управлять программно, должны иметь имена экземпляров (**Instance Name**), которые задается на палитре **Properties (Инспектор свойств).**

Свойства, которые мы будем рассматривать в данной теме, относиться к **глобальным свойствам**. Отличительной особенностью является то, что *перед именем* свойства обязательно ставиться *знак подчеркивания ( _)*. Глобальные свойства не относиться ни к одному классу. Все глобальные свойства применимы для клипов. Большинство их так же можно использовать для кнопок, а зачастую и динамических текстовых полей.

**Свойства экземпляров, используемые в программной анимации** (данные свойства можно использовать для клипов, кнопок и динамических текстовых полей):

| Свойство | Описание | Единица измерения |
|---|---|---|
| `_x / _y` | Координаты по осям X / Y | Пиксели (px) |
| `_width / _height` | Ширина / Высота | Пиксели (px) |
| `_xscale / _yscale` | Масштаб по осям X / Y | Проценты (%) |
| `_rotation` | Угол поворот | Градусы:<br>Положительные значения – поворот по часовой стрелке, отрицательные – против часовой стрелки |
| `_visible` | Видимость | Значение «true» - экземпляр видим, значение «false» - не видим |
| `_alpha` | Прозрачность | Проценты (%):<br>Значение «100» - экземпляр полностью видим, значение «0» - не видим |

Чтобы **изменить некоторое свойство** необходимо указать *имя экземпляра* клипа, кнопки или динамического текстового поля (**Instance Name**, задается на палитре **Properties**). После него необходимо указать оператор **dot** *(точка)* и *название того свойства*, которое требуется изменить. Далее следует знак математической операции и новое значение свойства.

```
Например:
obj._width = 300;    // ширина экземпляра «obj» теперь равна 300 px,
obj._x  += 10;       // а его координата x увеличилась на 10 px
(аналог данного кода: obj._x = obj._x + 10)
```

**Знаки математических операций:**

**=** (присваивание), **+** (сложение), **-** (вычитание),
***** (умножение), **/** (деление), **%** (остаток от деления)

**Важно**, что *вычисление прозрачности* экземпляра создает *дополнительную* вычислительную *нагрузку*, поэтому чтобы сделать объект полностью не видимым предпочтительнее использовать свойство **_visible**, а не свойство **_alpha**:

```
obj._visible = false;     // предпочтительно
obj_alpha = 0;            // не желательно
```

Также в процессе анимации также можно изменять режим наложения - **blendMode**. Это свойство не является глобальным.

```
Пример: obj.blendMode = «overlay»;
```

## Обработчик события загрузки клипа (onLoad)

Часто в начале воспроизведения ролика необходимо выполнить некоторые стартовые действия, которые в дальнейшем больше не должны происходить.

Для этой цели используется обработчик **onLoad**. Все **действия**, помещенные в тело обработчика, выполняются **только один раз – при загрузке клипа**. Можно назначить этот обработчик не какому-то конкретному клипу, а главной временной шкале (использую ключевые слова *_root* или *this*), тогда код обработчика выполниться **единственный** раз при загрузке всего ролика.

*Синтаксис обработчика также меняется в зависимости от того, где размещен его код.*

Пример 1. Код размещен в ключевом кадре главной временной шкалы.

```
// ключевое слово this указывает на объект к которому относится
обработчик, в данном случае на главную временную шкалу ролика:
this.onLoad = function() {    // при загрузке ролика
     stop();   // останавливается анимация главной временной шкалы
}
```

Пример 2. Код размещен на экземпляре некоторого клипа:

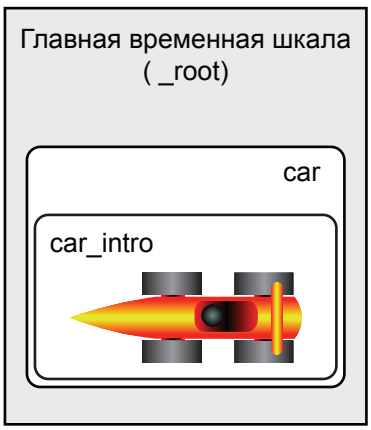
```
onClipEvent(load){              // при загрузке данного клипа
     trace ("Клип загружен"); // выводиться отладочный текст
}
```

## Сочетание программной анимации и анимации на временной шкале ролика

В некоторых случаях требуется **создать анимацию** объекта **на временной шкале** ролика (например, объект двигается по сложной траектории). Но при этом нужно **программно управлять свойствами** объекта (изменять масштаб, угол поворота и т.д.).

Если начать программно изменять свойства клипа, который является частью анимации временной шкалы, то эта анимация будет остановлена. В такой ситуации необходимо *вложить клип с графикой внутрь пустого клипа*. Пустой клип-контейнер будет участвовать в анимации. А свойства внутреннего клипа с графическими элементами можно изменять программно.

**Пример**. На главной временной шкале поместим клип *«car»* и создадим для него анимацию движения по траектории. Внутрь этого клипа поместим еще один клип *«car_intro»*, в котором нарисуем объект анимации (машину).




Теперь с главной временной шкалы можно изменять свойства внутреннего клипа, не опасаясь, что при этом остановиться анимация. Например, можно увеличить клип при нажатии на кнопку "b_zoom", расположенную на главной шкале ролика.

Для этого добавляем в кадр главной шкалы следующий код:

```
// по нажатию на кнопку:
b_zoom.onRelease = function() {
     // увеличиваем масштаб внутреннего клипа:
     car.car_intro._xscale *= 1.2;
     car.car_intro._yscale *= 1.2;
}
```

## Изменение цвета клипов

Метод `setRGB()` позволяет закрасить экземпляр клипа **однородным** цветом. При этом указанным оттенком закрашиваются **все заполненные пиксели клипа**. **Прозрачность** пикселей **сохраняется**. Также после применения метода `setRGB()` сохраняется прозрачность экземпляра клипа, заданная программно или при помощи свойства *alpha* на палитре **Properties** (Инспектор Свойств).

**Оттенок цвета** задается 16-ричным числом в модели RGB (гекс-кодом). Перед ним обязательно указывается приставка «`0x`», указывающая на 16-ричный формат числа.

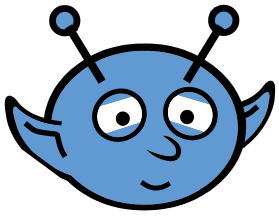
**Примеры задания цвета:**

| | | |
|---|---|---|
| Белый `0xFFFFFF`, | Черный `0x000000`, | Серый `0x7F7F7F` |
| Красный `0xFF0000`, | Зеленый `0x00FF00`, | Синий `0x0000FF` |

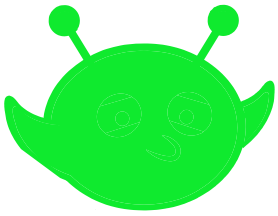
**Получить оттенок**, которым закрашен клип, позволяет метод `getRGB()`. Однако данный метод возвращает **ноль**, если к экземпляру клипа ранее **не применялся метод** `setRGB()`.

**Пример**. Создадим на главной временной шкале клип *«alien»* и изменим его цвет.

```
// создаем обект Color для управления цветом:
var colorful = new Color("alien");

// назначаем новый цвет:
colorful.setRGB(0x0FEA2E);

// считываем цвет объекта:
trace (colorful.getRGB().toString(16));
```



клип до применения функции setRGB



клип после применения функции setRGB

# Практическая работа

*Все файлы, необходимые для выполнения данной работы, находятся на раздаточном диске в папке «**lesson_02\Practice_Work_2**» (включая примеры выполненных заданий).*

**Задание 1. Программное изменение свойств клипа**

1. Откройте файл ***01_start.fla***. В этом файле на главной временной шкале находятся два клипа: «*alien*» - клип с инопланетянином, содержащий множество вложенных клипов, свойства которых нужно изменять и «*butbar*» - клип, внутри которого находятся управляющие кнопки.

2. Двойным щелчком войдите в режим редактирования эталона клипа «*butbar*». На его временной шкале в слое «action» уже имеется пример кода для изменения угла наклона антенн инопланетянина (свойство `_rotation`):

```
_root.butbar.b1.onRelease = function() {     //управление антеннами
     _root.alien.head.left_antenna._rotation = random(45);
     _root.alien.head.right_antenna._rotation = 180-random(45);
}
```

*Функция* `random(45)` *генерирует случайные числа в диапазоне от 0 до 45.*
*Подробно функции генерации случайных чисел будет рассмотрены на Занятии 3.*

3. Первой кнопке уже присвоено имя экземпляра «*b1*». Присвойте уникальные имена экземпляров (**Instance Name**) всем оставшимся кнопкам.

4. Допишите обработчики событий для кнопок так, чтобы каждая кнопка управляла каким-либо свойством клипа с инопланетянином или одного из вложенных клипов. При этом используйте абсолютную и относительную адресацию.

*Пример готового задания находиться в файле **01_sample.fla***

**Задание 2. Программное изменение цвета клипа**

1. Откройте файл ***02_start.fla***. В данном файле находятся следующие элементы:
• «*alien*» - клип, внутри которого находятся три вложенных клипа, цвет которых будет изменяться
• Блок кнопок «*Mode*», который содержит 3 кнопки. При нажатии одной из этих кнопок определяется, какой из вложенных клипов будет менять цвет.
• Блок кнопок «*Color*», который содержит 9 кнопок. При наведении мыши на одну из этих кнопок вызывается функция перекрашивания клипа в цвет, соответствующий данной кнопке.

2. Для всех кнопок, входящих в блок «*Color*», уже прописаны скрипты вызовы пользовательской функции `myColor()`. В качестве аргумента передается цвет, соответствующий данной кнопке. Скрипты добавлены непосредственно на экземпляры кнопок. Пример кода для первой кнопки:

```
on (rollOver) {
     myColor(0x003366);
}
```

3. Саму функцию `myColor()` необходимо объявить на главной временной шкале ролика в слое «*action*». Эта функция должна изменять цвет клипа на тот цвет, который передается в качестве аргумента. Имя клипа, который необходимо перекрашивать берется из переменной (в нашем примере это переменная `color_target`). Значение данной переменной изменяется с помощью блока кнопок «*Mode*».

```
// объявление переменных:
var color_num;                    // новый цвет
var color_target = "skin_mc";     // вложенный клип, который будет изменять
цвет

// объявление пользовательской функции:
function myColor(color_num): Void{
      // создали объект управления цветом:
      var colorful = new Color("alien."+color_target);
      // перекрасили клип:
      colorful.setRGB(color_num);
}
```

4. Теперь чтобы значение переменной `color_target` изменялось, допишите обработчики для кнопок блока «*Mode*». Имена экземпляров кнопкам уже присвоены: `b_mode_1`, `b_mode_2` и `b_mode_3`. Чтобы пользователь видел, какой выбран режим для активной кнопки установите параметр `_yscale = 150`, а для остальных кнопок `_yscale = 100`:

```
// изначально выделена первая кнопка:
b_mode_1._yscale = 150;

// события по кнопкам, изменяющим режим, т.е. перекрашиваемый клип:

b_mode_1.onRelease = function() {
      // увеличить размер данной кнопки
      his._yscale = 150;
      // уменьшить остальные кнопки
      b_mode_2._yscale = b_mode_3._yscale = 100;
      // переопределить перекрашиваемый клип
      color_target = "skin_mc";
}

b_mode_2.onRelease = function() {
      this._yscale = 150;
      b_mode_1._yscale = b_mode_3._yscale = 100;
      color_target = "kovta_mc";
}

b_mode_3.onRelease = function() {
      this._yscale = 150;
      b_mode_1._yscale = b_mode_2._yscale = 100;
      color_target = "bruki_mc";
}
```

*Пример готового задания находиться в файле **02_sample.fla***

# Занятие 3

*Продолжительность 180 минут*

**Тема занятия:**

Основы программной анимации

# Основы программной анимации

## Обработчик события смены кадров ролика (onEnterFrame). Покадровая программная анимация

При создании программно управляемых роликов крайне часто используется событие **onEnterFrame**. Оно происходит при каждой смене кадров ролика. Чем **выше** частота смены кадров *(fps)*, тем **чаще** будет происходить данное событие.

Событие **onEnterFrame** удобно применять для создания покадровой программной анимации. При каждой смене кадров изменяется одно или несколько свойств клипа.

Например, чтобы реализовать горизонтальное движение клипа *«fish»*, находящегося на главной временной шкале ролика, достаточно поместить в кадр главной шкалы следующий код:

```
this.onEnterFrame = function() { // при каждой смене кадров ролика
      fish._x += 10; // координата клипа увеличивается на 10 px
}
```

Если требуется разместить код обработчика события непосредственно на экзапляр анимирумого клипа, то код будет иметь следующий вид:

```
onClipEvent(enterFrame){          // при каждой смене кадров ролика
      this._x +=10;    // координата клипа увеличивается на 10 px
}
```

Ключевое слове `this` указывает на экземпляр, к которому относиться данный обработчик.

Однако при данном способе анимации объект быстро покинет границы ролика. Чтобы предотвратить это, необходимо с помощью логических условий ограничить область анимации.

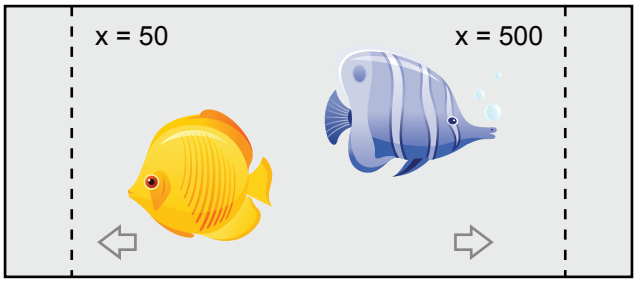
**Пример**. Создадим 2 клипа *«fish_1»* и *«fish_2»*, которые будут двигаться по горизонтали в промежутке от x = 50 до x = 500. Скорости движения у клипов разные. При достижении границ области каждый клип должен развернуться на 180 градусов и начать движение в обратную сторону. Оба клипа размещены на главной временной шкале ролика.

Ниже приведен пример кода, реализующего данный пример.
Код необходимо разместить в ключевом кадре главной временной шкалы ролика:

```
// объявим переменные, отвечающие за скорости движения клипов
(количество пикселей, на которые перемещается объект за один кадр)

var vx1 = 10;    // движение слева направо
var vx2 = -5;    // движение справа налево

// объявим переменные, отвечающие за положение границ области, внутри
которой двигаются объекты:

var border_left = 50;
var border_right = 500;

// добавляем обработчик события для клипа «fish_1»
fish_1.onEnterFrame = function() { // при каждой смене кадров ролика:
```

```
      // перемещаем объект:
      this._x += vx;

      // проверяем координаты:
      // Способ 1
      if (this._x > border_left && this._x < border_right){
            // объект ВНУТРИ заданной области - ничего не делаем
      }
      else {
            vx1 = - vx1;     // смена занка = движение в обратную сторону
            this._xscale = - this._xscale;   // поворот объекта
      }
}

// добавляем обработчик события для клипа «fish_2»
fish_2.onEnterFrame = function() {      // при каждой смене кадров ролика:
      // перемещаем объект:
      this._x += vx2;

      // проверяем координаты:
      // Способ 2
      if (this._x <= border_left || this._x >= border_right){
            // объект ВНЕ заданной области
            vx2 = - vx2;     // смена занка = движение в обратную сторону
            this._xscale = - this._xscale;         // поворот объекта
      }
}
```

Можно добавлять **к каждому объекту** совой обработчик смены кадров. Можно создать **единый** обработчик `onEnterFrame` для всего ролика. Так как скорость смены кадров задается для ролика целиком, а не для отдельного клипа, разницы в выполнении кода не будет.




## Определение границ ролика

Положение границ рабочей области ролика *(в пикселях)* можно определять двумя способами:

**1. Использование глобальных свойств и ключевого слова _root:**

`_root._width` - общая ширина всех объектов ролика,
`_root._height` - общая высота всех объектов ролика.

При этом будут возвращаться размеры той прямоугольной области, в которой размещаются **все объекты ролика**. Для пустого ролика размеры будут равны *нулю*.

**2. Использование свойств класса Stage**

`Stage.width` - ширина области публикации ролика,
`Stage.height` - высота области публикации ролика.

При этом **всегда** будут возвращаться те значения, которые Вы установили в качестве размеров рабочей области ролика (Modify > Document > Dimensions).

## Объект Math

Объект *Math* предназначен для работы с математическими функциями и константами, без которых не обходиться создание реалистических анимаций и эффектов.

**Математические константы**

`Math.PI`
Число π . Активно используется в расчетах в связи с тем, что тригонометрические функции принимают значения (а обратные тригонометрические функции возвращают результат) в радианах. *Приблизительно равняется 3,14159265358979.*

`Math.E`
Число e, экспонента. *Приблизительно равно 2,71828182845905*.

`Math.SQRT2`
Корень из двух. Активно используется в связи с тем, что корень из двух входит в выражения значений тригонометрических функций в узловых точках (например, 30°, 60°).
*Приблизительно 1,4142135623731.*

`Math.SQRT1_2`
Корень из 1/2. Полезен, так как именно такому значению равняются синус и косинус от 45°. *Приблизительно 0,707106781186548.*

`Math.LN2`
Натуральный логарифм от 2. Полезен при переходе от натуральных логарифмов к логарифмам по основанию 2.
*Приблизительно 0,693147180559945.*

`Math.LN10`
Натуральный логарифм от 10. Позволяет переходить от натуральных логарифмов к десятичным. *Приблизительно 2,30258509299405.*

`Math.LOG2E`
Логарифм от числа e по основанию 2. *Приблизительно 1,44269504088896.*

`Math.LOG10E`
Логарифм от числа e по основанию 10. *Приблизительно 0,434294481903252.*

Все перечисленные математические константы являются **бесконечными иррациональными дробями**. Хранимые же свойствами объекта Math, их значения приблизительны. Это приводит к дополнительным ошибкам в расчетах, и, например, отчасти поэтому Math.sin(Math.PI) **не** равняется в точности нулю.

**Математические функции**

*Возведение в степень.* Возводит число X в степень Y

```
pow(x: Number, y: Number) : Number
trace(Math.pow(2,3));     // Выводит: 8
```

*Квадратный корень.* Извлечение квадратного корня из числа X

```
sqrt(x: Number) : Number
trace(Math.sqrt(9));      // Выводит: 3
```

*Экспоненциальная функция.* Возведение числа e в степень X ( или вычисление $e^X$)

```
exp(x: Number) : Number
trace(Math.exp(2));       // Выводит: 7.38905609893065
```

*Натуральный логарифм.* Вычисление натурального логарифма числа X

```
log(x: Number) : Number
trace(Math.log(10))       // Выводит: 2.30258509299405
```

*Модуль.* Вычисление модуля числа X

```
abs(x: Number) : Number
trace(Math.abs(-10))      // Выводит: 10
```

*Сравнение пары чисел.*
Выводит наименьшее число из пары X и Y

```
min(x: Number, y: Number) : Number
trace(Math.min(5, -10)); // Выводит: -10
```

Выводит наибольшее число из пары X и Y

```
max(x: Number, y: Number) : Number
trace(Math.max(5, -10)); // Выводит: 5
```

**Тригонометрические функции**

*Прямые тригонометрические функции:*
sin и cos возвращают значение в радианах
360° соответствует 2π радиан, т.е. одному градусу эквивалентно π /180 радиан

```
sin(x: Number): Number
cos(x: Number): Number
tan(x: Number): Number
```

**Пример:**

```
trace(Math.cos(Math.PI));      // Выводит: -1
```

*Обратные тригонометрические функции (аркфункции):*
Возвращают угол по значению функции

```
acos(x: Number):Number
asin(x: Number):Number
atan(tangent: Number):Number          (задается угол)
atan2(y: Number, x: Number):Number (задаются координаты точки на круге)
```

**Примеры:**

```
trace(Math.asin(1)/Math.PI*180);              // Выводит: 90
trace(Math.acos(0)/Math.PI*180);              // Выводит: 90
trace(Math.atan(Infinity)/Math.PI*180);       // Выводит: 90
```

(*Infinity* - математическая бесконечность)

**Округление чисел**

Округление до **ближайшего** целого (дробная часть 0,5 округляется в большую сторону)

```
round(x: Number) : Number
trace(Math.round(112.5));     // Выводит: 113
```

Округление до **наименьшего** ближайшего целого

```
floor(x: Number) : Number
trace(Math.floor(-112.8));     // Выводит: -113
```

Округление до **наибольшего** ближайшего целого

```
ceil(x: Number) : Number
trace(Math.ceil(-112.8));     // Выводит: -112
```

**Генерация случайных чисел**

Функция `Math.random()` возвращает псевдо случайные дробные числа `n`, где `0 <= n < 1`

Функция `random(X)` возвращает псевдо случайные целые числа `n`, где `0 <= n < X-1`

```
random(100);   // Выводит целые числа от 0 <= n < 99
```

**Пример.** Функция возвращает случайные числа в заданном диапазоне.

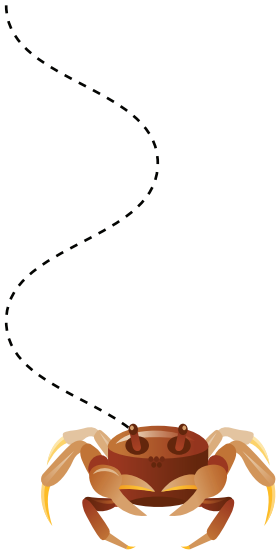
```
// в качестве агрументов принимаются границы диапазона:
function randRange(min:Number, max:Number):Number {
     var randomNum:Number     =    Math.floor(Math.random() *
                                   (max - min + 1)) + min;

    return randomNum; // вызвращается случайное число
}
// в цикле выводится 10 случайных чисел n, где 4 <= n <= 11:
for (var i = 0; i < 10; i++) {
    var n:Number = randRange(4, 11);    // вызов функции
    trace(n);  // отладочный вывод случайного числа
}
```

## Программное движение объекта по кривой

При простой программной анимации объект движется по *прямой линии*. Чтобы создать анимацию по некоторой кривой, необходимо **математически описать** траекторию движения.

Рассмотрим **волнообразное** движение объекта. Для описания данной траектории движение подходят функции **sin** и **cos**. При этот координату объекта по одной оси просто увеличиваем, а вторую координаты рассчитываем по формуле:
`Math.cos(angle*Math.PI/360)*amplitude` или
`Math.sin(angle*Math.PI/360)*amplitude`, где

`angle` – переменная, задающая угол
(ее необходимо изменять в процессе анимации)
`amplitude` – переменная, задающая амплитуду
(не изменяется при анимации)



**Пример.** Создадим клип *«mc_move»*. Чтобы создать для него волнообразное движение сверху вниз поместим в кадр временной следующий код:

```
// объявим переменные с параметрами движения:
var angle = 0; // угол, задает начальное смещение
var amplitude = 5;  // амплитуда движения

// добавляем обработчик:
mc_move.onEnterFrame = function() { // при каждой смене кадров
     // увеличиваем угол:
     angle += 5;

     // передвигаем объект:
     // рассчитываем координату X
     this._x += Math.cos(angle*Math.PI/360)*amplitude;

     // просто увеличиваем координату Y
     this._y += 3;

     // если вышли за границы ролика
     if (this._y > Stage.height) {
          // вернуть клип на исходное положение по оси Y
          (выше границы видимости ролика)
          this._y = - this._height;
     }
}
```

## Управление объектами с помощью клавиатуры

### Особенности событий клавиатуры

Flash-плейер может реагировать на клавиатурный ввод только, если он находиться **в фокусе** (окно плеера активно). В противном случае события клавиатурного ввода будет получать то приложение, которое активно в данный момент. Это может создавать сложности при воспроизведении flash-ролика на HTML-странице, (фокус будет на браузере, а не плеер).

**Перевести фокус ввода на flash-ролик можно:**
- Либо создав **кнопку** (например, PLAY) без нажатия не которую ролик не начнет воспроизведения. Нажав эту кнопку, пользователь переведет фокус на flash-плеер.
- Либо автоматически перевести фокус на flash-плеер, используя метод **`focus()`** JavaScript. (более предпочтительный способ).

**У flash-плейера**, как у большинства программ, есть **свои «горячие»** клавиши и сочетания. Чтобы отключить их поддержку, нужно добавить в ролик следующий код:

```
fscommand ("trapallkeys", "true");
```

*Функции fscommand не работают при тестировании ролика в программе Flash.*
*«Горячие» клавиши операционной системы и браузера блокировать нельзя.*

В процессе тестирования ролика большинство клавиш срабатывать не будет. Чтобы **отключить** собственные «горячие» клавиши **программы Flash**, необходимо *в меню flash-плеера* выполнить команду **Control > Disable Keyboards Shortcuts** (галочка должна быть установлена).

### Обработка клавиатурного ввода

Существует несколько способов обработки клавиатурного ввода. Исторически первым появилось событие кнопок `keyPress`, которое позволяло «отлавливать» нажатие клавиш. Однако этот способ устарел и практически не используется.

### События onKeyDown и onKeyUp

Событие *onKeyDown* происходит, когда пользователь **нажимает** клавишу на клавиатуре, а событие *onKeyUp* возникает, в момент **отпускания** клавиши.

Событие клавиатуры onKeyDown **существенно отличается** от события мыши onMouseDown. Событие **onMouseDown** происходит **один раз** при нажатии левой кнопки мыши. При этом не имеет значения, удерживалась ли кнопка или сразу же была отпущена. Событие **onKeyDown при удерживании** клавиши будет **повторяться** до тех пор, пока пользователь не отпустит кнопку. Как часто будет повторяться данное событие, задается настройками операционной системы (в среднем 10-30 событий в секунду).

Чтобы обработать события onKeyDown и onKeyUp, нужно создать «**слушатель события**» *(Listener)*. Для этого необходимо создать новый объект и добавить его в список «слушателей событий» объекта Key, при помощи метода `add Listener()`.

**Пример.** При нажатии любой кнопки клавиатуры в панели отладочного вывода Output появиться некоторое сообщение.

```
var obj:Object = {};      // создаем пустой объект
Key.addListener(obj);     // добавляем его в список «слушателей»

// создаем обработчик события:
obj.onKeyDown = function():Void {
     trace("Нажата кнопка");
}
```

**Удалить** объект из списка слушателей можно, используя метод `removeListener()`.

**Клипы** тоже могут «слушать» клавиатурный ввод, но для этого надо **установить фокус** на нужный клип. Тот объект, который находиться в фокусе будет отмечен желтой рамкой.

**Пример.** Создаем клип *«key_mc»*, рисуем в нем любой графический объект. Чтобы изменять свойства клипа при нажатии на кнопку (например, передвигать по оси X) добавляем в ключевой кадр временной шкалы следующий код.

```
key_mc.focusEnabled = true; // разрешаем клипу принять фокус
Selection.setFocus(key_mc); // устанавливаем фокус на данный клип

// Добавляем обработчик:
key_mc.onKeyDown = function():Void {
     this._x += 10; // перемещаем клип
}
```

Если одновременно было нажато **несколько клавиш**, для каждой из них произойдут **индивидуальные** события onKeyDown и onKeyUp. Однако используя данные события, можно лишь узнать, что некоторая клавиша была нажата или отпущена. Какая же именно это была клавиша, определяется при помощи методов `getCode()` и `getAscii()` объекта Key.

**Методы getCode()и getAscii()**

Метод `getCode()` возвращает код последнее **нажатой клавиши**. При этом возвращаемое значение не зависит от текущей раскладки клавиатуры и регистра. Например, русским символам «а», «А» и латинским «f», «F» будет соответствовать одинаковый код, так как все они находятся на одной клавише.

Метод `getAscii()` возвращает код последнего **введенного символа**. Возвращаемый код уникален для каждого символа и зависит от раскладки и регистра. Код символа соответствует стандартной таблице **ASCII** *(American Standard Code for Information Interchange - американский стандартный код для обмена информацией)*.

```
trace("Код клавиши: " + Key.getCode() );
trace("ASCII Код символа:" + Key.getAscii() );
```

Если ни одна клавиша не была нажата, оба метода возвращают ноль.

**Управление объектом. Проверка нажатия конкретной клавиши**

Метод `isDown()` позволяет определить нажатие конкретной клавиши. Кроме того этот метод может регистрировать одновременное нажатие нескольких клавиш и нажатие кнопок мыши.

В качестве аргумента передается код клавиши *(Key code, не зависящий от раскладка и регистра)*. Кроме того, все **системные клавиши** зарегистрированы в качестве свойств объекта Key, что позволяет обращаться к ним по «именам свойств» *(Key Property)*.

Используя коды клавиш можно обратиться к любым клавишам клавиатуры.
*Таблицы кодов приведены в Приложении.*

При **управлении объектом** необходимо постоянно проверять нажаты или нет управляющие клавиши. В таком случае проверку событий клавиатуры помещают **внутрь обработчика** `onEnterFrame` (проверка смены кадров).

**Пример.** Создадим клип *«car»* и будет передвигать его по рабочей области ролика с помощью клавиш-стрелок.

```
// обработчик события смены кадров:
car.onEnterFrame = function () {
      if (Key.isDown(Key.LEFT)    // обращение по имени свойства
            car._x -= 5;
      if (Key.isDown(Key.RIGHT)) // обращение по имени свойства
            car._x+ = 5;
      if (Key.isDown(38))         // обращение по коду клавиши - UP
            car._y -= 5;
      if (Key.isDown(40))         // обращение по коду клавиши - DOWN
            car._y += 5;
}
```

## Мониторинг пересечения клипов

Используя метод `hitTest()`, можно определить, *пересекаются два клипа* между собой или нет. Это особенно важно при движении объекта. Когда имеются некоторые препятствия, которые либо ограничивают область движения, либо изменяют характер данного движения (например, затормаживают или ускоряют).

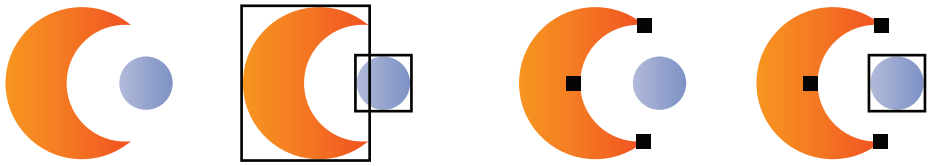
Так же метод `hitTest()` может определить *принадлежит* некоторая *точка клипу* или нет.
*Пример такого использования будет рассмотрен на занятии 5.*

Чтобы определить наличие пересечения двух клипов *«mc_1»* и *«mc_2»*, расположенных на главной временной шкале, нужно в кадре данной шкалы ввести следующий код:
`mc_1.hitTest(mc_2);` или `mc_2.hitTest(mc_1);`

*Порядок указания клипов не важен.*
В качестве **результата** метод `hitTest()` возвращает логическую величину `true` (истина), если клипы имеют общие точки, и `false` (ложь), если они не пересекаются.

К сожалению, **точность** метода `hitTest()` *не очень велика*, так проверка пересечения клипов происходит только **вдоль границ прямоугольной рамки**, ограничивающей клипы, а не по всей площади клипов. Поэтому, чем *ближе форма клипа к прямоугольнику*, тем *выше точность* данного метода. Для клипов сложной формы результат может оказать не приемлемым.

Чтобы добиться хорошего результата **для клипов сложной формы**, внутрь данного клипа необходимо вложить несколько **прямоугольных клипов** *(«датчиков»)*. И проверять соприкосновение, не клипа целиком, а всех его датчиков.



есть пересечение нет пересечение

Если нужно определять пересечения **движущихся** клипов, то метод `hitTest()` помещают внутрь обработчика смены кадров (`onEnterFrame`). При этом метод `hitTest()` будет возвращать величину `true` (истина) до тех пор, пока клипы полностью не разойдутся в пространстве ролика.

**Пример.** Создадим клип *«obj»*, который будет перемещать с помощью клавиатуры. Создадим еще два клипа *«border»* и *«ground»*. Если клип *«obj»* соприкоснется с клипом *«border»*, он должен отскочить (начать движение в обратную сторону). При пересечении с клипом *«ground»* скорость движения клипа *«obj»* должна уменьшиться, направление движения не меняется.

Код, реализующий данный пример, представлен ниже и должен быть помещен в кадр главной временной шкалы ролика:

```
// объявляем переменные:
var k_move;                 // коэффициент перемещения объекта
var k_direct = 1;           // направление движения:
                            // 1 = слева направо, -1 = справа налево
var k_speed;                // величина скорости движения

// добавляем обработчик смены кадров:
this.onEnterFrame = function() {

      // расчет коэффициента перемещения объекта
      k_move = k_direct * k_speed;

      // перемещение объекта с помощью кнопок-стрелок:
      if (Key.isDown(Key.LEFT))
            obj._x -= k_move;
      if (Key.isDown(Key.RIGHT))
            obj._x += k_move;
      if (Key.isDown(Key.UP))
            obj._y -= k_move;
      if (Key.isDown(Key.DOWN))
            obj._y += k_move;
```

```
      // определение пересечения клипов:
      if (obj.hitTest(ground)) {  // пересечение с кликом ground:
            k_speed = 2;          // замедляем движения
      }
      else {
            k_speed = 5;          // ускоряем движение
      }

      if (obj.hitTest(border)){   // пересечение с кликом border:
            k_direct = - k_direct;      // отталкиванием клип
      }
}
```

# Практическая работа

*Все файлы, необходимые для выполнения данной работы, находятся на раздаточном диске в папке «**lesson_03\Practice_Work_3**» (включая примеры выполненных заданий).*

**Задание 1. Управление объектами с помощью клавиатуры и мониторинг пересечения клипов**

1. Откройте файл ***start.fla***. Перемещая клип с черепахой с помощью клавиатуру, нужно захватывать клипы с ракушками и перемещать их на клип с крабом. Задание считается выполненным, когда будут перемещены все пять ракушек.

Все объекты в ролике уже нарисованы и имеют уникальные имена экземпляров:
• `obj` *(черепаха)* – клип, перемещаемый с помощью клавиатуры
• `border_1 - border_3` *(морские звезды)* – клипы, при пересечении с которыми движущийся объект отталкивается
• `shell_0 - shell_4` *(ракушки)* – клипы, которые необходимо перемещать
• `crab` *(краб)* – клип, на который нужно переместить все ракушки.

2. Вначале откорректируйте клип с крабом. Для этого двойным щелчком войдите в режим редактирования эталона клипа «*crab*». Внутри клипа находятся 5 клипов-индикаторов `ball_0 - ball_4`. Изначально их нужно сделать не видимыми.

3. Также нужно погасить видимость клипа «*hit*», по которому определяется пересечения с ракушками.

4. Для этого внутри символа «*crab*» в слое «*action*» поместите следующий код:

```
hit._visible = false;
for (i=0; i<5; i++){
      this["ball_"+i]._visible = false;
}
```

5. Весь последующий код нужно разместить на главной временной шкале ролика в слое «*action*». Код примера разделен на несколько блоков. Первый блок *"ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ"* уже прописан в файле ***start.fla***:

```
var k_move;           // коэффициент перемещения объекта
var k_direct = 1;     // направление движения:
                      // 1 = слева направо, -1 = справа налево
var k_speed = 5;      // величина скорости движения

var shellNum;         // номер перетаскиваемой ракушки
var shellMax = 5;     // максимальное количество перетаскиваемых ракушек
var shellWell = 0;    // количество уже перенесенных ракушек

// массив переменных, определяющий перенесена данная ракушка или еще нет
var shellMoved=new Array();

// массив переменных, определяющих будет ли ракушка двигаться вместе с
объектом
var shellMove = new Array();

// заполнение обоих массивов начальными значениями:
for (i=0; i<shellMax; i++){
     shellMove[i] = false;
     shellMoved[i] = false;
}
```

7. Чтобы объект (черепаха) перемещался с помощью клавиатуры необходимо в блоке *"ОБРАБОТЧИК СМЕНЫ КАДРОВ"* добавить следующий код:

```
// расчет коэффициента перемещения объекта
k_move = k_direct * k_speed;

// перемещение объекта с помощью кнопок-стрелок:
if (Key.isDown(Key.LEFT)){  // если нажата указанная кнопка:
     obj._x -= k_move;      // перемешаем объект
}
if (Key.isDown(Key.RIGHT)){
     obj._x += k_move;
}
if (Key.isDown(Key.UP)) {
     obj._y -= k_move;
}
if (Key.isDown(Key.DOWN)){
     obj._y += k_move;
}
```

8. Определяем пересечение клипа «*obj*» с клипами «*borderX*»:

```
if   (obj.hit.hitTest(border_1) || obj.hit.hitTest(border_2) ||
     obj.hit.hitTest(border_3)){
          k_direct = - k_direct;   // отталкиванием клип
}
```

9. Чтобы при отпускании клавиш клавиатуры направление движения возвращалось к исходному, необходимо в блоке *"ОБРАБОТЧИК ОТПУСКАНИЯ КНОПОК КЛАВИАТУРЫ"* добавить соответствующий обработчик:

```
// добавляем обработчик:
var k1:Object = new Object();
Key.addListener(k1);

// функция обработчика:
k1.onKeyUp = function(){         // при отпускании указанной клавиши:

      if(Key.getCode() == 37){    //LEFT
            // восстанавливаем исходное направление движения
            k_direct = 1;
      }
      if(Key.getCode() == 39){    //RIGHT
            k_direct = 1;
      }
      if(Key.getCode() == 38){    //UP
            k_direct = 1;
      }
      if(Key.getCode() == 40){    //DOWN
            k_direct = 1;
      }
}
```

Отдельные события на отпускание каждой из кнопок нам понадобятся в дальнейшем.

10. Теперь определяем пересечение движущегося клипа с ракушками. Для этого добавляем в блок *"ОБРАБОТЧИК СМЕНЫ КАДРОВ"* следующий код:

```
// пересечения с ракушками:
for (i=0; i<5; i++){
      if (obj.hit.hitTest(_root["shell_"+i]))
            // ракушка будет перемещаться вместе с объектом
            shellMove[i] = true;
      else
            // ракушка не будет перемещаться вместе с объектом
            shellMove[i] = false;
}
```

11. Теперь нужно модернизировать коды, перемещения объекта.
В блоке *"ОБРАБОТЧИК СМЕНЫ КАДРОВ"* для каждой клавиши теперь нужно перемещать не только сам объект (obj), но и все ракушки, в которыми есть пересечение:

```
if (Key.isDown(Key.LEFT)){  // если нажата указанная кнопка:
      obj._x -= k_move;     // перемешаем объект
      for (i=0; i<5; i++){  // обходим все ракушки
            // если ракушка движется вместе с объектом
            if (shellMove[i])
                  // перемещаем ракушку
                  _root["shell_"+i]._x -= k_move;
      }
}
```

Аналогичные действия нужно сделать для всех управляющий клавиш.

12. В блоке *"ОБРАБОТЧИК ОТПУСКАНИЯ КНОПОК КЛАВИАТУРЫ"* так же нужно изменить код. Теперь при отпускании кнопок не только возвращается исходное направление движения, но и отбрасываются в противоположную сторону все ракушки, с которыми есть пересечение. Это позволяет предотвратить «залипание» соприкоснувшихся объектов.

```
if(Key.getCode() == 37){    //LEFT
      // восстанавливаем исходное направление движения
      k_direct = 1;
      for (i=0; i<5; i++){  // обходим все ракушки
            // если ракушка движется вместе с объектом
            if (shellMove[i])
                  // отталкиваем ракушку в противоположном направлении
                  _root["shell_"+i]._x -= k_move;
      }
}
```

Аналогичные действия нужно сделать для всех управляющий клавиш.

13. Чтобы при каждом запуске ролика ракушки попадали в разные точки ролика, необходимо задавать их координаты как случайное число. При этом может возникнуть такая ситуация, что ракушки попадут в закрещенную зону (например, в область морской звезды). Для того, чтобы можно было заново распределить объекты, при неудачном попадании введена кнопки «*распределить ракушки заново*» (имя экземпляра `b_random`). Чтобы реализовать произвольное распределение нужно в блок *"ПОЛЬЗОВАТЕЛЬСКИЕ ФУНКЦИИ"* добавить следующий код:

```
// произвольное распределение ракушек:

// объявление функции:
function shellRandom(){
      for (i=0; i<5; i++){
            _root["shell_"+i]._x = random(Stage.width);
            _root["shell_"+i]._y = random(Stage.height - 200) + 200;
      }
}
// первичный вызов функции:
shellRandom();

// повторное распределение по кнопке:
b_random.onRelease = function(){
      shellRandom();
}
```

14. Осталось определить пересечение ракушек и краба.
Для этого в блок *"ОБРАБОТЧИК СМЕНЫ КАДРОВ"* добавляем следующий код:

```
// пересечение с крабом:
for (i=0; i<5; i++){
      if (crab.hit.hitTest(_root["shell_"+i])){
            // ракушка становиться не видимой
            _root["shell_"+i]._visible = false;
            // в массиве отмечаем данную ракушку, как перенесенную
            shellMoved[i] = true;
```

```
            // вызов функции, определяющей количество перенесенных ракушек
            changeMass();
            // включаем видимость клипа-индикатора
            crab["ball_"+(shellWell-1)]._visible = true;
      }
}
```

15. Пользовательская функция `changeMass()` нужна для определения количества уже перенесенных ракушек. ЕЕ необходимо объявить в блоке *"ПОЛЬЗОВАТЕЛЬСКИЕ ФУНКЦИИ"*:

```
// подсчет количества перенесенных ракушек:
function changeMass(){
      shellWell = 0;              // обнуляем переменную счетчик
      for (j=0; j<5; j++){        // обходим все ракушки
            if(shellMoved[j])     // если уже перенесена
                  shellWell++;    // увеличиваем переменную счетчик
      }
}
```

16. Самое последнее, нужно отследить момент, когда все объекты успешно доставлены на место – это считается успешно выполненным заданием! В блок *"ОБРАБОТЧИК СМЕНЫ КАДРОВ"* нужно добавить следующий код:

```
// конец игры:
if (shellWell == shellMax){      // если перенесены все ракушки
      crab._xscale = 80;         // увеличиваем краба
      crab._yscale = 80;
}
```

*Пример готового задания находиться в файле **sample.fla***

# Занятие 4

*Продолжительность 180 минут*

**Темы занятия:**

1. Динамическая загрузка данных из библиотеки ролика

2. Работа с текстом

3. Загрузка мультимедиа-ресурсов из внешних файлов

# Тема 1. Динамическая загрузка данных из библиотеки ролика

## Присоединение клипов из библиотеки данного ролика

Экземпляры символов можно создавать динамически. Метод `attachMovie()` позволяет программно создавать экземпляры клипов, находящихся в библиотеке данного ролика.

Чтобы динамически присоединить клип из библиотеки, его **эталону** нужно п**рисвоить идентификатор**. Для этого необходимо:
1. Выделить эталон клипа в библиотеке ролика (палитра **Library**)
2. Открыть свойства клипа (**Symbol Properties**) и в блоке **Linkage** (Связывание) установить флажок **Export for ActionScript** (Сделать доступным для ActionScript).
3. При этом станет доступно поле **Identifier** (Идентификатор), в которое нужно ввести уникальный идентификатор для программного доступа к этому символу.
4. Так же автоматически будет установлен параметр **Export in frame 1**. Благодаря этому параметру все символы загружаются в *первый* кадр ролика. И при воспроизведении (в основном через Интернет) не возникает пауз на загрузку необходимых символов.

После назначения идентификатора применяем метод `attachMovie()`.
*Его синтаксис представлен ниже:*

`clip.attachMovie(«id», «name» , depth, [initObject])`, где
`clip` – имя клипа, в который будет добавлен экземпляр.
Если оно не задано клип будет добавлен на главную временную шкалу.
«`id`» - строка, задающая идентификатор того символа, который требуется загрузить
«`name`» - строка, задающая имя создаваемого экземпляра (Instance Name)
`depth` – номер виртуального слоя, на который будет помещен экземпляр
`[initObject]` – необязательный параметр, задающий свойства экземпляра

**Важно:**
1. На один виртуальный слой можно загрузить только один объект.
2. Если загружать объект на занятый виртуальный слой, новый объект вытолкнет предыдущий.
3. Содержимое виртуального слоя с большим номером закрывает собой содержимое слоя с меньшим номером слоя. Виртуальные слои закрывают собой содержимое обычных слоев.

Метод `attachMovie()` возвращает *ссылку* на созданный экземпляр. Экземпляр помещается в **точку начала координат** (точку регистрации) того клипа, куда он был загружен. Если клип загружался в главную временную шкалу *(_root)*, то он попадет в верхний левый угол ролика.

**Задать положение клипа** (так же как и другие свойства экземпляра) можно с помощью необязательного параметра `initObject`. Свойства указываются внутри фигурных скобок.

**Пример:**
// для клипа с идентификатором *«map_1»* создаем экземпляр, который помещается в точку x=0, y=0 главной временной шкалы:

```
attachMovie ("map_1", "map_1_mc", 1);
```

// чтобы изменить свойства загруженного экземпляра, обращаемся к нему по имени

экземпляра, присвоенному при создании:
```
map_1_mc._xscale = 50;
```

// для клипа с идентификатором *«map_2»* создаем экземпляр, который помещаем внутрь клипа *«load_mc»* и задаем координаты (будут отсчитываться от точки регистрации клипа *«load_mc»*):
```
load_mc.attachMovie("map_2", "map_2_mc", 0, {_x:50, _y:100});
```

// изменяем свойства загруженного клипа:
```
load_mc.map_2_mc._xscale = 50;
```

В качестве номера уровня можно задать параметр `this.getNextHighestDepth()`, который автоматически вычисляет номер следующего свободного виртуального слоя.

## Удаление закруженных объектов

Для удаления **динамически** созданных экземпляров служит метод `removeMovieClip()`. Он требует указания только одного параметра – **имя экземпляра** (Instance Name) того символа, который необходимо удалить.

**Пример.** Удаление программно созданного экземпляра *«map_1_mc»*:
При этом допустимы два синтаксиса применения метода:
```
map_1_mc.removeMovieClip();
removeMovieClip(map_1_mc);
```

## Работа с общими ресурсами во время выполнения

Если имеет проект, в котором несколько swf-роликов должны работать с одинаковыми символами, то можно создать **библиотеку совместного доступа** *(shared libraries)*.

В этом случае в один swf-файл помещают все общие символы (например, элементы интерфейса). Другие файлы динамически загружают из него нужные символы. Библиотеки совместного доступа позволяют уменьшать количество скачиваемой информации, поэтому они активно используются при создании сайтов.

**Пример**. Создадим библиотеку совместного доступа
1. Создадим файл *«library.fla»*, *в нем будут храниться общие символы.*
При публикации он получит имя *«library.swf»*.
2. Внутрь файла поместим символ кнопки и откроем свойства символа (**Symbol Properties**).
3. В блоке **Sharing** (Общий доступ) установим параметр **Export for runtime sharing** (Экспортировать при выполнении). Присвоим идентификатор (**Identifier)** *«button_1»*. После чего в поле **URL** укажем имя, которое получит файл после публикации *«library.swf»*.

4. Создадим второй файл *«clip.fla»*, *он будет обращаться к библиотеке.*
5. Создадим в файле *«clip.fla»* пустой символ и откроем его свойства (**Symbol Properties**).
6. В блоке **Sharing** (Общий доступ) установим параметр **Import for runtime sharing** (Импортировать при выполнении). В поле идентификатор (**Identifier)** укажем нужный символ - *«button_1»*. А в поле **URL** укажем имя файла библиотеки общего доступа - *«library.swf»*.

Созданный пустой символ можно использовать в файле *«clip.fla»*. При публикации в него автоматически будет загружено соответствующее содержимое из файла *«library.fla»*.

# Тема 2. Работа с текстом

## Динамические текстовые поля и поля ввода

В отличие от **статических** *(Static)* текстовых полей значения, выводимые в **динамические** поля *(Dynamic)* и **поля ввода** *(Input)*, могут быть **изменены программно**. В поля ввода пользователь также может вводить значения в процессе выполнения ролика.

Для динамических полей и полей ввода программно можно изменять не только, хранимый в них текст, но и *параметры форматирования* (цвет текста, шрифт и т.д.), а также п*араметры самого поля* (положение на рабочей области, размер и т.д.).

Все параметры статических полей, *кроме изменения ориентации текста, верхних и нижних индексов*, **доступны** для динамических полей и полей ввода.

**Дополнительно доступны следующие параметры:**

- **Render text as HTML** *(Применять HTML теги)*. Если параметр включен, то отобраается результат применения тегов. В противном случае теги HTML отображаются, как обычные текстовые символы.
- **Show border around text** *(Показать рамку вокруг текста)*. При включении данного параметра вокруг текстового поля отображается черная рамка с белым фоном (стиль не настраивается).
- **Behavior** Отвечает за перенос строк. Доступны следующие варианты:
  ***Single line*** (Однострочный) - весь текст в одну строку
  ***Multiline*** (Многострочный) - строки автоматически переносятся
  ***Multiline no wrap*** (Многострочный без переносов) – каждый абзац текста отображается единой строкой.
- **Character Embedding** *(Внедрение глифов - символов шрифта)*. Параметр не доступен, если используются шрифты устройств.

**Внедрение шрифтов**

Для динамических полей и полей ввода сохраняются используемые шрифты, а не векторные контура символов (как для статических полей).

Если необходимо использовать шрифт, отличный от системных шрифтов, у большинства пользователей его может не быть. В этом случае необходимо внедрить символы шрифта в ролик. Для этого нажимаем кнопку **Character Embedding** *(Внедрение символов)* и в появившемся диалогом окне выбираем необходимые символы.

Наиболее часто используются следующие наборы:

| | |
|---|---|
| *Punctuation* – знаки пунктуации | *Numerals* - только числа |
| *Basic Latin* – базовая латиница | *Cyrillic* – кириллические символы |

Наборы символов можно **дополнить**, введя необходимые символы в поле **Include these characters** *(Включить символы)*. Кнопка **Auto Fill** – автоматически выбирает все различные символы из числа тех, что в данный момент введены в динамическое поле или поле ввода.

После того, как выбраны все требуемые символы, необходимо **нажать кнопку OK**.

Символы будут внедрены, и при этом несколько *увеличиться объем ролика*. Если важно сохранить малый объем ролика, следует внедрять символы экономно.

Если используются **шрифты устройств** или внедрение не производилось, объем ролика *не увеличивается*. Но если у пользователя отсутствует нужный шрифт, произойдет автоматическая замена на ближайший установленный шрифт.

Чтобы **отменить** внедрении символов, внизу диалогового окна **Character Embedding** *(Внедрение символов)* необходимо нажать **кнопку Don't Embed**. Внедрение всех символов для **данного** текстового поля будет отменено.

## Чтение и запись текста

Чтобы программно управлять динамическим полем или полем ввода, ему необходимо **присвоить имя экземпляра** *(Instance Name)*.Тексту, находящемуся в динамических полях и полях ввода, соответствует свойство `text`. С его помощью можно считывать и записывать текст в поле.

**Пример.** Создадим динамическое поле и присвоим ему имя экземпляра *«text_field»*. Чтобы **записать** значение в данное поле, нужно в ключевой кадр поместить следующий код:

```
text_field.text = "new text";  // вывести текст
```

При этом предыдущий текс будет удален.

Чтобы **добавить** текст, нужно использовать составной оператор присваивания (+=).

```
text_field.text += " !";       // добавить текст
```

**Переход на новую строчку** можно осуществить с помощью сочетания «`\n`»:

```
text_field.text += "\n !!";    // переход на новую строку
```

Можно выводить не только текстовые строки, но и **значения переменных**.
При этом числовые переменные будут автоматически преобразованы в строковые данные.

```
var forText = 33;
text_field.text += forText;    // добавить значение переменной
```

Для **чтения** текущего значения текстового поля также используется свойство text.

```
trace(text_field.text);        // считать текст из поля
```

## Форматирование текста с помощью тегов HTML

Кроме свойства `text` для записи в динамические поля и поля ввода можно использовать свойство `htmlText`. В этом случае текс можно форматировать с помощью HTML тегов.

**Доступны следующие теги:**

`<P>` - параграф `<BR>` - перенос строки
`<A>` - гипер-ссылка `<LI>` - элементы списка (разделитель - кружок)
`<B>` - жирный `<I>` - курсив `<U>` - подчеркнутый
`<FONT>` - форматирование текста `<IMG>` - изображение
`<TAB>` - *не стандартный* - задает в тексте символ табуляции
`<TEXTFORMAT>` - *не стандартный* - расширенное форматирование
`<SPAN>` - предназначен для применения классового стиля CSS к произвольным символам

**Пример. Форматирование текста.**

```
text_field.htmlText = "<B><I><FONT SIZE='20'>Форматирование &quot;
HTML&quot; тегами</FONT></I></B>";
```

**Служебные символы** в тексте необходимо заменять:

| | | | |
|---|---|---|---|
| `&gt` - | соответствует «<» | `&lt` - | соответствует «>» |
| `&quot` - | задает кавычки | `&amp` - | вводит амперсанд (&) |

Другие символы во flash не поддерживаются.

Для вывода **специальных символов** используется конструкция:
«`&#number;`», где `number` —десятичный номер символа в кодировке ASCII

**Пример:**

```
text_field.htmlText += "&#169; 2009";     // © 2009
```

**Пример использования нестандартного тега <TAB>:**

```
text_field.htmlText+="1<TAB>2<TAB><TAB>3\t\t\t4";
//результат:   1    2         3               4
```

**Нестандартный тег <TEXTFORMAT>** предоставляет дополнительные возможности форматирования:

- `LEFTMARGIN, RIGHTMARGIN` - отступы слева и справа соответственно
- `INDENT` - отступ первой строки абзаца
- `BLOCKINDENT` - отступ абзаца слева
- `LEADING` - расстояние между строками
- `TABSTOPS` - определяет, какое смещение строки в пунктах вызовет каждый символ табуляции

**Пример:**

```
text_field.htmlText+="<TEXTFORMAT    LEADING='-5'    LEFTMARGIN='30'>
Абзац текста с дополнительным форматирование. </TEXTFORMAT>";
```

## Загрузка данных из внешних текстовых файлов

Когда в ролике имеется множество текстовых данных, то эффективнее загружать их из внешнего файла, а не внедрять в ролик. Во-первых, это дает возможность *редактировать текст без использования* программы Flash и перекомпиляции ролика (например, заказчик в случае надобности может самостоятельно исправить номер телефона в ролике-заставке сайта). Во-вторых, множество текста может замедлить компиляцию ролика.

Текстовые данные можно загружать из `txt` и `xml`-файлов.
*В этом курсе мы рассматриваем загрузку из txt-файлов.*

В данном случае **каждому динамическому полю** должна соответствовать одна **текстовая переменная**, текст которой храниться во внешнем текстовом файле. Внутрь ролика текст не помещают, а только определяют стиль форматирование для каждого из текстовых полей.

**Пример.** Создадим ролик, имитирующий аудио-плеер. Текстовые данные загружаем из файлов.

**1. Создание динамических текстовых полей**
Добавим на главную шкалу ролика 3 динамических текстовых поля. Первому полю присвоим имя экземпляра *«text_version»*, в него будем выводить версию плеера. Второму полю присвоим имя *«text_track»*, оно будет соответствовать названию музыкальной композиции. Третье поле назовем *«text_author»*, в него будем выводить название автора трека.

**2. Создание текстовых файлов**
Создадим 2 текстовых файла. Первый назовем *«player_info.txt»*, второй – *«music_info.txt»*.

**3. Объявление текстовых переменных**
В одном текстовом файле можно объявить произвольное количество текстовых переменных. При этом формат должен быть следующим:
`name_1 = text 1& name_2 = text 2 & …. & nameN = text N,` где
`name_1, name_2 … nameN` – произвольные имена переменных
`text 1, text 2 … text N` – соответствующий им текст
Переменные разделяются между собой знаком «**&**».

Объявим в файле *«player_info.txt»* одну переменную:
```
plVersion = 1.0
```

В файле *«music_info.txt»* объявим две переменные:
```
track1 = Rock And Roll & author1 = LED ZEPPELIN
```

**4. Загрузка текстовых файлов для разбора**
При загрузке внешних текстовых данных крайне важен вопрос **кодировки**. Программа Flash по умолчанию считает все загружаемые данные указаны в кодировке **Unicode** (поддерживаются варианты UTF-8, UTF-16 BE (Big Endian) и UTF-16 LE (Little Endian)). Файлы, созданные в стандартном редакторе Windows («Блокноте»), будет иметь кодировку **win-1251**, что приведен к искажению русских букв при загрузке текста.

Чтобы программа Flash интерпретировала загружаемые файлы, исходя из кодировки установленной по умолчанию в операционной систему нужно в первом кадре ролика поместить следующий код:
```
System.useCodepage = true;  // кодировка Операционной системы
```

Чтобы **загрузить** текстовый файл используется метод `loadVariablesNum`. Данный метод не возвращает никакого значения и принимает следующие параметры:
`loadVariablesNum(«url», level, [«method»]),` где
`«url»` - путь к файлу
`level` - номер программного уровня, на который будет загружен файл
`[«method»]` - *необязательный параметр* - метод GET или POST

**Важно:**
На один программный уровень можно загрузить только один файл.
Если уровень занят предыдущее содержимое «выталкивается».
Номер программного уровня задается неотрицательным числом.
Нулевой уровень (_level0) соответствуют главной временной шкале ролика (_root)

Загрузим файл *«player_info.txt»* на уровень «0», а файл *«music_info.txt»* - на уровень «1»:
```
loadVariablesNum("player_info.txt", 0); // файлы, находятся в той же папке,
loadVariablesNum("music_info.txt", 1);  // что и swf-ролик
```

Можно указывать абсолютный или относительный путь к файлу. Относительная адресация предпочтительнее:
```
//loadVariablesNum("source/sample.txt", 0);   // файл лежит внутри папки
//loadVariablesNum("../sample.txt", 0);       // файл на один уровень выше
```

**5. Связывание динамических полей и текстовых переменных.**
После того, как все необходимые файлы загружены, нужно указать какая переменная соответствует каждому динамическому полю. При этом используется свойство динамических полей `variable`:
`fieldName.variable = "textVar"`, где
`fieldName` – имя экземпляра динамического поля
`"textVar"` – строка задающая путь к текстовой переменной

Для файлов, загруженный на уровень ноль «**0**», к переменным можно обращаться 3-я способами:
```
fieldName.variable = "/:name_1";
fieldName.variable = "_root. name_1";
fieldName.variable = "_level0. name_1";
```

Для файлов загруженные на другие программные уровни, только одним способом:
`fieldName.variable = "_level`***N***`. name_1";, где` ***N*** `- номер уровня`

**В нашем примере** связать поля и переменные нужно следующем образом:
```
text_version.variable = "plVersion";
text_track.variable = "track1";
text_author.variable = "author1";
```

Вместо программного задания свойства `variable` можно прописать его на палитре **Properties** *(Инспектор свойств)*.

## Прокрутка текста – реализация простого «скрола»

Для прокручивания динамических текстовых полей используют свойство `scroll`. Данное свойство задает номер строки, которая будет первой отображаться в текстовом поле.
Так же могут быть полезны следующие свойства:
- `bottomScroll` - номер последней отображаемой строки
- `maxscroll` - максимально допустимое значение свойства scroll. Значение `maxscroll` равно общему числу одновременно отображаемых строк текста + 1.

**Пример**. Создадим динамическое текстовое *"text_field"*. По нажатию на кнопку *"b_up"* прокручиваем текст на одну строчку вверх, по кнопке *"b_up"* - прокручиваем текст вниз:

```
b_up.onRelease = function(){
     text_field.scroll--;
}
```

```
b_up.onRelease = function(){
     text_field.scroll++;
}
```

*Оператор* ***--*** *(декремент) уменьшает переменную на 1,* ***++*** *(инкремент) увеличивает на 1.*

# Тема 3. Загрузка мультимедиа-ресурсов из внешних файлов

## Динамическая загрузка swf-роликов

Для загрузки внешних swf-файлов используются методы `loadMovie()`, `loadMovieNum()`.

Метод `loadMovie()` является более универсальным. Он позволяет загрузить содержимое внешнего файла **в экземпляр клипа**. При этом, если клип был не пустым, его содержимое удаляется. Данный метод имеет следующие варианты синтаксиса:
`loadMovie("url", target, [method])` и
`target.loadMovie("url")`, где
`"url"` – строка, задающая путь к загружаемому swf-ролику
`target` – экземпляр клипа, в который будет загружен файл
`[«method»]` - *необязательный параметр* - метод GET или POST

**Пример.** Загрузим в клип с именем *«toLoad_mc»* содержимое файла *«movie.swf»:*
```
toLoad_mc.loadMovie("movie.swf "); или
loadMovie("movie.swf ", "toLoad_mc");
```

Метод `loadMovieNum()` позволяет загрузить внешний файл, не в клип, а **на программный уровень**. Он имеет следующий синтаксис:
`loadMovieNum("url", level, [method])`, где
`"url"` – строка, задающая путь к загружаемому swf-ролику
`level` – номер программного уровня
`[«method»]` - *необязательный параметр* - метод GET или POST

**Ограничения**. При использовании метода `loadMovieNum()` загруженный ролик помещается в точку x=0, y=0 (верхний левый угол). Если загружаемый ролик в свою очередь загружает внешние данные, при использовании данного метода они не будет отображаться.

**Пример**. Загрузим файл *«movie.swf»* на программный уровень «**2**».
```
loadMovieNum("movie.swf", 2);
```

Интересный эффект будет при загрузке внешнего файла на **уровень «0»** или в главную временную шкалу **«_root»**. *Исходный ролик полностью исчезнет.* Размеры ролика будут изменены под размеры загруженного файла.
**Пример:**
```
_root.loadMovie("movie.swf ");
loadMovieNum("movie.swf", 0);
```

## Удаление объектов

Для того чтобы удалить **содержимое клипа** используется метод `unloadMovie()`. Сам клип при этом не удаляется. Для данного метода возможно два варианта синтаксиса:
`clip.unloadMovie()` или
`unloadMovie(clip)`, где `clip` – имя удаляемого экземпляра клипа

Для очистки **программного уровня** применяется метод `unloadMovieNum()`. Все содержимое уровня при этом удаляется. В качестве аргумента указывается номер очищаемого уровня:

```
unloadMovieNum(3);   // удаляется 3-й программный уровень
```

## Загрузка внешних графических файлов

Загрузка растровых изображений ничем **не отличается** от загрузки внешних swf-файлов. Для загрузки изображений используются методы `loadMovie()` и `loadMovieNum()`. Поддерживаются следующие форматы **JPEG**, **GIF** и **PNG**.

**Пример.**
// загрузка изображения *«pict.jpg»* внутрь клипа *«toLoad_mc»:*

```
toLoad_mc.loadMovie("pict.jpg");
```

// загрузка изображения «pict.jpg» на программный уровень 2:

```
loadMovieNum("pict.jpg",2);
```

## Загрузка аудио файлов, управление звуком

Для программного управления звуками необходимо создать объект класса `Sound`:

```
var my_sound:Sound = new Sound();
```

Далее в созданный объект можно *добавить звук из библиотеки* ролика или *загрузить* его из внешнего **mp3-файла** (другие форматы файлов не поддерживаются).

Для присоединения из библиотеки используется метод `attachSound()`. В качестве аргумента он требует идентификатор, задаваемый в свойствах эталона символа. Задание идентификатора для звуковых объектов осуществляется также как и в случае клипов *(рассмотрено ранее)*.

**Пример.** Присоединение звука с идентификатор «sound_1»:

```
my_sound.attachSound("sound_1"), где
my_sound – созданный ранее объект класса Sound
```

Для загрузки из внешнегофайла применяется метод `loadSound()`:

```
snd.loadSound(URL, isStreaming), где
```

`snd` - созданный ранее объект класса Sound
«`URL`» - путь к загружаемому mp3-файлу
`isStreaming` – определяет тип загрузки. Значение «`true`» соответствует потоковой загрузке. При значении «`false`» звук должен быть полностью загружен до начала воспроизведения ролика (событийные звуки).

**Пример.** Присоединение звука из файл «sound.mp3» в потоковом режиме:

```
my_sound.loadSound("sound.mp3", true)
```

После загрузки звук сразу начинает воспроизведение.
**Для управления доступны следующие методы:**

`my_sound.stop()` – остановить воспроизведение
`my_sound.start()` – начать воспроизведение
`my_sound.setVolume(`***N***`)` – назначить громкость, где ***N*** задает громкость звука , Величина ***N*** принимает значения от 0 до 100.

# Практическая работа

*Все файлы, необходимые для выполнения данной работы, находятся на раздаточном диске в папке «**lesson_04\Practice_Work_4**» (включая примеры выполненных заданий).*

**Задание 1. Загрузка данных из внешних файлов**

1. Откройте файл **start.fla**. В этом задании нарисован магнитофон, из внешних файлов будут загружать аудио треки, текстовые данные и графические изображения. Все данные подготовлены и находятся в папке «*lesson_04\source*». Информация подготовлена для 8 различных треков.

2. Основной код разместите в слое «*action*» главной временной шкалы:

```
// объявляем переменные:
var track_number = 0;                    // номер исполняемого трека
var max_track = 8;                       // общее количество треков
var my_sound:Sound = new Sound();        // создаем звуковой объект

mafon_mc.stop();                         // остановить анимацию магнитофона
```

3. Для воспроизведения аудио файла, соответствующего выбранному треку объявите пользовательскую функцию, которую будет вызываться по нажатию кнопки PLAY:

```
function music_play() {
      // загрузка аудио-файла
      my_sound.loadSound("../source/0"+track_number+".mp3", true);
      // вывод номера воспроизводимого трека
      track_num.text = track_number;
      // воспроизведение анимации
      mafon_mc.play();
}
```

4. Динамическому текстовому полю, в которое выводиться номер трека присвойте имя экземпляра «*track_num*». Так же присвойте уникальные имена экземпляров всем кнопкам магнитофона. В нашем примере кнопки имеют следующие имена:

- *b_play* – начать воспроизведение
- *b_stop* – остановить воспроизведение
- *b_next* – переход к следующему треку
- *b_prev* – переход к предыдущему треку
- *b_info* – показать информацию о треке

6. Добавьте обработчики для основных кнопок:

```
b_play.onRelease = function(){          // PLAY
      if(!track_number){           // при первом нажатии:
            track_number = 1 ;     // устанавливаем первый трек
      }
      music_play();                // запускаем анимацию и звук
}
b_stop.onRelease = function(){          // STOP
      mafon_mc.stop();       // остановить анимацию
      my_sound.stop();       // остановить звук
}
```

```
b_next.onRelease = function(){         // NEXT
      if (track_number < max_track){
            track_number++;        // увеличить номер трека
            music_play();          // запускаем анимацию и звук
      }
}
b_prev.onRelease = function(){         // PREV
      if (track_number > 1){
            track_number--;        // уменьшить номер трека
            music_play();    /     / запускаем анимацию и звук
      }
}
```

7. Кнопка *«b_info»* должна загружать клип с дополнительной информацией из библиотеки ролика. Поэтому сначала нужно сделать данный клип доступным для управления с помощью ActionScript. Для этого в библиотеке ролика выберите эталон с именем «*dop_info*» и в параметрах данного символа (**Symbol Properties**) в блоке связывание (**Linkage**) установите флажок **Export for ActionScript**. В качестве идентификатора оставьте «*dop_info*»

8. Чтобы информация о треке появлялась при наведении мыши на кнопку дополнительной информации, и исчезала при отведении мыши, добавьте в слой «action» следующий код:

```
// загрузка из библиотеки клипа с дополнительной информацией:
b_info.onRollOver = function(){
      if(track_number){
            info_load.attachMovie("dop_info", "dop_inf_mc", 1);
      }
}

// удаление клипа с дополнительной информацией:
b_info.onRollOut = function(){
      if(track_number){
            info_load.dop_inf_mc.removeMovieClip();
      }
}
```

9. Клип будет появляться, но пока он не содержит нужной информации. Чтобы отобразить название трека и изображение в эталоне символа «*dop_info*» в первом кадре слоя «*action*» разместите следующий код:

```
// загрузка внешнего текстового файла:
loadVariablesNum("../source/0"+_root.track_number+".txt", 0);

// отображение текстовый переменных в соответствующий динамических полях:
text_artist.variable = "_root.band_name";
text_albom.variable = "_root.albom_name";
text_track.variable = "_root.track_name";

// загрузка внешнего изображения:
pict_load.loadMovie("../source/0"+_root.track_number+".jpg");
```

*Пример готового задания находиться в файле **sample.fla***

# Занятие 5

*Продолжительность 180 минут*

**Тема занятия:**

Работа с мышью и создание эффектов

# Работа с мышью и создание эффектов

## Создание эффекта «падающий снег»

Суть эффекта «падающий снег» или однотипных эффектов заключается в том, что создается один эталонный символ (например, снежинка). Программно создается множество экземпляров данного символа, которые произвольно «разбрасываются» по области ролика. Чтобы объекты не выглядели одинаковыми, для каждого экземпляра задаются индивидуальные свойства (масштаб, прозрачность и т.д).

Для программного создания экземпляров клипов применяются два метода `attachMovie()` и `duplicateMovieClip()`.

Метод `attachMovie()`, рассмотренный ранее, создает новый экземпляр, основываясь на эталоне символа из библиотеки. Метод `duplicateMovieClip()` создает копию экземпляра символа, который уже находиться на рабочей области ролика.

**Метод duplicateMovieClip()**

Данный метод имеет следующий синтаксис:
`clip.duplicateMovieClip("name", depth)`, где
`clip` – имя экземпляра, который должен быть скопирован
`"name"` – имя, которое получит новый экземпляр
`depth` – номер виртуального слоя

При создании новый экземпляр помещается в туже точку, где находился копируемый клип.

**Пример 1. Создания эффекта «падающий снег» с использованием метода attachMovie()**

1. Создадим новый клип и нарисуем в нем снежинку. **Эталон** клипа сделаем доступным для ActionScript (**Export for ActionScript**) и присвоим ему **идентификатор** *«snow»*.
2. В ключевом каре главной временной шкалы введем следующий код:

```
var snowNumber = 100; // количество снежинок

for (i=1; i<=snowNumber; i++) {  // в цикле
      // добавляем снежинку из библиотеки:
      _root.attachMovie("snow", "snow_"+i, i);

      // задаем произвольные координаты:
      _root["snow_"+i]._x = Math.random()*Stage.width;
      _root["snow_"+i]._y = Math.random()*Stage.height;
}
```

3. Код, перемещающий снежинки и задающий их индивидуальные свойства в этом примере разместим не на главной шкале, а **внутри эталона клипа-снежинки**:

```
// функция определяющая параметры данной снежинки:
function setParam (){
      this.maxSnowSpeed = random(4)+1; // скорость падения
      this.scaleK = random(60)+10;     // коэффициент  масштабирования
      this._xscale = scaleK;           // масштабирование по оси X
      this._yscale = scaleK;           // масштабирование по оси Y
      this._alpha = random(80)+10;     // прозрачность
      this._rotation = random(360);    // поворот
}

setParam (); // определить стартовые параметры

// при каждой смене кадров:
this.onEnterFrame = function() {
      // перемещение по оси Y (падение снежинки):
      this._y += maxSnowSpeed;

      // если достигнут низ ролика:
      if (this._y > Stage.height) {
            this._y = -10;   // вернуть снежинку наверх
            setParam ();     // определить новые параметры снежинки
      }
}
```

**Пример 2. Создания эффекта «падающий снег» с помощью метода duplicateMovieClip()**

1. Создадим новый клип и нарисуем в нем снежинку. Поместим его экземпляр выше границ рабочей области и присвоим **имя экземпляра** *«snow»*.
2. В ключевом каре главной временной шкалы введем следующий код:

```
var snowNumber = 100;       // количество снежинок

for (i=0; i<snowNumber; i++) {   // в цикле
      // дублируем клип со снежинкой
      newSnow = snow.duplicateMovieClip("snow"+i, i);

      // задаем произвольные координаты (выше области видимости y < 0):
      newSnow._x = Math.random()*Stage.width;
      newSnow._y = - Math.random()*Stage.height;

      // задаем параметры снежинки:
      // смещение по оси Y (скорость полета):
      newSnow.maxSnowSpeed = Math.random()*4;
      // смещение по оси X (поправка на ветер):
      newSnow.wind = Math.random()*7;
      // коэффициент масштабирования и прозрачности:
      snowK = Math.random()*10+25;
      newSnow._xscale = newSnow._yscale = snowK;  // масштаб
      newSnow._alpha = snowK;                     // прозрачность

      // при каждой смене кадров ролика:
      newSnow.onEnterFrame = function() {
```

```
            // поворачиваем снежинку:
            this._rotation -= this.maxSnowSpeed+1;

            // если снежинка вышла за границы ролика:
            if (this._y>Stage.height || this._x>Stage.width || this._x<0){

                  // задаем новые координаты (выше области видимости y < 0)
                  this._x = Math.random()*Stage.width;
                  this._y = - Math.random()*Stage.height;
            }
            // если не вышла:
            else {
                  // перемещаем снежинку вниз:
                  this._y += this.maxSnowSpeed+1;
                  this._x += this.wind-4;
            }
      }
}
```

*В первом примере* снежинки равномерно распределятся по всей области ролика. *Во втором* - они будут появляться в верхней части ролика и лишь постепенно закроют все доступную площадь.

## Создание пользовательского курсора

Чтобы создать собственный курсор необходимо, во-первых, погасить видимость системного курсора. Во-вторых, создать собственный клип-курсор и заставить его перемещаться за мышью.

За работу с мышью отвечает объект **Mouse**. Отключить видимость системного курсора можно с помощью метода `hide()` данного объекта:

```
Mouse.hide();   // отключить видимость системного курсора
```

Вернуть видимость позволяет метод `show()`:

```
Mouse.show();   // включить видимость системного курсора
```

**Перемещать** клип-курсор за мышью можно двумя различными способами:
1. Назначать клипу координаты указателя мыши: `_xmouse` и `_ymouse`
2. Использовать метод `startDrag()`.

Если курсор представлен одним клипом, то удобнее использовать метод `startDrag()`. *Подробно будет рассмотрен позднее*. Если же за указателем мыши должны перемещаться несколько клипов, то необходимо назначать им координаты указателя мыши.

*При создании клипа-курсора нужно правильно выбрать положение **точки регистрации** (обозначается крестиком и задается для эталона). Так именно эта точка принимает координаты указателя мыши.*

**Пример. Для создания собственного курсора необходимо:**

1. Нарисовать курсор и конвертировать его в клип.
2. Поместить экземпляр клипа-курсора вне области видимости ролика и назначить ему имя экземпляра (например, *«cursor_mc»*).
3. На главную шкалу ролика поместить следующий код:

```
cursor_mc.onEnterFrame = function(){
     Mouse.hide();          // отключить видимость системного курсора

     // начать перемещение клипа-курсора вслед за мышкой:
     startDrag(cursor_mc, true);
}
```

**Перемещение маски вслед за курсором мыши**

Чтобы перемещать маску за указателем мыши, необходимо фурму, задающую маску, конвертировать в клип.

**Пример. Создадим составной курсор.** Клип *«cursor_mc»* будет видимой частью курсора. А клип *«cursor_mask_mc»* будет задавать форму маски. Чтобы перемещать оба клипа за мышью добавим на главную временную шкалу следующий код:

```
this.onEnterFrame = function(){
     Mouse.hide();          // отключить видимость системного курсора

     // обоим клипам, образующим курсор, назначить координаты мыши:
     cursor_mc._x = cursor_mask_mc._x = _xmouse;
     cursor_mc._y = cursor_mask_mc._y = _ymouse;
}
```

**Использование «слушателей» события**

В предыдущих примерах слушателем событий мыши являлись клипы. Однако можно использовать в качестве слушателя объект класса **Object**. Для этого его надо добавить в список слушателей с помощью метода `addListener()`.

**Пример.** Создание пользовательского курсора с использованием «слушателя» события. Создадим клип-курсор *"cursor_mc"* и и поместим в кадр главной шкалы следующий код:

```
var ml:Object = new Object();     // создаем "слушатель события"
Mouse.addListener(ml);            // добавляем его в список слушателей

ml.onMouseMove = function(){      // функция "слушателя"
     Mouse.hide();                // отключить видимость системного курсора
     startDrag(cursor_mc, true); // начать перемещение
     updateAfterEvent();          // принудительное обновление экрана
}
```

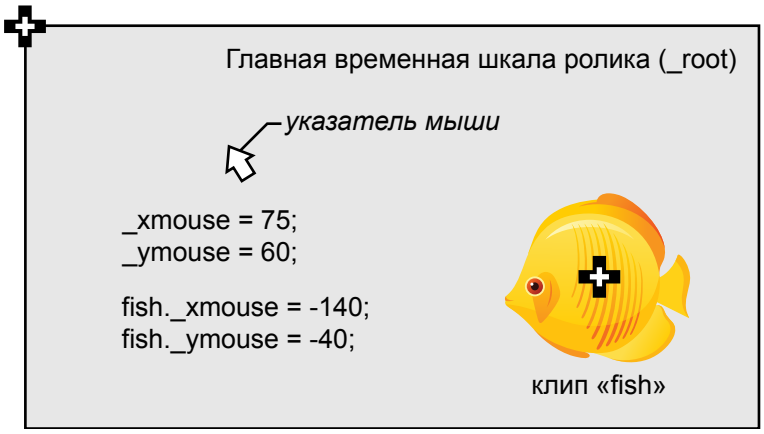
## Отслеживание координат курсора

Свойства `_xmouse` и `_ymouse` позволяют получить текущие координаты указателя мыши. Эти свойства доступны **только для чтения**.

Важно, что возвращаемые ими значения отсчитывается **от точки начала координат** (точки регистрации) того клипа, в качестве свойств которого они вызываются. Если целевой **клип не задан**, то координаты отсчитываются от *главной временной шкалы* (`_root`), для которой точка начала координат находиться в верхнем левом углу ролика.

**Пример.** Следующие две команды *эквивалентны* и возвращают одинаковое значение:
`trace(_xmouse);` и `trace (_root._xmouse);`

Команды `trace(_xmouse);` и `trace(clip._xmouse);`
в общем случае будут возвращать *разные значения*. Значения совпадут только, если точка регистрации клипа *«clip»* совпадает с точкой регистрации главной временной шкалы.




## Локальные и глобальные координаты

В программе Flash *каждый клип обладаем собственной системой координат*. Точка регистрации устанавливается для каждого эталона клипа и задает для него положение начала осей координат.

Если необходимо сравнить положение клипа, вложенного внутрь другого клипа, с некоторой точкой на главной временной шкале, то необходимо выполнить преобразование локальных координат клипа в глобальные координаты главной шкалы.

Для этого предназначен метод `localToGIobal()`, имеющий следующий синтаксис:
`clip.localToGlobal(point)`, где

`clip` - имя экземпляра клипа, чью систему координат требуется перевести в глобальную

`point` - объект класса Object. В свойствах "x" и "y" данного объекта должны быть сохранены преобразуемые координаты. Результат произведенных вычислений метод `localToGlobal()` также записывает в эти свойства.

При преобразовании системы координат метод `localToGlobal()` **учитывает** все преобразования клипа (масштабирование и поворот не повлияют на корректность его работы).

**Пример.** На главной временной шкале размещен клип *«flower»*, центр его координат находиться в точке x = 275, y = 200. Внутрь данного клипа поместим еще один клип *«move_mc»*, который имеет координаты x = 25, y = 50. Преобразуем локальные координаты клипа *«move_mc»* в глобальные координаты ролика. Для этого на главную шкалу поместим следующий код:

```
// создаем объект, координаты которого будут преобразовываться:
var myPoint:Object = {x:flower. move_mc._x, y:flower. move_mc._y};

// преобразуем координаты:
flower.localToGlobal(myPoint);

//выводим глобальные координаты
trace("x= " + myPoint.x + " y= " + myPoint.y); // x = 300,  y = 250
```

При необходимости можно выполнить **обратное преобразование** из глобальной системы координат в локальную. Для этого предназначен метод `globalToLocal ()`, во многом аналогичный предыдущему.

**Пример.** Преобразуем глобальные координаты x = 50, y = -50 в локальные координаты клипа *«flower»*, центр его координат находиться в точке x = 275, y = 200:

```
// создаем объект, координаты которого будут преобразовываться:
var myPoint2:Object = {x:50, y:-50};

// преобразуем координаты:
flower.globalToLocal(myPoint2);

// выводим локальные координаты:
trace([myPoint2.x, myPoint2.y]); // Выводит: -225, -250
```

## Определение принадлежности точки клипу

Для того, чтобы определить принадлежит ли точка клипу, следует использовать метод `hitTest()` в следующей форме:
`clip.hitTest(x, y, shapeFlag)`, где:

- `clip` – имя экземпляра клипа, на принадлежность к которому проверяется точка
- `x` и `y` – координаты точке, для которой проверяется принадлежность
- `shapeFlag` – логический параметр. При значении «`true`» принадлежность точки проверяется только по отношению к залитым пикселям клипа. При значении «`false`» проверяется все площадь рамки, ограничивающей клип.

**Пример.** Будем проверять точку, определяемую текущими координатами курсора, на принадлежность заполненным пикселям клипа с именем *«flower»*.

```
this.onEnterFrame = function():Void {
     if (flower.hitTest(_xmouse, _ymouse, true))
          trace («Попали в клип!»);
}
```

## Буксировка клипов - реализация «Drag and Drop»

Для буксирования (перетаскивания) клипов используется методы `startDrag()` и `stopDrag()`. Первый метод **начинает** буксирование (клип перемещается вслед за указателем мыши). Второй **прекращает** буксировку клипа.

Метод `startDrag()` имеет следующий синтаксис:
`startDrag(target, [lock, left, top, right, bottom])`, где

`target` – имя экземпляра буксируемого клипа

`[lock]` – *необязательный параметр* – Значение «`true`» накладывает ограничения на область перемещения объекта. При этом последовательно задаются левая (left), верхняя (top), правая (right) и нижняя (bottom) границы области. Значения границ задаются в пикселях относительно начало координат родительского клипа. При значении «`false`» - объект может перемещаться по всей площади ролика.

Имя целевого клипа не обязательно передавать как параметр. Можно вызывать метод `startDrag()` для нужного клипа *(например, «target»)*, используя следующий синтаксис:
`target.startDrag();`

Метод `stopDrag()` требует указания только одного параметра – имя экземпляра, для которого нужно остановить буксировку:
`startDrag(target);` *или* `target.startDrag();`
где `target` – имя экземпляра клипа.

**Пример.** Создадим клип *«drag_mc»*, который можно перемещать по области ролика с помощью мыши. Пока нажата левая кнопка мыши объект буксируется. При отпускании, буксировка прекращается. Следующий код нужно поместить в кадр главной временной шкалы:

```
// при нажатии кнопки мыши:
drag_mc.onPress = function(){
      // начать буксировку (область перемещения не ограничена):
      this.startDrag();

      // область перемещения ограничена:
      //startDrag(this, true, 225,50,225,500);
}

// при отпускании кнопки мыши:
drag_mc.onRelease = function(){
      stopDrag();      // остановить буксировку
}

// при выведении мыши за область клипа (кнопки мыши нажата):
drag_mc.onDragOut = function(){
      stopDrag();      // остановить буксировку
}
```

Внутри обработчика события клипа, если требуется остановить буксировку данного клипа метод `stopDrag()` можно применять *без указания имени целевого клипа*. Он автоматически будет применен к тому клипу, в чьем обработчике был вызван данный метод.

## Создание пустого клипа

При создании всевозможных эффектов часто возникает задача создать пустой клип. В этот клип в последствии программно загружают необходимые данные (другой клип, изображение и т.д.).

Для создания пустых экземпляров клипов используется метод `createEmptyMovieClip()`: `createEmptyMovieClip(name, depth)`, где

`"name"` - строка, задающая имя создаваемого экземпляра

`depth` - номер виртуального слоя

Данный метод в**озвращает ссылку** на созданный экземпляр клипа.
Параметр `this.getNextHighestDepth()` автоматически подставляет номер следующего свободного слоя.

## Создание эффекта «шлейф мыши»

Эффекты «шлейф мыши» весьма разнообразны. Однако принцип создания в основном сводиться к следующему:

- В библиотеке ролика создается статический или анимированный элемент «шлейфа»
- При перемещении мыши динамически создаются экземпляры этого элемента, которые перемещаются вслед за указателем мыши.

**Пример 1. Поднимающиеся пузырьки.** Шлейф с произвольным количеством элементов.

Для создания данного эффекта необходимо:
1. Создать клип и нарисовать в нем элемент шлейфа. Создать внутри данного клипа анимацию – пузырек поднимается вверх и при этом исчезает (прозрачность уменьшается до нуля). Чтобы избежать зацикливания в конце анимации, нужно добавить команду `stop();`

2. Для клипа с анимированным пузырьком разрешить программный доступ к эталону (**Export for ActionScript**) и назначить ему идентификатор *«bubble»*

3. После чего добавить на главную временную шкалу следующий код:

```
// создаем пустой клип:
this.createEmptyMovieClip("cursor_mc", 50);

// создаем "слушатель события":
var ml:Object = new Object();
Mouse.addListener(ml);

// функция "слушателя":
ml.onMouseMove=function()
{
      // создаем ссылку "trail", которая будет указывать на элемент "bubble_mc"
      // добавленный из библиотеки в клип "cursor_mc":
      var trail:MovieClip = cursor_mc.attachMovie("bubble",
      "bubble_mc", cursor_mc.getNextHighestDepth());
```

```
      // перемещаем и масштабируем клип “trail”:
      trail._x=_xmouse;
      trail._y=_ymouse;
      trail._xscale = trail._yscale =(Math.random()*100)+20;
}
```

**Пример 2. «Шлейф» фиксированной длины.**

В данном примере 7 кружков перемещаются за указателем мыши. Для реализации нужно:
1. Создать клип и нарисовать в нем элемент «шлейфа»
2. Задать данному клипу идентификатор «trail_element»
3. На главную временю шкалу поместить следующий код:

```
// параметры “шлейфа” мыши:
var trail_array:Array = Array(7);       // число элементов
var speed = 3;                          // скорость
var distance = 10;                      // промежутки между элементами
var distance_mouse = 20;                // расстояние от курсора до первого элемента

//  в цикле for для каждого элемента “шлейфа”:
for (var i = 0; i<trail_array.length; i++) {
      // создаем пустой клип:
      var trail_mc = _root.createEmptyMovieClip(i+”mc”, i);

      // добавляем в него клип “trail_element” из библиотеки:
      trail_mc.attachMovie(“trail_element”, “trail”+i,
                           trail_mc.getNextHighestDepth());

      if (i) {   // ВСЕ КРОМЕ ПЕРВОГО ЭЛЕМЕНТА:
            // определяется предыдущий элемент:
            trail_mc.prevClip = _root[(i-1)+”mc”];
            // при каждой смене кадров:
            trail_mc.onEnterFrame = function() {
                  // данный элемент перемещается за предыдущим элементом:
                  this._x += (this.prevClip._x - this._x + distance)/speed;
                  this._y += (this.prevClip._y - this._y)/speed;
            }
      }
      else {           // ПЕРВЫЙ ЭЛЕМЕНТА:
            // при каждой смене кадров
            trail_mc.onEnterFrame = function() {
                  // первый элемент перемещается за мышью:
                  this._x += (_root._xmouse - this._x + distance_mouse)/speed;
                  this._y += (_root._ymouse - this._y)/speed;
            }
      }
}
```

# Практическая работа

*Все файлы, необходимые для выполнения данной работы, находятся на раздаточном диске в папке «**lesson_05\Practice_Work_5**» (включая примеры выполненных заданий).*

**Задание 1. Создание пользовательского курсора**

1. Создайте новый документ

2. Нарисуйте клип, который будет являться курсором

3. Разместите экземпляр клипа вне рабочей области ролика и присвойте ему уникально имя экземпляра (например, cursor_mc).

4. В ключевой кадр главной временной шкалы поместите добавьте следующий код:

```
// создаем "слушатель события":
var ml:Object = new Object();

// добавляем его в список слушателей:
Mouse.addListener(ml);

// функция "слушателя":
ml.onMouseMove = function(){
      // отключить видимость системного курсора:
      Mouse.hide();
      // начать перетаскивать клип-курсор вслед за мышкой
      startDrag(cursor_mc, true);
      // принудительное обновление экрана
      updateAfterEvent();
}
```

5. При желании сделайте клип, преследующим курсор мыши или добавьте эффект «шлейф мыши».

*Пример готового задания находится в файле **01_sample.fla***

**Задание 2. Буксировка клипа**

1. Откройте файл ***02_start.fla***. В данном примере будет реализован элемент управления. При перемещении «ползунка» будет изменять значение некоторого параметра. Диапазон допустимых значений от -100 до 100.

2. Двойным щелчком войдите в режим редактировании эталона клипа «*move_bar*». Всем вложенным элементам присвойте имена экземпляров:

- *crab* – перемещаемый клип («ползунок»)
- *bar* – клип, показывающий область перемещения
- *text_field* – динамическое текстовое поле, в которое будет выводиться текущее значение контролируемого параметра.

3. Так как нам нужен независимый элемент управления, весь программный код поместите внутрь клипа «*move_bar*» в слой «*action*»:

```
// границы буксировки объекта:
var borderL = - bar._width/2 + crab._width/4;
var borderR = bar._width/2 - crab._width/4;

// значение контролируемого параметра:
var pst;
```

4. Чтобы реализовать буксировку (перетаскивание) клипа добавите следующий код:

```
// обработчики событий мыши для клипа:
crab.onPress = function(){       // нажатие кнопки мыши
     // начать буксировку (область ограничена):
     startDrag(this, true, borderL,0,borderR,0);
}

crab.onRelease = function(){     // отпускание кнопки
     this.stopDrag();            // остановить буксировку
}

crab.onDragOut = function(){     // мышь вышла за пределы объекта
     this.stopDrag();            // остановить буксировку
}
```

5. Значение контролируемого параметра определяем по взаиморасположению перетаскиваемого клипа «*crab*» и неподвижного «*bar*». При этом важно, что у обоих клипов **точки регистрации** находятся в центре и при начальном положении **координаты** обоих клипов **совпадают**. Вычисленное значение параметра округляется и выводится в динамическое текстовое поле:

```
// обработчик смены кадров:
crab.onEnterFrame = function(){
     // рассчитываем значение параметра:
     pct = (borderL - crab._x) / borderL * 100;
     // выводим его в динамическое поле:
     text_field.text = Math.round(pct-100);
}
```

6. Теперь при необходимости на главной временной шкале можно создать несколько экземпляров клипа «*move_bar*». Каждый из них может управлять значением своего параметра.

*Пример готового задания находиться в файле **02_sample.fla***

# Занятие 6

*Продолжительность 180 минут*

**Тема занятия:**

Создание простого flash-сайта

# Создание простого flash-сайта

## Создание элементов навигации. Преимущества использования клипов вместо кнопок

При создании элементов навигации можно использовать два типа символов: **клипы** (*Movie Clip*) и **кнопки** (*Button*). В обоих случаях экземпляру символа можно присвоить уникально имя экземпляра и управлять им с помощью языка ActionScript.

В **простых проектах**, где количество управляющих элементов не велико, использование **кнопок** позволяет сократить время разработки. Так как смена состояний кнопки (пассивное состояние, неведение и нажатие на кнопку) происходит автоматически.

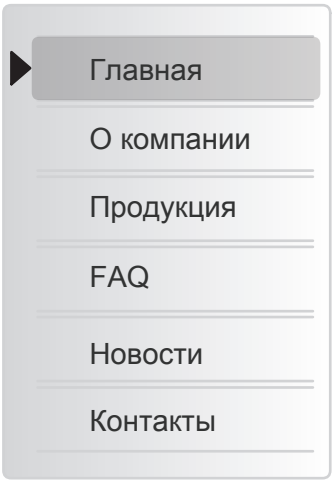
Когда речь идет о проектах **со сложной навигацией**, удобнее создать один эталонный символ, описать его поведение и программно добавлять в ролик необходимое число экземпляров данного символа. Для этой цели необходимо использовать **клипы**, так как *внутрь их эталонов* можно *помещать программные коды*, что не допускается для кнопок. Это означает что, при создании **эталонного символа** элемента навигации мы помещаем в него не только все **графические элементы**, но и **коды обработчиков** или любой другой код, реализующий поведение элемента навигации.

При этом на временной шкале клипа (подобно кнопкам) создаются **ключевые кадры** – по одному на каждое состояние элемента навигации. Основное отличие от кнопок заключается в том, что **все обработчики** смены состояний придется *прописать вручную*. Но Вы получаете возможность легко запрограммировать **дополнительные состояния**, например, элемент навигации не доступен или выбран (открыта соответствующая ему страница сайта).

**Пример использования клипа в качестве элемента навигации**

Создадим вертикальное меню сайта. При щелчке мыши на какой-либо пункт указатель активного пункта плавно перемещается на новую позицию.

1. Создадим в библиотеке ролика пустой клип (Movie Clip).
2. В первом кадре нарисуем пассивное состояние элемента навигации (мышь вне элемента). Добавим второй ключевой кадр и нарисуем в нем состояние наведения мыши.
3. Чтобы элемент изменял состояние при наведении и отведении мыши. В первый кадр его эталона добавим следующий код:




```
stop();

onRollOver = function(){
      gotoAndStop(2);
}
onRollOut = function(){
      gotoAndStop(1);
}
```

4. На главной временной шкале поместим нужное количество экземпляров элемента навигации (один под другим). *Имена экземпляров в этом примере присваивать не нужно.*

5. Создадим клип, который будет указывать на выбранный пункт меню. Поместим его на главную временную шкалу и присвоим имя экземпляра, например, `select_mc`

6. Теперь чтобы плавно перемещать указатель выбранного пункта меню (клип `select_mc`) в ключевом кадре главной временной шкалы разместим следующий код:

```
var toMove;           // переменная, показывающая точку, в которую должен
                      // переместиться клип-выделение (по оси Y)

var speed = 0.1;      // скорость перемещения выделения


// создаем пользовательскую функцию. Эта функция получает "ссылку"
// на ту кнопку, по которой щелкнул пользователь:

function StartMove (button_select){

     // присваиваем переменной toMove значение координаты Y данной кнопки
     toMove = button_select._y;
}

// при каждой смене кадров ролика перемещаем
// выделение к точке "toMove" по оси Y:

this.onEnterFrame = function() {
     select_mc._y += (toMove - select_mc._y) * speed;
}
```

7. Чтобы передать функции `StartMove()` «ссылку» на ту кнопку, по которой щелкнул пользователь в эталон элемента навигации нужно добавить следующий код:

```
onRelease = function() {
     _parent.StartMove(this);
}
```

*Ключевое слово* `this` *будет указывать на тот экземпляр элемента навигации, по которому щелкнул пользователь.*

## Динамическая загрузка содержимого страниц

Содержимое страниц flash-сайта лучше всего загружать **из внешних файлов**. Это позволяет изменять данные без использования редактора Flash и перекомпиляции flash-ролика.

Flash-ролик в данном случае становиться своего рода **шаблоном оформления**, в который включены **эффекты анимации**, позволяющие создать красочные переходы при смене данных.

Для размещения *графических изображений* создаются **пустые клипы**, для которых определяются необходимые параметры (например, положение, прозрачность и т.д.). Для *текстовых данных* создаются **пустые динамические поля** и определяются параметры их форматирования. Внешние *аудио файлы* воспроизводятся с помощью объектов класса `Sound`.

Процесс загрузки данных из внешних файлов подробно рассмотрен на Занятии 4.

## Эмуляция процесса загрузки на клиентском компьютере

Редактор Flash позволяет оценить скорость загрузки созданного ролика через Интернет, а так же помогает выявить самые «тяжелые» кадры ролика.

*Для эмуляции процесса загрузки на клиентском компьютере необходимо:*
1. Выполнить тестирование созданного ролика (**Control > Test Movie** или *CTRL+Enter*). При этом откроется окно flash-плеера.

2. В меню flash-плеера выполнить команду **View > Bandwidth Profiler** (*CTRL+B*). В верхней части окна плеера появиться **диаграмма**, показывающая какой объем данных требуется загрузить для каждого кадра ролика. На данной диаграмме по горизонтали отображаются кадры ролика, по вертикали – количество загружаемых данных (в килобайтах).

3. Далее с помощью команды **View > Download Setting** необходимо установить нужную **скорость подключения**. Скорость подключения выбирается из списка, настроить пользовательские значения можно с помощью пункта **Customize**.

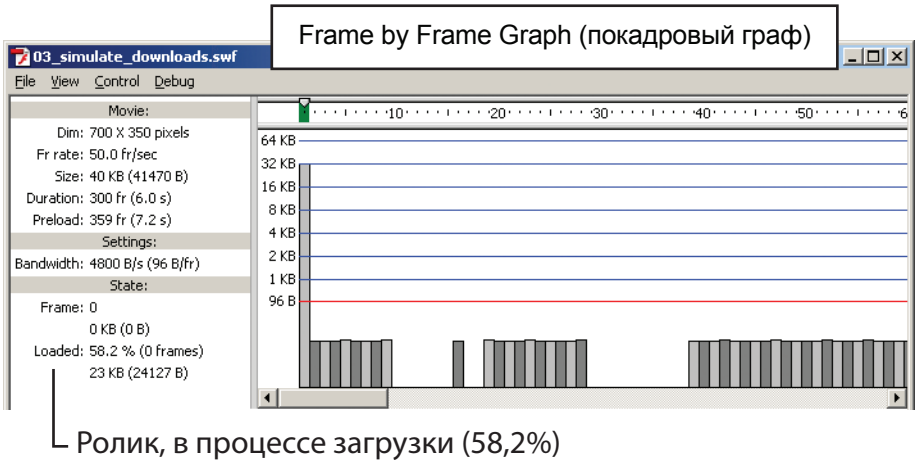
4. Чтобы **просмотреть ролик** в режиме эмуляции загрузки через Интернет необходимо в меню flash-плеера выполнить команду **View > Simulate Download** (*CTRL+Enter*).

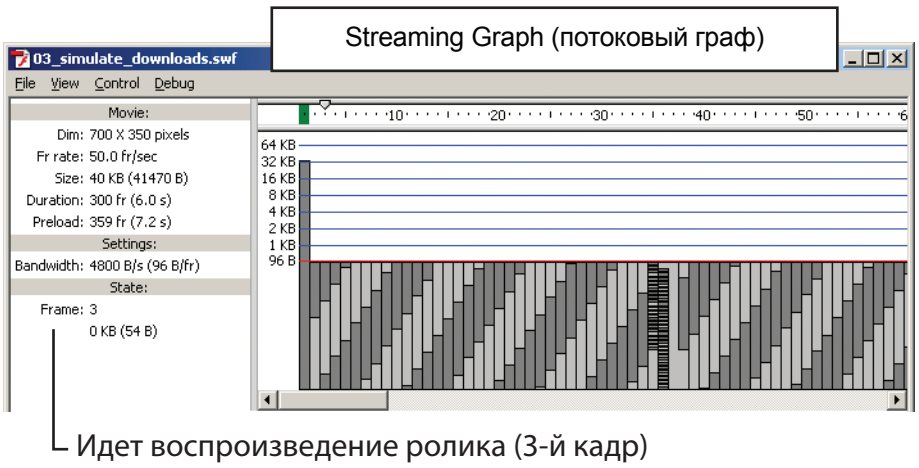
*Диаграмма загрузки ролика доступна в двух режимах:*
**Frame by Frame Graph** *(покадровый граф)*. Показывается количество данных, загружаемых непосредственно для каждого кадра ролика. При этом если некоторые кадры не требуют загрузки новых данных, на диаграмме образуются пустоты.

**Streaming Graph** *(потоковый граф)*. В реальности flash-плеер никогда не простаивает при загрузке данных ролика. Если какой-то кадр не требует новых данных, то загружаются данные для следующего кадра колика.

*Выбрать нужный режим* диаграммы можно через меню **View** flash-плеера.

Внешний вид диаграммы загрузки в обоих режимах представлен на рисунке:




**Красная горизонтальная черта** показывает максимальное количество данных, которые будут скачены без задержек при выбранной скорости подключения через Интернет.

*Если имеются «выступы» за красную линию, то ролику необходим предзагрузчик.*

Слева от диаграммы отображается информация о ролике и процессе загрузки:
**Movie** *(параметры ролика)*:

- *Dim* (dimensions) – размеры ролика (px)
- *Fr rate* (frame rate) – частота смены кадров (fr/sec)
- *Size* – объем файла в килобайтах/байтах (Kb/B)
- *Duration* – длительность (fr – кадры / s – секунды)
- *Preloaded* – примерная длительность загрузки при выбранной скорости подключения (fr – кадры / s – секунды)

**Settings** *(настройки подключения к Интернет)*:

- *Bandwidth* – пропускная способность бит/секунда (B/s) или кадр/секунда (fr/s)

**State** *(текущее состояние)*:

- *Frame* – номер воспроизводимого кадра ролика
- *Loaded* – сколько процентов ролика загружено

## Методы построения предзагрузчиков

Предзагрузчиком называется некоторая простая анимация, которая «развлекает» пользователя в то время, *пока идет процесс загрузки основного ролика*. Также как правило, отображается текущий *процент загрузки* ролика.

Классический предзагрузчик строиться на **трех-кадровом цикле.**
Суть метода заключается в следующем:

- **В первый кадр** ролика помещают **только** клип, который является предзагрузчиком.
- **Во второй кадр** размещают **скрипт**, который сравнивает *общее количество байт* в ролике и *текущее количество загруженных байт*. Если ролик **не догружен**, выполняется переход **на первый кадр**. Через интервал времени, определяемый частотой смены кадров ролика, flash-плеер снова перейдет на кадр два. К этому времени *измениться* текущее количество загруженных байт ролика.
- *Переходы между первым и вторым карами* будет происходить до тех пор, пока ролик не будет скачен полностью. Все это время пользователь будет видеть анимацию, размещенную в предзагрузчике. Предзагрузчики стараются сделать максимально «легкими», чтобы не приходилось ожидать загрузки самого предзагрузчика.
- Как только ролик будет **полностью скачен** *скрипт*, размещенный во втором кадре ролика, выполнит **переход не третий** (или любой последующий) кадр ролика. С этого кадра начнется основная анимация, а клип с предзагрузчиком будет удален с временной шкалы ролика.
- **В последнем кадре** ролика нужно сделать переход на кадр, с которого начинается основная анимация. В противном случае, так как flash-плеер при достижении конца ролика автоматически переходит на первый кадр, на долю секунды будет виден кадр с предзагрузчиком.

**Пример1. Создание простого предзагрузчика**

1. Откройте ролик, для которого необходимо сделать предзагрузчик.

2. **Сдвиньте** все кадры временной шкалы вправо так, что перед началом основной анимации образовалось минимум **два пустых кадра**. *Для редактирования множества кадров используйте опцию временной шкалы Edit Multiple Frames.*

3. Создайте два пустых слоя «*loader*» и «*action*».

4. Создайте клип, который будет являться предзагрузчиком, поместите в эталон клипа несложную анимацию. Добавьте экземпляр предзагрузчика **в первый кадр** слоя «*loader*». **Оборвите видимость** данного слоя там, где начинается основная анимация ролика (*вставки пустого ключевого кадра – F7*).

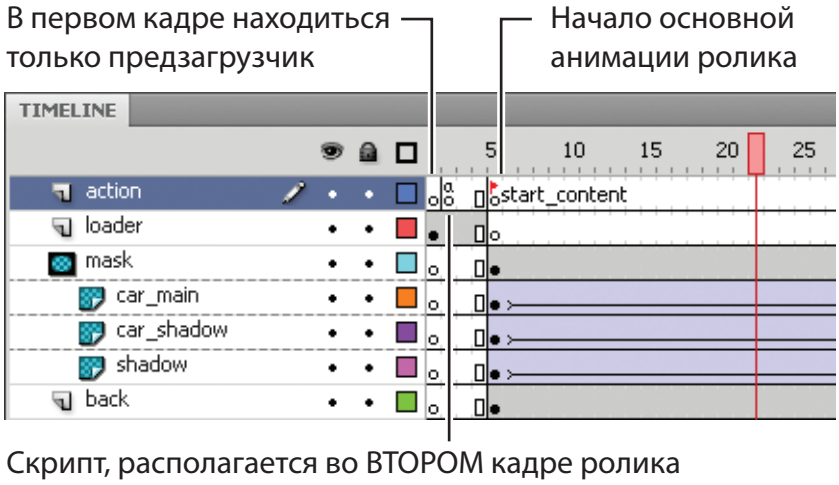
5. В слое «*action*» создайте **ключевой кадр** там, откуда *начинается основная анимация*. Присвойте данному кадру **метку**, например, `start_content`.

6. Во втором кадре ролика в слое «*action*» создайте еще один ключевой кадр и поместите в него следующий код:

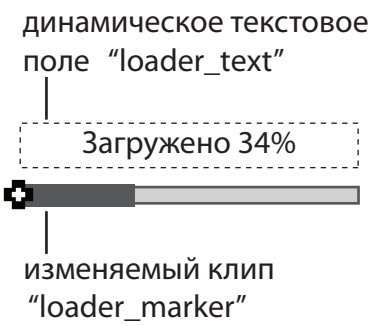
```
// количество кадров в ролике (постоянно):
var total_fr = this._totalframes;

// количество загруженных кадров (меняется):
var loaded_fr = this._framesloaded;

if(loaded_fr < total_fr)    // если ролик недогружен:
     gotoAndPlay (1);                  // возврат на первый кадр
else                        // если ролик скачен полностью:
     gotoAndPlay («start_content»);   // переход к началу анимации
```

7. **В последнем кадре** ролика в слое «*action*» добавить ключевой кадр и поместите в него код, который будет пересылать flash-плеер не на первый кадр ролика, а на кадр начала анимации:
`gotoAndPlay («start_content»);`

Пример временной шкалы ролика с предзагрузчиком приведен ан рисунке:




**Пример 2. Предзагрузчик, отображающий процент загрузки ролика**




В данном случае необходимо выполнить **все** действия, описанные в **Примере 1**.

Клип с предзагрузчиком необходимо **изменить**:

1. Внутрь клипа с предзагрузчиком нужно добавьте *динамическое текстовое поле*, в которое будет выводиться **процент** загрузки ролика. Присвойте данному полю имя экземпляра, например, `loader_text`.

2. Для большей наглядности поместите внутрь клипа другой клип, длина которого будет расти по мере загрузки ролика. Присвойте ему имя экземпляра, например, `loader_marker`

В первый кадр слоя «*action*» нужно добавить код, который будет рассчитывать процент загрузки ролика, выводить его в динамическое поле и масштабировать клип-индикатор:

```
// переменная, определяющая процент загрузки ролика:
```

```
var pct = Math.round(this.getBytesLoaded()/this.getBytesTotal()*100);

// масштабируем клип, показывающий процент загрузки ролика:
loader_pct.loader_marker._xscale = pct;

// выводим процент загрузки в текстовое поле:
loader_pct.loader_text.htmlText = "Загружено "+pct+" %";
```

**Код второго кадра остается без изменений!**

**Важно:**

- Правильно установить **точку регистрации** для клипа индикатора загрузки (`loader_marker`). Так как масштабирование клипа идет именно от этой точки. *В нашем пример точка регистрации находиться слева.*
- Так как мы связываем горизонтальный *масштаб клипа-индикатора* (`_xscale`) и *процент загрузки ролика*, то изначально клип индикатор должен быть нарисован так, чтобы 100% его длины совпадали с индикацией полной загрузки ролика.

**Проверка загрузки конеретного кадра ролика**

Описанные выше примеры, проверяют факт полной загрузки ролика. Иногда встает задача отследить момент **загрузки конкретного кадра ролика**. Для этой цели используют метод `ifFrameLoaded()`. В качестве аргумента передается номер требуемого кадра.

Чтобы закончить работу предзагрузчика сразу, как будет загружен некоторый кадр (например кадр 50) необходимо выполнить **все** действия **Пример 1**.
Но во второй ключевой кадр слоя «*action*» поместить следующий код:

```
// альтернативный способ - проверка загрузки конкретного кадра:
gotoAndPlay(1);
ifFrameLoaded (50){   // как только будет загружен кадр 50
     gotoAndPlay ("start_content");   // перейти к началу анимации
}
```

## Работа со временем и датой

Для работы со временем и датой в ActionScript предназначен класс `Date`. Каждый экземпляр этого класса представляется 64-битным числом с плавающей точкой и указывает на какой-то конкретный момент времени. Дата и время задаются *единым числом*, равным **количеству миллисекунд**, прошедших от нуля часов **1 января 1970** (время определяется по Гринвичу *GMT - Greenwich Mean Time*).

Чтобы создать объект Дата/Время и поместить в него **текущие значения** необходимо просто создать новый экземпляр класса `Date`:

```
// ТЕКУЩЯЯ ДАТА:
var my_data1:Date=new Date();
```

Чтобы задать какой-то **конкретный момент времени**, его нужно указать как параметр при создании нового экземпляра класса `Date`:

```
var date=new Date(year, month, date, hour, minute, second, millisecond)
```

- `year` - год. Отсчет лет на компьютере традиционно ведется с 1970 года. Годы в промежутке с 1900 по 1999 можно указывать 2-мя, а не 4-мя цифрами (81), можно задавать год отрицательным числом: 1899 году соответствует число –1
- `month` - месяц, месяцы номеруются с 0
- `date` - число, отсчет чисел ведется с 1
- `hour` - час, принимает значения от 0 до 23
- `minute` - минута, может изменяться от 0 до 59
- `second` - секунда, отсчет секунд ведется с 0
- `millisecond` - миллисекунда, может изменяться от 0 до 999

*Например:*

```
// ЗАДАНИЕ ДАТЫ:
var my_data2:Date=new Date(2003,2,5);         // год/месяц/число
trace(my_data2);    //Wed Mar 5 00:00:00 GMT+0300 2003

var my_data3:Date=new Date(-1046833200000); // число миллисекунд
trace(my_data3);    //Fri Oct 30 00:00:00 GMT+0300 1936
```

Чтобы выделить из объекта дата **отдельные составляющие** используют следующие методы:

```
var time = new Date(); // создали экземпляр класса Date

// локальное временя:                  // стандартное временя:
time.getFullYear();                    time.getUTCFullYear();
time.getMonth();                       time.getUTCMonth();
time.getDate();                        time.getUTCDate();
time.getHours();                       time.getUTCHours();
time.getMinutes();                     time.getUTCMinutes();
time.getSeconds();                     time.getUTCSeconds();
time.getMilliseconds();                time.getUTCMilliseconds();
```

*Стандартное временя - время на гринвичском меридиане (GMT), измеренное относительно шкалы UTC.*

Для преобразования даты, хранимой в объекте `Date` в число (`Number`) предназначен метод `getTime()`. Как правило результат присваивают в качестве значения некоторой *переменной*, с которой потом можно выполнять **математические операции**.

*Например:*

```
// начальная дата, преобразованная в число:
var begin_time:Number=(new Date()).getTime();

// дата окончания некоторого процесса, преобразованная в число:
var win_time:Number = (new Date()).getTime();

// математическая операция(разница между датами):
var delta_time = win_time - begin_time;
```

```
// результат преобразуем в новый объект Date:
var time:Date = new Date(delta_time);
```

**Пример. Аналоговые и цифровые часы**


1. Создайте динамическое текстовое поле и присвойте ему имя экземпляра `digital_time`.
2. Нарисуйте циферблат часов и три стрелки. Конвертируйте каждую стрелку в клип и установите точки регистрации так, чтобы они совпадали с центром циферблата.
3. Присвойте экземплярам клипов со стрелками уникальные имена.

*Например*, `arrow_h` – часовая стрелка
`arrow_m` – минутная стрелка
`arrow_s` – секундная стрелка

4. В кадр главной временной шкалы добавьте следующий код:

```
this.onEnterFrame=function() {
      var time = new Date(); // текущая дата-время

      // определение локального времени:
      var hours = time.getHours();
      var minutes = time.getMinutes();
      var seconds = time.getSeconds();

      // АНАЛОГОВЫЕ ЧАСЫ:
      // повотор часовой стрелки:      360 градусов / 12 делений = 30 градусов +
      // + поправка на прошедшие минуты в данном часе
      arrow_h._rotation = (hours*30) + (minutes/2);

      // поворот минутной стрелки:     360 градусов / 60 делений = 6 градусов
      arrow_m._rotation = 6 * minutes;

      // поворот секундной стрелки:    360 градусов / 60 делений = 6 градусов
      arrow_s._rotation = 6 * seconds;

      // ЦИФРОВЫЕ ЧАСЫ:
      // форматирование:
      if (length(hours) == 1)
            hours = "0" + hours;
      if (length(minutes) == 1)
            minutes = "0" + minutes;
      if (length(seconds) == 1)
            seconds = "0" + seconds;

      // вывод текущего времени:
      digital_time.text = hours+":"+minutes+":"+seconds;
}
```

## Вызов функции через определенные интервалы времени

Чтобы пользовательская функция автоматически вызывалась через определенные промежутки времени, в ActionScript используют функцию `setInterval()`, которая имеет следующий синтаксис:

`setInterval(func, time)`, где
- `func` – имя пользовательской функции
- `time` – интервал в миллисекундах, через который необходимо вызывать пользовательскую функцию.

При этом пользовательская функция будет вызывать постоянно вплоть до закрытия ролика. Чтобы иметь возможность **оборвать** вызов функции при выполнении *некоторого условия* в ролике, необходимо при вызове функции присвоить ей **порядковый номер (ID) интервала**:

```
var intervalIdTimer:Number;    // ID интервала
intervalIdTimer = setInterval(showTime, 1000);
```

Чтобы оборвать вызов пользовательской функции необходимо использовать функцию `clearInterval()`, в качестве параметра которой передается порядковый номер (ID) того, интервала который требуется удалить:

```
clearInterval(intervalIdTimer);
```

### Пример. Таймер обратного отсчета

Допустим, что на выполнение некоторого задания в ролике пользователю отведено 90 секунд, если он не успевает выполнить задание, выполняется переход к следующему кадру и появляется сообщение о том, что задание не выполнено.

1. Создайте динамическое текстовое поле, в которое будет выводиться время, отведенное на выполнение задания. Присвойте ему имя экземпляра, например, `t_time`.

2. В ключевой кадр главной временной шкалы поместите следующий код:

```
var total_time = 90;              // отведенное время в секундах
var intervalIdTimer:Number;       // ID интервала

// объявление пользовательской функции обоажения времени:
function showTime(){
      // если время истекло:
      if (total_time == 0){
            // вывести нужный текст
            trace("Вы проиграли...");
            // очистить ID интервала
            clearInterval(intervalIdTimer);
            // перейти к следующему кадру ролика
            nextFrame();
      }
```

```
      // вычисляем время:
      var minut = Math.floor(total_time/60); // выделяем минуты
      var second = total_time % 60;          // выделяем секунды

      // "форматируем" время:
      if (length(minut) == 1){
            minut = "0" + minut;
      }
      if (length(second) == 1){
            second = "0" + second;
      }
      // отображаем время:
      t_time.text = minut+":"+second;

      // уменьшаем оставшееся время:
      total_time --;
}

showTime();      // первичный вызов функции

// повторый вызов функции через определенный интервал - пользовательская
функция showTime() будет вызывать один раз в секунду:

intervalIdTimer = setInterval(showTime, 1000);
```

## Однократный вызов функции с задержкой

Для **однократного** вызова пользовательской функции через определенный промежуток времени используется функция `setTimeout()`, которая имеет следующий синтаксис:

`setTimeout(func, time)`, где
- `func` – имя пользовательской функции
- `time` – интервал в миллисекундах, через который необходимо вызвать пользовательскую функцию.

При этом пользовательская функция будет вызвана **только один раз**.

# Приложение

# Коды символов и клавиш

**Таблица букв, цифр и знаков припенания**

| Символ / Клавиша | Код | ASCII код |
|---|---|---|
| A | 65 | 65 |
| B | 66 | 66 |
| C | 67 | 67 |
| D | 68 | 68 |
| E | 69 | 69 |
| F | 70 | 70 |
| G | 71 | 71 |
| H | 72 | 72 |
| I | 73 | 73 |
| J | 74 | 74 |
| K | 75 | 75 |
| L | 76 | 76 |
| M | 77 | 77 |
| N | 78 | 78 |
| O | 79 | 79 |
| P | 80 | 80 |
| Q | 81 | 81 |
| R | 82 | 82 |
| S | 83 | 83 |
| T | 84 | 84 |
| U | 85 | 85 |
| V | 86 | 86 |
| W | 87 | 87 |
| X | 88 | 88 |
| Y | 89 | 89 |
| Z | 90 | 90 |
| 0 | 48 | 48 |
| 1 | 49 | 49 |
| 2 | 50 | 50 |
| 3 | 51 | 51 |
| 4 | 52 | 52 |
| 5 | 53 | 53 |
| 6 | 54 | 54 |
| 7 | 55 | 55 |
| 8 | 56 | 56 |
| 9 | 57 | 57 |

| Символ / Клавиша | Код | ASCII код |
|---|---|---|
| a | 65 | 97 |
| b | 66 | 98 |
| c | 67 | 99 |
| d | 68 | 100 |
| e | 69 | 101 |
| f | 70 | 102 |
| g | 71 | 103 |
| h | 72 | 104 |
| i | 73 | 105 |
| j | 74 | 106 |
| k | 75 | 107 |
| l | 76 | 108 |
| m | 77 | 109 |
| n | 78 | 110 |
| o | 79 | 111 |
| p | 80 | 112 |
| q | 81 | 113 |
| r | 82 | 114 |
| s | 83 | 115 |
| t | 84 | 116 |
| u | 85 | 117 |
| v | 86 | 118 |
| w | 87 | 119 |
| x | 88 | 120 |
| y | 89 | 121 |
| z | 90 | 122 |
| ; : | 186 | 59 |
| = + | 187 | 61 |
| - _ | 189 | 45 |
| / ? | 191 | 47 |
| ` ~ | 192 | 96 |
| [ { | 219 | 91 |
| \ \| | 220 | 92 |
| ] } | 221 | 93 |
| " ' | 222 | 39 |
| , | 188 | 44 |
| . | 190 | 46 |
| / | 191 | 47 |

**Таблица системных клавиш**

| Имя свойства (Key Property) | Описание (код клавиши) | Код | ASCII код |
|---|---|---|---|
| BACKSPACE | Клавиша "Возврат" - Backspace | 8 | 8 |
| CAPSLOCK | Клавиша "Верхний регистр" - Caps Lock | 20 | 0 |
| CONTROL | Клавиша "Управление" - Control | 17 | 0 |
| DELETEKEY | Клавиша "Удалить" - Delete | 46 | 127 |
| DOWN | Клавиша "Стрелка Вниз"- Down Arrow | 40 | 0 |
| END | Клавиша "В конец"- End | 35 | 0 |
| ENTER | Клавиша "Ввод"- Enter | 13 | 13 |
| ESCAPE | Клавиша "Отмена" - Escape | 27 | 27 |
| HOME | Клавиша "В начало" - Home | 36 | 0 |
| INSERT | Клавиша "Вставка"- Insert | 45 | 0 |
| LEFT | Клавиша "Стрелка Влево" - Left Arrow | 37 | 0 |
| PGDN | Клавиша "Страница Вниз" - Page Down | 34 | 0 |
| PGUP | Клавиша "Страница Вверх" - Page Up | 33 | 0 |
| RIGHT | Клавиша "Стрелка Направо" - Right Arrow | 39 | 0 |
| SHIFT | Клавиша модификатор Shift | 16 | 0 |
| SPACE | Клавиша "Пробел" - Spacebar | 32 | 32 |
| TAB | Клавиша "Табуляция" - Tab | 9 | 9 |
| UP | Клавиша "Стрелка Вверх" - Up Arrow | 38 | 0 |

**Функциональные клавиши**

| Клавиша | Код | ASCII код |
|---|---|---|
| F1 | 112 | 0 |
| F2 | 113 | 0 |
| F3 | 114 | 0 |
| F4 | 115 | 0 |
| F5 | 116 | 0 |
| F6 | 117 | 0 |
| F7 | 118 | 0 |
| F8 | 119 | 0 |
| F9 | 120 | 0 |
| F10 | не используется | |
| F11 | 122 | 0 |
| F12 | 123 | 0 |
| F13 | 124 | 0 |
| F14 | 125 | 0 |
| F15 | 126 | 0 |
| Num Lock | 144 | 0 |
| ScrLk | 145 | 0 |
| Pause/Break | 19 | 0 |

**Дополнительная цифровая клавиатура**

| Клавиша | Код | ASCII код |
|---|---|---|
| Numpad 0 | 96 | 48 |
| Numpad 1 | 97 | 49 |
| Numpad 2 | 98 | 50 |
| Numpad 3 | 99 | 51 |
| Numpad 4 | 100 | 52 |
| Numpad 5 | 101 | 53 |
| Numpad 6 | 102 | 54 |
| Numpad 7 | 103 | 55 |
| Numpad 8 | 104 | 56 |
| Numpad 9 | 105 | 57 |
| Multiply | 106 | 42 |
| Add | 107 | 43 |
| Enter | 13 | 13 |
| Subtract | 109 | 45 |
| Decimal | 110 | 46 |
| Divide | 111 | 47 |