Лекция 4

Знакомство с объектами: написание своих объектов, время жизни, статические переменные.

Видимость объектов и ссылка на null

- Каждый объект после создания существует (живёт) пока хотя бы одна переменная хранит его адрес (на него есть хотя бы одна ссылка). Если ссылок больше не остаётся – объект умирает.

```
public class MainClass
{
  public static void main (String[] args)
  {
    Tommy
    Cat cat = new Cat("Tommy");
    cat = null;

    Sammy
    Cat cat1 = new Cat("Sammy");
      Maisy
      Cat cat2 = new Cat("Maisy");
      cat2 = cat1;

      Ginger
      cat1 = new Cat("Ginger");
       cat2 = null;
  }
}
```

— Объект «кот Томми» существует всего одну строчку с момента создания. Уже на следующей строке единственную переменную, которая хранит на него ссылку, «обнуляют» и объект уничтожается Java-машиной.

— Объект «кот Семми» после создания хранится в переменной cat1. Или, если быть точным, там хранится ссылка на него. Парой строчек ниже эта ссылка копируется в cat2. После этого в cat1 сохраняется ссылка на другой объект, и ссылка на «кот Семми» остаётся только в cat2. Наконец, в последней строке метода main, последнюю ссылку на объект обнуляют.

— Объект «кот Мейси» существует сразу после создания всего одну строчку. На следующей строке переменной cat2 присваивают другое значение, и ссылка на «кот Мейси» теряется. Объект становится недостижимым и считается мусором (объект умер).

— Объект «кот Джинджер» существует сразу после создания и до конца метода. Вместе с окончанием метода будет уничтожена переменная cat2, и следом за ней и объект «кот Джинджер».

finalize()

- Если нужно, чтобы какая-то переменная перестала хранить ссылку на объект — можно присвоить ей значение null или же ссылку на другой объект.

- **finalize**() вызывается Java-машиной у объекта перед тем, как объект будет уничтожен. Фактически этот метод — противоположность конструктору. В нем можно освобождать ресурсы, используемые объектом.

- Этот метод есть у класса Object и, следовательно, есть в каждом классе (все классы в Java считаются унаследованными от класса Object и содержат копию его методов). Ты можешь просто написать в твоём классе такой же метод, и он будет вызываться перед уничтожением объектов этого класса.

```
class Cat
{
  String name;

  Cat(String name)
  {
    this.name = name;
  }

  protected void finalize() throws Throwable
  {
    System.out.println(name + " destroyed");
  }
}
```

- Но! Java-машина сама решает — вызвать данный метод или нет. Чаще всего объекты, созданные в методе и объявленные мусором после его завершения, уничтожаются сразу же и без всяких вызовов метода **finalize**(). Этот метод скорее дополнительная страховка, чем надёжное решение. Лучшим вариантом будет освобождать любые используемые ресурсы (обнулять сохраненные ссылки на другие объекты), когда объект ещё жив. Важно знать две вещи: такой метод существует, и не всегда вызывается.

Время жизни объекта

- В Java случайно потерять объект очень сложно — если у вас есть ссылка на него, значит, объект гарантированно жив.

- Ссылки на объекты нельзя поменять. Нельзя увеличить или уменьшить. Также нельзя создать ссылку на объект — ее можно только присвоить. Или обнулить.

- Если обнулить (стереть) все ссылки на объект, то не получится получить на него ссылку и обращаться к нему.

- Часто наблюдается обратная ситуация — слишком много неиспользуемых живых объектов. Часто программисты создают объекты десятками и хранят их в различных списках для обработки, но никогда эти списки не чистят.

- Чаще всего ненужные объекты помечаются программистами, как неиспользуемые и все. А удалением их из списков никто не занимается. Так что для больших Java-программ характерно раздувание — все больше и больше неиспользуемых объектов остаются жить в памяти.

Статические методы и классы

- Когда мы описываем переменные в классе, мы указываем, будут ли эти переменные созданы всего один раз или же нужно создавать их копии для каждого объекта. По умолчанию создаётся новая копия переменной для каждого объекта.

- Статические переменные — существуют в одном экземпляре, и обращаться к ним нужно по имени класса (внутри класса к статической переменной можно обращаться просто по имени):

Метод по умолчанию	Статический метод
● Объявление класса	

class Cat //класс
{
String name; //переменная
 	
Cat(String name) //конструктор
{
this.name = name;
//инициализация переменной
}
}
	class Cat //класс
	{
	String name;
	//обычная переменная
	static int catCount;
	//статическая переменная
	
	Cat(String name)
	{
	this.name = name;
	Cat.catCount++; //увеличиваем значение статический переменной на 1
	}
	}

● Код в методе main	

Cat cat1 = **new** Cat("Vaska"); //создали один объект, его name содержит строку «Vaska»
Cat cat2 = **new** Cat("Murka"); //создали один объект, его name содержит строку «Murka»
System.out.println(cat1.name);
System.out.println(cat2.name);
	System.out.println(Cat.catCount);
	Cat cat1 = **new** Cat("Vaska");
	
	System.out.println(Cat.catCount);
	Cat cat2 = **new** Cat("Murka");
	
	System.out.println(cat1.name);
	System.out.println(cat2.name);
	System.out.println(Cat.catCount);

● Вывод на экран	
Vaska Murka	0 1 Vaska Murka 2

- Методы класса тоже делятся на две категории. **Обычные методы** вызываются у объекта и имеют доступ к данным этого объекта. **Статические методы** не имеют такого доступа — у них просто нет ссылки на объект, они способны обращаться либо к статическим переменным этого класса либо к другим статическим методам.

- Статические методы не могут обращаться к нестатическим методам или нестатическим переменным!

- Каждая обычная переменная класса находится внутри объекта. Обратиться к ней можно только имея ссылку на этот объект. В статический метод такая ссылка не передается.

- В обычные методы передается, неявно. В каждый метод неявно передается ссылка на объект, у которого этот метод вызывают. Переменная, которая хранит эту ссылку, называется **this**. Таким образом, метод всегда может получить данные из своего объекта или вызвать другой нестатический метод этого же объекта.

- В статический метод вместо ссылки на объект передается null. Поэтому он не может обращаться к нестатическим переменным и методам — у него банально нет ссылки на объект, к которому они привязаны.

Нестатические методы

Как выглядит код	Что происходит на самом деле

Cat cat = **new** Cat();
String name = cat.getName();
cat.setAge(17);
cat.setChildren(cat1, cat2, cat3);
	Cat cat = **new** Cat();
	String name = Cat.getName(cat);
	Cat.setAge(cat, 17);
	Cat.setChildren(cat, cat1, cat2, cat3);

При вызове метода в виде «объект» точка «имя метода», на самом деле вызывается метод класса, в который первым аргументом передаётся тот самый объект. Внутри метода он получает имя this. Именно с ним и его данными происходят все действия.	

Статические методы

Как выглядит код	Что происходит на самом деле

`Cat cat1 = new Cat();`
`Cat cat2 = new Cat();`
`int catCount = Cat.getAllCatsCount();`
	`Cat cat1 = new Cat();`
	`Cat cat2 = new Cat();`
	`int catCount = Cat.getAllCatsCount(null);`

При вызове статического метода, никакого объекта внутрь не передаётся. Т.е. this равен null, поэтому статический метод не имеет доступа к нестатическим переменным и методам (ему нечего неявно передать в обычные методы).	

- Переменная или метод являются статическими, если перед ними стоит ключевое слово static.
- У такого подхода тоже есть свои преимущества.
- Во-первых, для того, чтобы обратиться к статическим методам и переменным не надо передавать никакую ссылку на объект.
- Во-вторых, иногда бывает нужно, чтобы переменная была в единственном экземпляре. Как, например, переменная System.out (статическая переменная out класса System).
- И в третьих, иногда нужно вызвать метод, еще до того, как будет возможность создавать какие-то объекты
- Метод main объявлен статическим? Чтобы его можно было вызвать сразу после загрузки класса в память. Еще до того, когда можно будет создавать какие-то объекты.

- Кроме статических методов есть ещё и статические классы. Что это такое рассмотрим в будущем, я лишь покажу пример, что такое может быть:

```
public class StaticClassExample
{
  private static int catCount = 0;

  public static void main(String[] args) throws Exception
  {
    Cat vaska = new Cat("Bella");
    Cat murka = new Cat("Tiger");

    System.out.println("Cat count " + catCount);
  }

  public static class Cat
  {
    private String name;

    public Cat(String name)
    {
      this.name = name;
      StaticClassExample.catCount++;
    }
  }

}
```

- Объектов класса Cat можно создавать сколько угодно. В отличие от, например, статической переменной, которая существует в единственном экземпляре.

- Основной смысл модификатора **static** перед объявлением класса — это регулирование отношения класса **Cat** к классу **StaticClassExample**. Смысл примерно такой: класс Cat не привязан к объектам класса StaticClassExample, и не может обращаться к обычным (нестатическим) переменным класса StaticClassExample.

Задачи

1. В классе **Cat** создать метод protected void finalize() throws Throwable.

```
public class Cat {
   //напишите тут ваш код
   public static void main(String[] args) {


   }
}
```

2. Добавить к классу **Cat** два статических метода: int getCatCount() и setCatCount(int), с помощью которых можно получить/изменить количество котов (переменную catCount). Метод setCatCount должен присваивать переменной catCount переданное значение.

```
public class Cat {
   private static int catCount = 0;

   public Cat() {
      catCount++;
   }

   public static int getCatCount() {
      //напишите тут ваш код
   }

   public static void setCatCount(int catCount) {
      //напишите тут ваш код
   }

   public static void main(String[] args) {

   }
}
```

3. Переставьте один модификатор static, чтобы пример скомпилировался.

```
public class Solution {

   public int A = 5;
   public int B = 2;
   public static int C = A * B;

   public static void main(String[] args) {
      A = 15;
   }
}
```

4. Реализовать статический метод double getDistance(**x1, y1, x2, y2**). Он должен вычислять расстояние между точками. Используй метод double Math.sqrt(double **a**), который вычисляет квадратный корень переданного параметра.

```
public class Util {
   public static double getDistance(int x1, int y1, int x2, int y2) {
      //напишите тут ваш код
   }

   public static void main(String[] args) {

   }
}
```

Массивы и списки: Array, ArrayList, знакомство с Generics.

Массивы

- Массив — это специальный тип данных, который может хранить не одно значение, а несколько.

- Давайте сравним обычный жилой дом и многоэтажку. В обычном доме чаще всего живет одна семья, а многоэтажка разбита на квартиры. Чтобы написать письмо семье, которая живет в обычном доме, надо указать его уникальный адрес. А чтобы написать письмо семье, которая живет в квартире, надо указать уникальный адрес дома и еще номер квартиры.

- Так вот, переменная-массив — это переменная-многоэтажка. В ней можно хранить не одно значение, а несколько. В такой переменной есть несколько квартир (ячеек), к каждой из которых можно обратиться по ее номеру (индексу). Для этого после имени переменной в квадратных скобках надо указать индекс ячейки, к которой обращаемся. Это довольно просто.

- Переменная-многоэтажка (переменная-массив) может быть любого типа, надо лишь вместо «ИмяТипа имяПеременной», написать «ИмяТипа[] имяПеременной».

	Код	Описание
1	`String[] list = new String[5];`	Создание массива на 5 элементов типа «строка»

2	`System.out.println(list[0]);`
	`System.out.println(list[1]);`
	`System.out.println(list[2]);`
	`System.out.println(list[3]);`
	`System.out.println(list[4]);`
		На экран будет выведено пять значений “**null**”.
		
		Чтобы получить значение, которое хранится в определенной ячейке массива, используйте квадратные скобки и номер ячейки

3	`int listCount = list.length;`	listCount получит значение 5 – количество ячеек в массиве list. list.length хранит длину(количество ячеек) массива.
4	`list[1] = "Mama";` `String s = list[1];`	При присваивании объектов ячейкам массива нужно указывать индекс/ номер ячейки в квадратных скобках.

5	`for (int i = 0; i < list.length; i++)`
	`{`
	`  System.out.println(list[i]);`
	`}`
		Вывод всех значений массива на экран.

- Переменная-массив требует дополнительной инициализации.

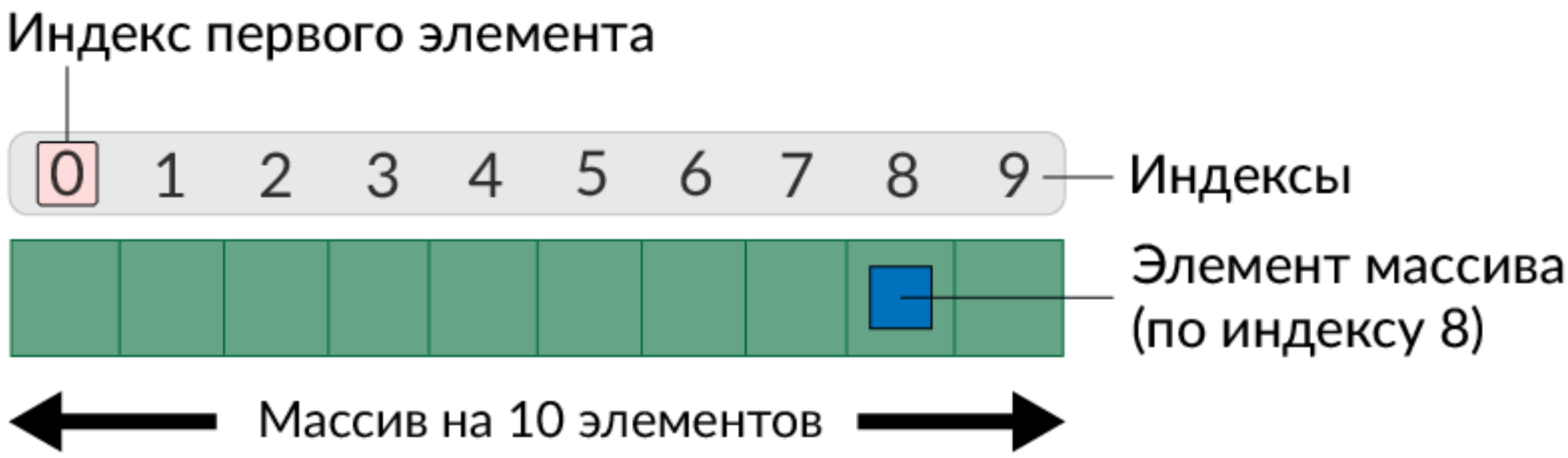
- Обычную переменную нужно было просто объявить и уже можно присваивать ей различные значения. С массивом все немного сложнее.

- Сначала надо создать контейнер на N элементов, а затем в него уже можно класть значения.

	Код	Описание
1	String[] list = **null;**	Переменная-массив list, ее значение – **null**. Она может хранить только контейнер для элементов. Контейнер надо создавать отдельно.
2	String[] list = **new** String[5];	Мы создаем контейнер на 5 элементов и кладем ссылку на него в переменную list. Этот контейнер содержит 5 квартир (ячеек) с номерами 0, 1, 2, 3, 4.
3	String[] list = **new** String[1];	Мы создаем контейнер на 1 элемент и кладем ссылку на него в переменную list. Чтобы занести что-то в этот контейнер надо написать list[0] = “Yo!”;
4	String[] list = **new** String[0];	Мы создаем контейнер на 0 элементов и кладем ссылку на него в переменную list. Ничего в этот контейнер занести нельзя!

Основные факты о массивах:

1) Java массив состоит из множества ячеек.
2) Доступ к конкретной ячейке идёт через указание её номера.
3) Все ячейки одного типа.
4) Начальное значение для всех ячеек – null, для примитивных типов – 0, 0.0 (для дробных), false (для типа boolean). Точно такое же, как и у простых неинициализированных переменных.
5) String[] list — это просто объявление переменной. Сначала нужно создать массив (контейнер) и положить его в эту переменную, а потом уже им пользоваться. См. пример ниже.
6) Когда мы создаём объект массив (контейнер), нужно указать, какой он длины – сколько в нем ячеек. Это делается командой вида: new TypeName[n];

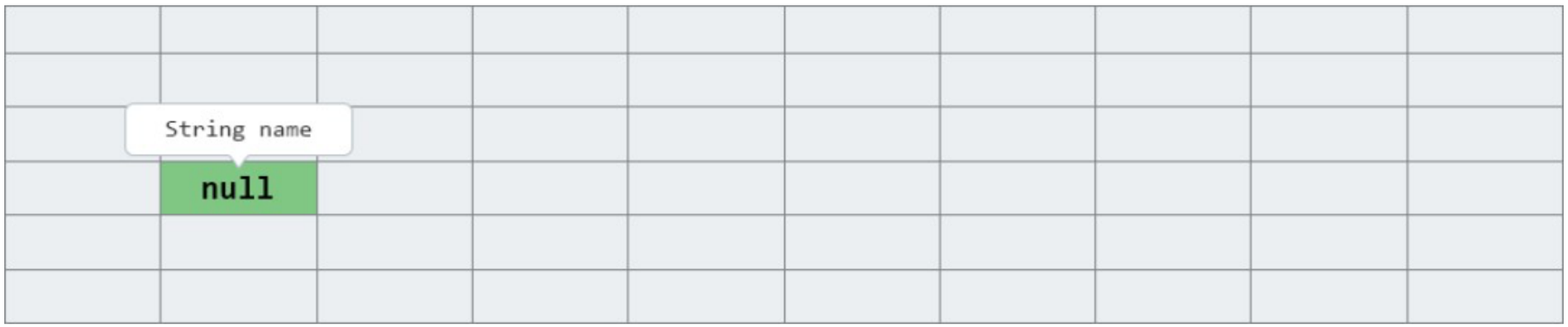


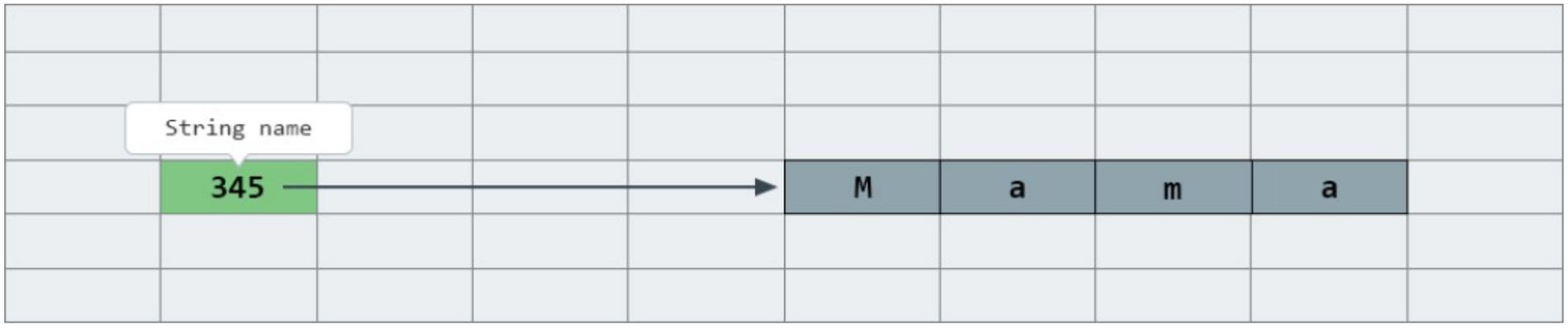
	Код	Описание
1	String s; String[] list;	s равно null list равно null
2	list = **new** String[10]; **int** n = list.length;	Переменная list хранит ссылку на объект — массив строк из 10 элементов. n равно 10
3	list = **new** String[0];	Теперь list содержит массив из 0 элементов. Массив есть, но хранить элементы он не может.
4	list = null; System.out.println(list[1]);	Будет сгенерировано исключение (ошибка программы) – программа аварийно завершится. list содержит пустую ссылку – null
5	list = **new** String[10]; System.out.println(list[11]);	Будет сгенерировано исключение (ошибка программы) – выход за границы массива. Если list содержит 10 элементов/ячеек, то их разрешённые индексы: 0,1,2,3,4,5,6,7,8,9 – всего 10 штук.

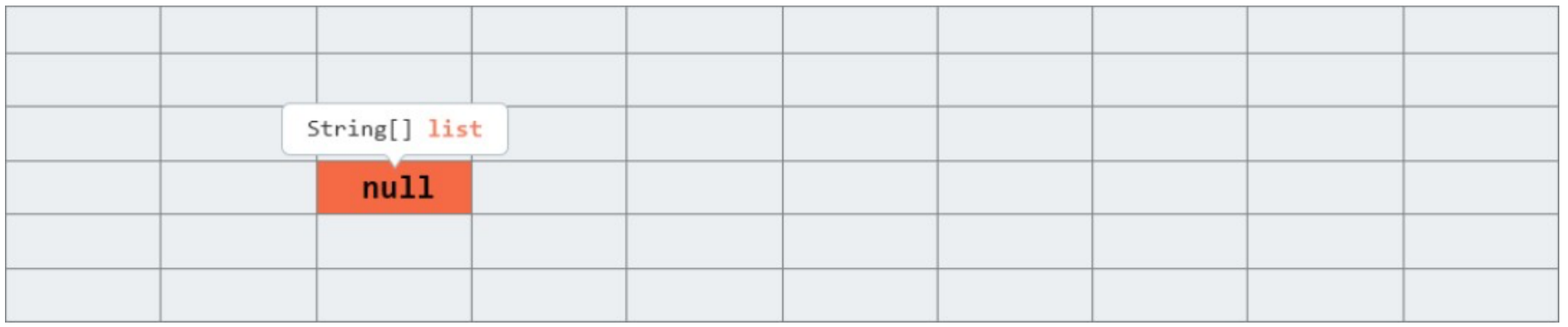
Быстрая (статическая) инициализация. Сумма элементов массива:

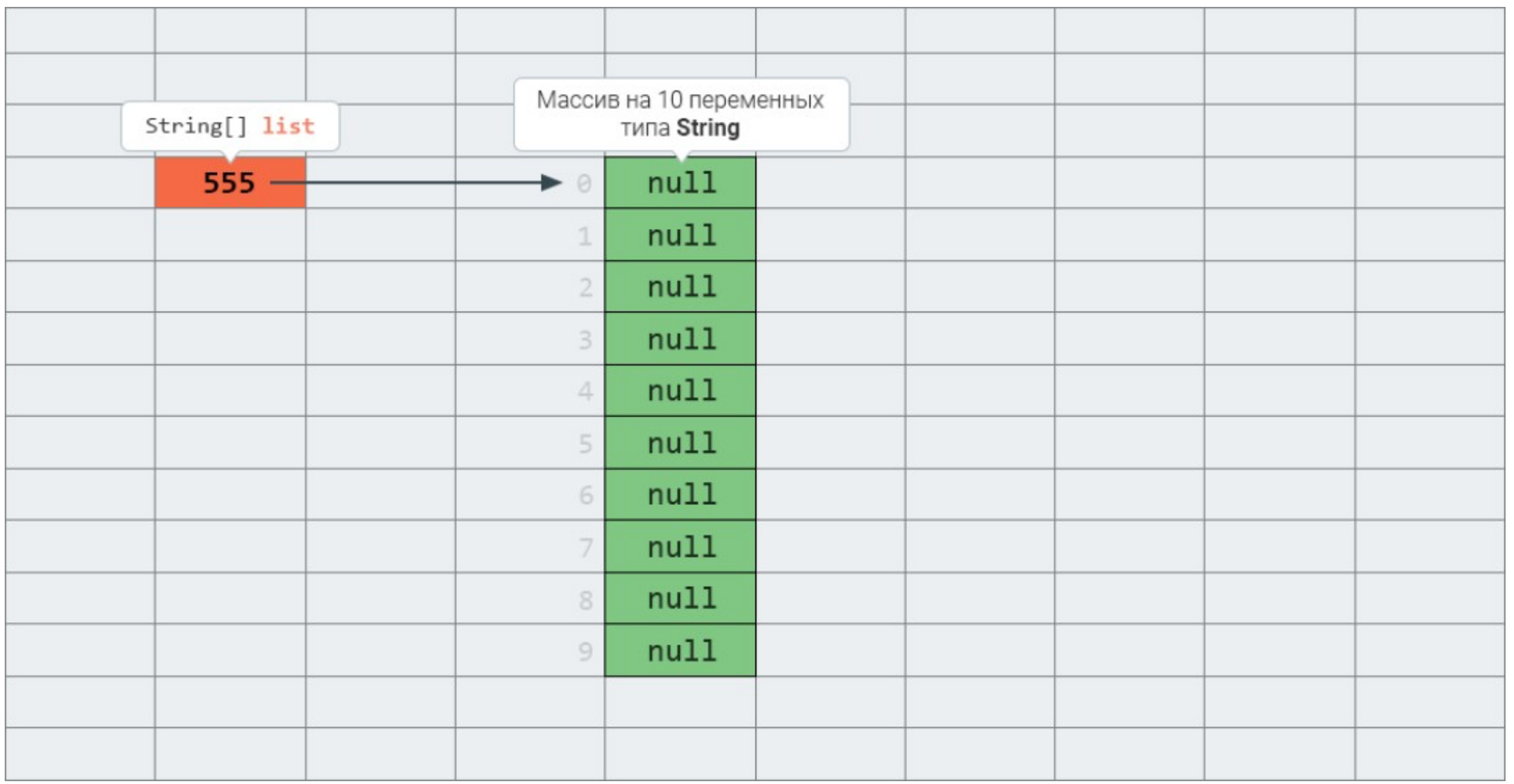
```
public class MainClass
{
    public static void main(String[] args) throws IOException
    {
        //это статическая инициализация
        int[] list = {5, 6, 7, 8, 1, 2, 5, -7, -9, 2, 0};

        //подсчёт суммы элементов
        int sum = 0;
        for (int i = 0; i < list.length; i++)
           sum += list[i];

        System.out.println("Sum is " + sum);
    }
}
```

Массивы в памяти

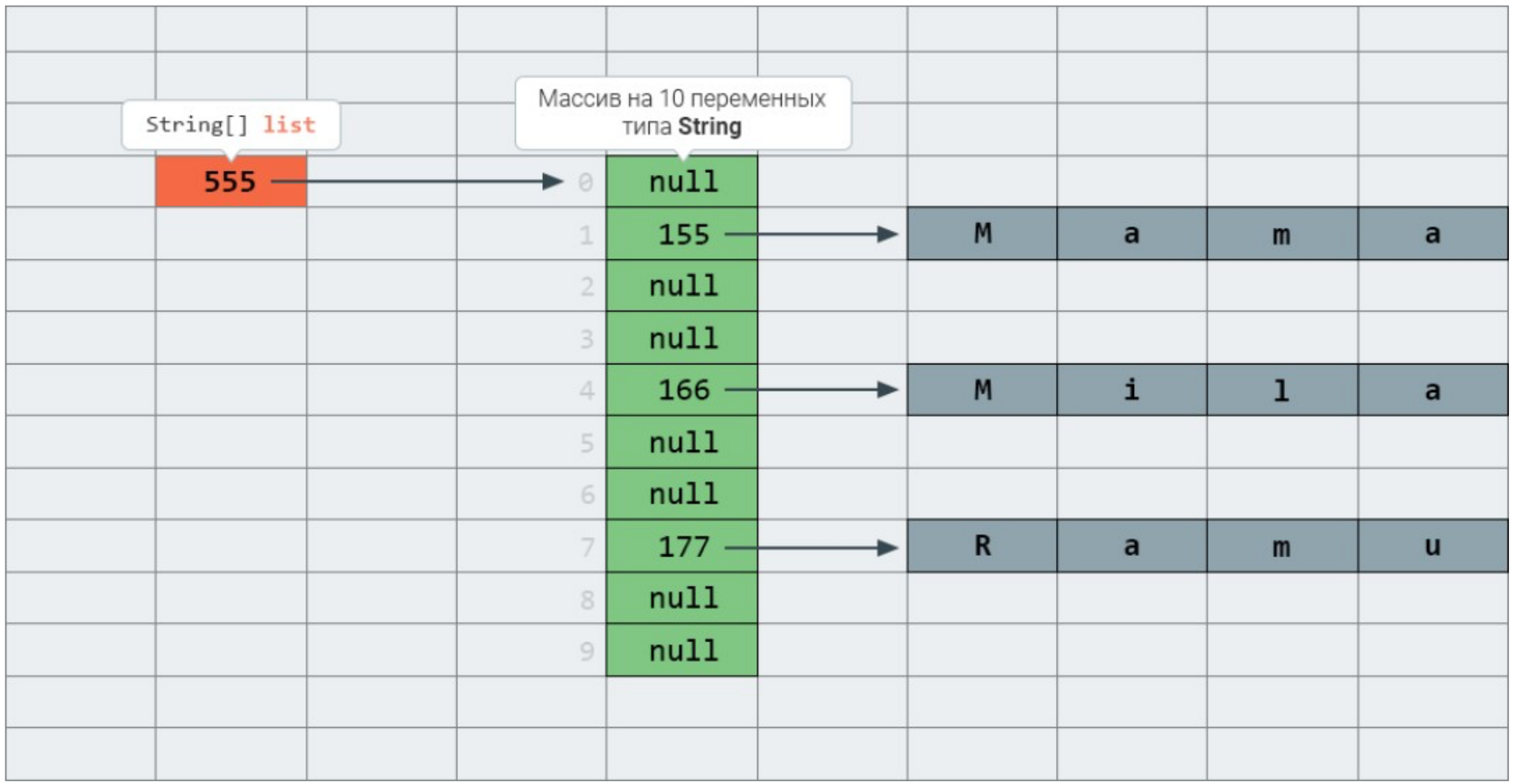
- Объявили переменную типа String
- 

- Присвоили значение
- 

- С массивами все немного сложнее:
- Объявили переменную типа String []
- 

- Создали массив на 10 элементов и присвоили переменной его ссылку
- 

- Присвоили ячейкам массива различные строки



Класс ArrayList

- Небольшая предыстория. Программистам очень не нравилось одно свойство массива – его размер нельзя изменять. Что делать, если нужно сохранить в массиве ещё три элемента, а свободное место только одно? Единственным решением проблемы нехватки места в массиве было создание массива очень большого размера, чтобы все элементы туда точно поместились. Но это часто приводило к нерациональному расходу памяти. Если чаще всего в массиве хранилось три элемента, но был хотя бы мизерный шанс, что там их будет 100, приходилось создавать массив на 100 элементов.

- В итоге, программисты написали класс **ArrayList** (списочный массив), который выполнял ту же работу, что и Array (массив), но мог изменять свой размер.

- Внутри каждого объекта типа **ArrayList** хранится обычный массив элементов. Когда ты считываешь элементы из ArrayList, он считывает их из своего внутреннего массива. Когда записываешь – записывает их во внутренний массив.

Array	ArrayList
Создание контейнера элементов	
`String[] list = new String[10];`	`ArrayList<String> list = new ArrayList<String>();`
Получение количества элементов	
`int n = list.length;`	`int n = list.size();`
Взятие элемента из массива/коллекции	
`String s = list[3];`	`String s = list.get(3);`
Запись элемента в массив	
`list[3] = s;`	`list.set(3, s);`

- **ArrayList** поддерживает несколько дополнительных действий, которые очень часто приходится делать программистам во время работы, и которых нет у массива. Например – вставка и удаление элементов из середины массива, и чтобы не оставалось дырок.

- изменение размера: когда нужно записать во внутренний массив ещё один элемент, а свободного места там нет, то внутри **ArrayList** делается вот что:
а) создаётся ещё один массив, в полтора раза больше размера внутреннего массива, плюс один элемент.
б) все элементы из старого массива копируются в новый массив.
в) новый массив сохраняется во внутренней переменной объекта ArrayList, старый массив объявляется мусором (мы просто перестаём хранить на него ссылку).

Array	ArrayList
Добавление элемента в конец массива	
Невозможно выполнить данное действие	list**.add**(s);
Вставка элемента в середину массива	
Невозможно выполнить данное действие	list**.add**(15, s);
Вставка элемента в начало массива	
Невозможно выполнить данное действие	list**.add**(0, s);
Удаление элемента из массива	
Можно стереть элемент с помощью list[3] = null. Но тогда останется «дыра» в массиве.	list**.remove**(3);

- Сравним работу с ArrayList с работой с массивом. Для примера решим такую задачу «ввести 10 строк с клавиатуры и вывести их на экран в обратном порядке» .

Используем Array	Используем ArrayList

public static void main(String[] args)
{
Reader r = **new** InputStreamReader(system.in);
BufferedReader reader = **new** BufferedReader(r);
 	
//ввод строк с клавиатуры
String[] list = **new** String[10];
for (**int** i = 0; i < list.length; i++)
{
String s = reader.**readLine**();
list[i] = s;
}
 	
//вывод содержимого массива на экран
for (**int** i = list.length - 1; i < 0; i--)
{
System.out.println(list[i]);
}
}
	public static void main(String[] args)
	{
	Reader r = **new** InputStreamReader(system.in);
	BufferedReader reader = **new** BufferedReader(r);
	
	//ввод строк с клавиатуры
	ArrayList<String> list = **new** ArrayList<String>();
	for (**int** i = 0; i < 10; i++)
	{
	String s = reader.**readLine**();
	list.add(s);
	}
	
	//вывод содержимого коллекции на экран
	for (**int** i = list.size() - 1; i < 0; i--)
	{
	System.out.println(list.get(i));
	}
	}

Дженерики (Generics)

- “Дженерики” — это типы с параметром. В Java классы-контейнеры позволяют указывать тип их внутренних объектов.

- Когда объявляется generic-переменную, то указывается не один тип, а два: тип переменной и тип данных, которые она у себя хранит.

- Хороший пример этого — ArrayList. Когда создается новый объект/переменную типа ArrayList, удобно указать, значения какого типа будут храниться внутри этого списка.

- про любой тип. Если в одном методе в ArrayList кладутся строки, а в другом мы работаем с его содержимым и ожидаем, что там будут только числа, программа упадет (закроется с ошибкой).

	Код	Пояснение
1	ArrayList<String> list = **new** ArrayList<String>();	Мы создали переменную list типа ArrayList. Занесли в нее объект типа ArrayList. В таком списке можно хранить только переменные типа String.
2	ArrayList list = **new** ArrayList();	Мы создали переменную list типа ArrayList. Занесли в нее объект типа ArrayList.В таком листе можно хранить переменные любого типа.
3	ArrayList<Integer> list = **new** ArrayList<Integer>();	Мы создали переменную list типа ArrayList. Занесли в нее объект типа ArrayList. В таком листе можно хранить только переменные типа Integer и int.

- В качестве типа-параметра можно поставить почти любой класс, даже тот, что напишете вы. Т.е. любой тип, кроме примитивных типов. Все классы-параметры должны быть унаследованы от класса Object. Нельзя писать ArrayList<int>. Для примитивных типов разработчики языка Java написали их непримитивные аналоги — классы, унаследованные от **Object**.

	Примитивный тип	Класс	Список
1	int	Integer	ArrayList<Integer>
2	double	Double	ArrayList<Double>
3	boolean	Boolean	ArrayList<Boolean>
4	char	Character	ArrayList<Character>
5	byte	Byte	ArrayList<Byte>

- Слияние списков.

```
public static void main(String[] args) throws IOException
{
   ArrayList<Integer> list1 = new ArrayList<Integer>();   //создание списка
   Collections.addAll(list1, 1, 5, 6, 11, 3, 15, 7, 8);   //заполнение списка

   ArrayList<Integer> list2 = new ArrayList<Integer>();
   Collections.addAll(list2, 1, 8, 6, 21, 53, 5, 67, 18);

   ArrayList<Integer> result = new ArrayList<Integer>();

   result.addAll(list1);   //добавление всех значений из одного списка в другой
   result.addAll(list2);

   for (Integer x : result)   //быстрый for по всем элементам, только для коллекций
   {
      System.out.println(x);
   }
}
```

Задачи

5. 1. Введите с клавиатуры **20 чисел**, сохраните их в список и рассортируйте по трём другим спискам:
   Число нацело делится на **3** (x%3==0), нацело делится на **2** (x%2==0) и все остальные.
   Числа, которые делятся на **3** и на **2** одновременно, например **6**, попадают в оба списка.
   Порядок объявления списков очень важен.
   2. Метод printList должен выводить на экран все элементы списка с новой строки.
   3. Используя метод printList выведите эти три списка на экран.
   Сначала тот, который для x%3, потом тот, который для x%2, потом последний.

```
public class Solution {
   public static void main(String[] args) throws Exception {
      //напишите тут ваш код
   }

   public static void printList(ArrayList<Integer> list) {
      //напишите тут ваш код
   }
}
```

6. Переставьте один модификатор static, чтобы пример скомпилировался.

```
public class Solution {
   public final int A = 5;
   public final static int B = 2;
   public final static int C = A * B;

   public static void main(String[] args) {
   }

   public static int getValue() {
      return C;
   }
}
```

7. Сделать класс **StringHelper**, у которого будут **2** статических метода:
   String multiply(String **s**, int **count**) - возвращает строку повторенную count раз.
   String multiply(String **s**) - возвращает строку повторенную **5** раз. Пример: Сириус -> СириусСириусСириусСириусСириус

```
public class StringHelper {
   public static String multiply(String s) {
      String result = "";
      //напишите тут ваш код
      return result;
   }

   public static String multiply(String s, int count) {
      String result = "";
      //напишите тут ваш код
      return result;
   }

   public static void main(String[] args) {

   }
}
```

8. Задача: Написать программу, которая вводит с клавиатуры **5** чисел и выводит их в возрастающем порядке.
   Пример ввода: 3 2 15 6 17
   Пример вывода: 2 3 6 15 17

```
import java.io.BufferedReader;
    import java.io.InputStreamReader;

/*
Числа по возрастанию
*/

public class Solution {
   public static void main(String[] args) throws Exception {
      BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

      //напишите тут ваш код
   }
}
```