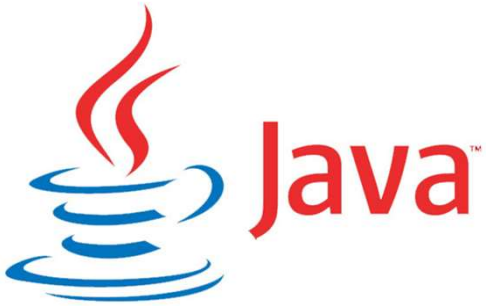
# Тема 10.
# Обработка строк



# Обработка строк

Реализация строк на Java представлена тремя основными классами:

- String
- StringBuffer
- StringBuilder.

# Обработка строк

Строка — объект, что представляет последовательность символов.

Для создания и манипулирования строками Java платформа предоставляет общедоступный финальный (не может иметь подклассов) класс java.lang.String.

Данный класс является неизменяемым — созданный объект класса String не

может быть изменен.

Можно подумать что методы имеют право изменять этот объект, но это неверно.

# Методы класса String

- **concat()** объединяет строки
- **valueOf()** преобразует объект в строковый вид
- **join()** соединяет строки с учетом разделителя
- **compare()** сравнивает две строки
- **charAt()** возвращает символ строки по индексу
- **getChars()** возвращает группу символов
- **equals()** сравнивает строки с учетом регистра

# Методы класса String

- **equalsIgnoreCase()** сравнивает строки без учета регистра
- **regionMatches()** сравнивает подстроки в строках
- **indexOf()** находит индекс первого вхождения подстроки в строку
- **lastIndexOf()** находит индекс последнего вхождения подстроки в строку
- **startsWith()** определяет, начинается ли строка с подстроки
- **endsWith()** определяет, заканчивается ли строка на определенную подстроку

# Методы класса String

- **replace()** заменяет в строке одну подстроку на другую
- **trim()** удаляет начальные и конечные пробелы
- **substring()** возвращает подстроку, начиная с определенного индекса до конца или до определенного индекса
- **toLowerCase()** переводит все символы строки в нижний регистр
- **toUpperCase()** переводит все символы строки в верхний регистр

# Класс String

Объекты String являются **неизменяемыми**, поэтому все операции, которые изменяют строки, фактически приводят к созданию нового объекта. Вопрос, а зачем?

Безопасность и String pool основные причины неизменяемости String в Java.

Безопасность объекта неизменяемого класса String обусловлена такими фактами:

- вы можете передавать строку между потоками и не беспокоиться что она будет изменена
- нет проблем с синхронизацией (не нужно синхронизировать операции со String)
- в Java строки используются для передачи параметров для авторизации, открытия файлов и т.д. - неизменяемость позволяет избежать проблем с доступом
- возможность кэшировать hash code. Об этом позже :-)

String pool позволяет экономить память и не создавать новые объекты для каждой повторяющийся строки. В случае с изменяемыми строками - изменение одной приводило бы к изменению всех строк одинакового содержания.

# Классы StringBuilder и StringBuffer

Объекты String являются **неизменяемыми**, поэтому все операции, которые изменяют строки, фактически приводят к созданию новой строки, что сказывается на производительности приложения.

Для решения этой проблемы, чтобы работа со строками проходила с меньшими издержками в Java были добавлены классы **StringBuffer и StringBuilder**.

По сути они напоминает расширяемую строку, которую можно изменять без ущерба для производительности.

# Потоки и процессы

Процесс — это совокупность кода и данных, разделяющих общее виртуальное адресное пространство. Чаще всего одна программа состоит из одного процесса.

Процессы изолированы друг от друга, поэтому прямой доступ к памяти чужого процесса невозможен (взаимодействие между процессами осуществляется с помощью специальных средств).

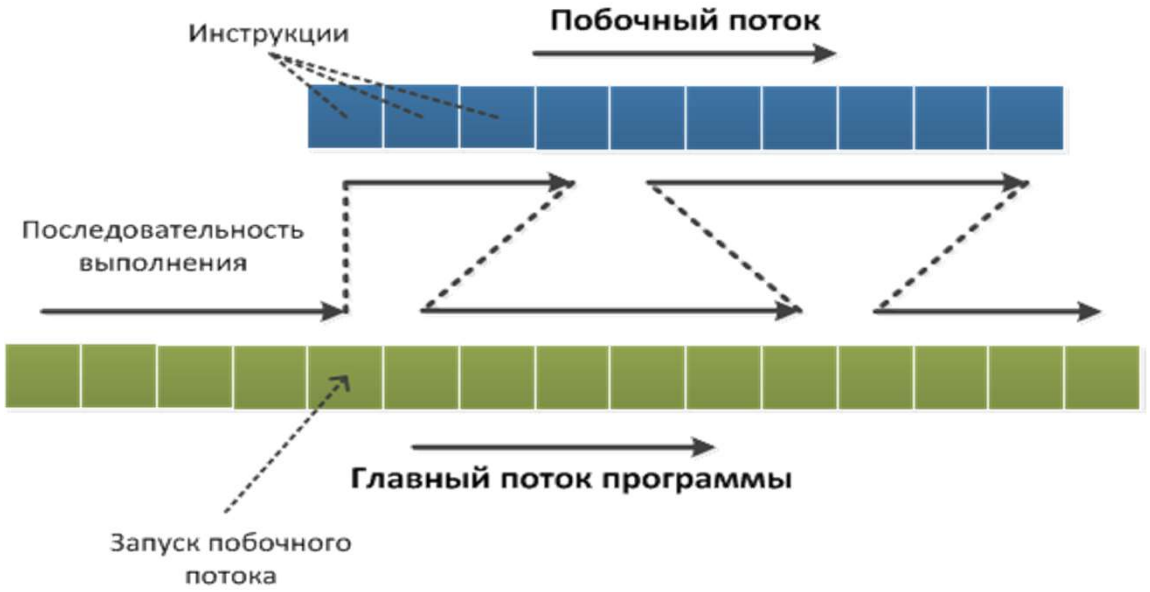
Для каждого процесса ОС создает так называемое «виртуальное адресное пространство», к которому процесс имеет прямой доступ. Это пространство принадлежит процессу, содержит только его данные и находится в полном его распоряжении.

При запуске программы операционная система создает процесс, загружая в его адресное пространство код и данные программы, а затем запускает главный поток созданного процесса.

Один поток – это одна единица исполнения кода. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса.

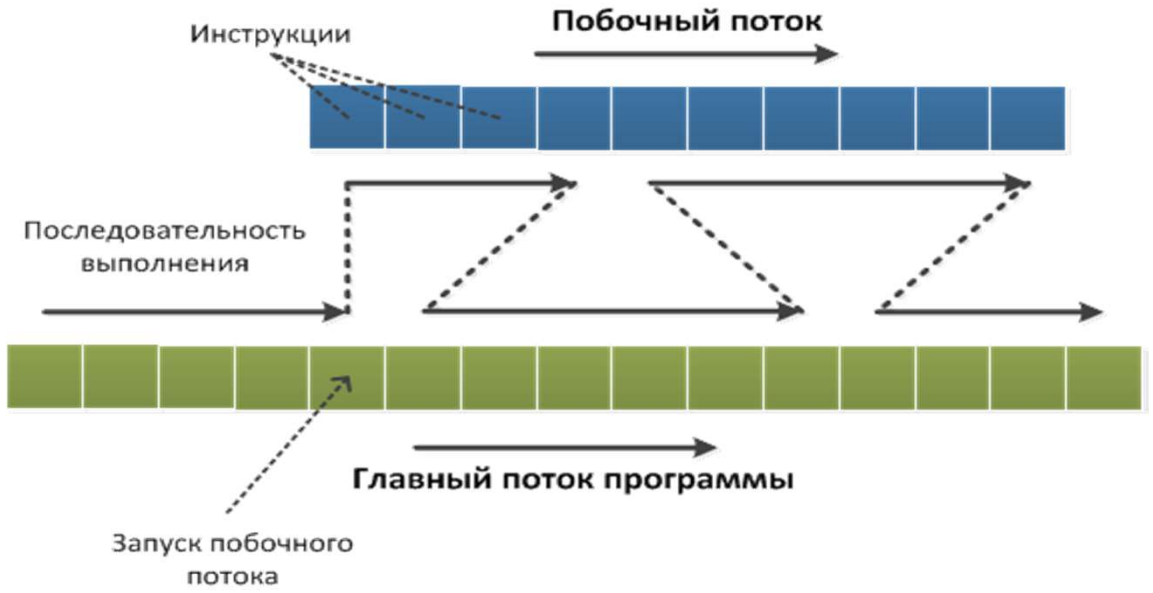
# Потоки и процессы

Многопоточность незаменима тогда, когда необходимо, чтобы графический интерфейс продолжал отзываться на действия пользователя во время выполнения некоторой обработки информации. Например, поток, отвечающий за интерфейс, может ждать завершения другого потока, загружающего файл из интернета, и в это время выводить некоторую анимацию или обновлять прогресс-бар. Кроме того он может остановить поток загружающий файл, если была нажата кнопка «отмена».



# Потоки и процессы

Каждый процесс имеет хотя бы один выполняющийся поток. Тот поток, с которого начинается выполнение программы, называется главным. В языке Java, после создания процесса, выполнение главного потока начинается с метода **main()**. Затем, по мере необходимости, в заданных программистом местах, и при выполнении заданных им же условий, запускаются другие, побочные потоки.



# Потоки и процессы

Многопоточность незаменима тогда, когда необходимо, чтобы графический интерфейс продолжал отзываться на действия пользователя во время выполнения некоторой обработки информации.

Например, поток, отвечающий за интерфейс, может ждать завершения другого потока, загружающего файл из интернета, и в это время выводить некоторую анимацию или обновлять прогресс-бар. Кроме того он может остановить поток загружающий файл, если была нажата кнопка «отмена».

# Синхронизация

Что же такое «синхронизация»?

Вне области программирования под этим подразумевается некая настройка, позволяющая двум устройствам или программам работать совместно. Например, смартфон и компьютер можно синхронизировать с Google-аккаунтом, личный кабинет на сайте — с аккаунтами в социальных сетях, чтобы логиниться с их помощью.

У синхронизации потоков похожий смысл: это настройка взаимодействия потоков между собой.

Представим, что несколько потоков записывают текст в одно и то же место — например, в текстовый файл или консоль.

Этот файл или консоль в данном случае становится общим ресурсом. Потоки не знают о существовании друг друга, поэтому просто записывают все, что успеют за то время, которое планировщик потоков им выделит.

# Синхронизация

**Код 1-го потока**

```
1  System.out.print ("Коле");
2  System.out.print ("");
3  System.out.print ("15");
4  System.out.print ("");
5  System.out.print ("лет");
6  System.out.println ();
```

**Код 2-го потока**

```
1  System.out.print ("Лене");
2  System.out.print ("");
3  System.out.print ("21");
4  System.out.print ("");
5  System.out.print ("год");
6  System.out.println ();
```

**Ожидаемый вывод на консоль**

```
Коле 15 лет
Лене 21 год
```

# Синхронизация

**Итоговый порядок выполнения**

```
System.out.print ("Коле");
System.out.print ("Лене");
System.out.print (" ");
System.out.print (" ");
System.out.print ("15");
System.out.print ("21");
System.out.print (" ");
System.out.print (" ");
System.out.print ("лет");
System.out.println ();
System.out.print ("год");
System.out.println ();
```

**Код 1-го потока**

```
System.out.print ("Коле");
//исполняется другая нить
//исполняется другая нить
System.out.print (" ");
System.out.print ("15");
//исполняется другая нить
//исполняется другая нить
System.out.print (" ");
System.out.print ("лет");
System.out.println ();
//исполняется другая нить
//исполняется другая нить
```

**Код 2-го потока**

```
//исполняется другая нить
System.out.print ("Лене");
System.out.print (" ");
//исполняется другая нить
//исполняется другая нить
System.out.print ("21");
System.out.print (" ");
//исполняется другая нить
//исполняется другая нить
//исполняется другая нить
System.out.print ("год");
System.out.println ();
```

**Реальный вывод на консоль**

```
Коле Лене  15 21 лет
год
```

# Синхронизация

Причина кроется в том, что потоки работали с общим ресурсом, консолью, не согласовывая действия друг с другом. Если планировщик потоков выделил время Потоку-1, тот моментально пишет все в консоль. Что там уже успели или не успели написать другие потоки — неважно.

Поэтому в многопоточном программировании ввели специальное понятие мьютекс (от англ. «mutex», «mutual exclusion» — «взаимное исключение»).

Задача мьютекса — обеспечить такой механизм, чтобы доступ к объекту в определенное время был только у одного потока. Если Поток-1 захватил мьютекс объекта А, остальные потоки не получат к нему доступ, чтобы что-то в нем менять. До тех пор, пока мьютекс объекта А не освободится, остальные потоки будут вынуждены ждать.

Пример из жизни: представь, что ты и еще 10 незнакомых людей участвуете в тренинге. Вам нужно поочередно высказывать идеи и что-то обсуждать. Но, поскольку друг друга вы видите впервые, чтобы постоянно не перебивать друг друга и не скатываться в гвалт, вы используете правило с «говорящим мячиком»: говорить может только один человек — тот, у кого в руках мячик.

# Классы StringBuilder и StringBuffer

Эти классы похожи, практически двойники, они имеют одинаковые конструкторы, одни и те же методы, которые одинаково используются.

Единственное их различие состоит в том, что **класс StringBuffer синхронизированный и потокобезопасный**.

То есть класс StringBuffer удобнее использовать в многопоточных приложениях, где объект данного класса может меняться в различных потоках.

Если же речь о многопоточных приложениях не идет, то лучше использовать класс StringBuilder, который не потокобезопасный, но при этом работает быстрее, чем StringBuffer в однопоточных приложениях.

# Классы StringBuilder и StringBuffer

**Получение и установка символов**

Метод **charAt() получает, а метод setCharAt() устанавливает символ по определенному индексу:**

```
1  StringBuffer strBuffer = new StringBuffer("Java");
2  char c = strBuffer.charAt(0); // J
3  System.out.println(c);
4  strBuffer.setCharAt(0, 'c');
5  System.out.println(strBuffer.toString()); // cava
```

Метод **getChars()** получает набор символов между определенными индексами:

```
1  StringBuffer strBuffer = new StringBuffer("world");
2  int startIndex = 1;
3  int endIndex = 4;
4  char[] buffer = new char[endIndex-startIndex];
5  strBuffer.getChars(startIndex, endIndex, buffer, 0);
6  System.out.println(buffer); // orl
```

# Классы StringBuilder и StringBuffer

**Добавление в строку**

Метод **append()** добавляет подстроку в конец StringBuffer:

```
1 StringBuffer strBuffer = new StringBuffer("hello");
2 strBuffer.append(" world");
3 System.out.println(strBuffer.toString()); // hello world
```

Метод **insert()** добавляет строку или символ по определенному индексу в StringBuffer:

```
1 StringBuffer strBuffer = new StringBuffer("word");
2
3 strBuffer.insert(3, 'l');
4 System.out.println(strBuffer.toString()); //world
5
6 strBuffer.insert(0, "s");
7 System.out.println(strBuffer.toString()); //sworld
```

# Классы StringBuilder и StringBuffer

### Удаление символов

Метод **delete()** удаляет все символы с определенного индекса о определенной позиции, а метод **deleteCharAt()** удаляет один символ по определенному индексу:

```
1  StringBuffer strBuffer = new StringBuffer("assembler");
2  strBuffer.delete(0,2);
3  System.out.println(strBuffer.toString()); //sembler
4
5  strBuffer.deleteCharAt(6);
6  System.out.println(strBuffer.toString()); //semble
```

### Обрезка строки

Метод **substring()** обрезает строку с определенного индекса до конца, либо до определенного индекса:

```
1  StringBuffer strBuffer = new StringBuffer("hello java!");
2  String str1 = strBuffer.substring(6); // обрезка строки с 6 символа до конца
3  System.out.println(str1); //java!
4
5  String str2 = strBuffer.substring(3, 9); // обрезка строки с 3 по 9 символ
6  System.out.println(str2); //lo jav
```

# Классы StringBuilder и StringBuffer

### Замена в строке

Для замены подстроки между определенными позициями в StringBuffer на другую подстроку применяется метод **replace()**:

```
1 StringBuffer strBuffer = new StringBuffer("hello world!");
2 strBuffer.replace(6,11,"java");
3 System.out.println(strBuffer.toString()); //hello java!
```

Первый параметр метода replace указывает, с какой позиции надо начать замену, второй параметр - до какой позиции, а третий параметр указывает на подстроку замены.

### Обратный порядок в строке

Метод **reverse()** меняет порядок в StringBuffer на обратный:

```
1 StringBuffer strBuffer = new StringBuffer("assembler");
2 strBuffer.reverse();
3 System.out.println(strBuffer.toString()); //relbmessa
```

# Практика

Ввести n строк с консоли, найти самую короткую и самую длинную строки. Вывести найденные строки и их длину.

- для вывода результат используйте StringBuilder

# Практика

Ввести n строк с консоли. Вывести на консоль те строки, длина которых больше средней, а также длину.

- для вывода результат используйте StringBuilder

# Практика

Считайте с клавиатуры три строки. А затем:

1. Выведите на экран третью строку в неизменном виде.

2. Выведите на экран вторую строку, предварительно преобразовав ее к верхнему регистру.

3. Выведите на экран первую строку, предварительно преобразовав ее к нижнему регистру.

- для вывода результат используйте StringBuilder