Java Basics Procedural Decomposition Lesson 05

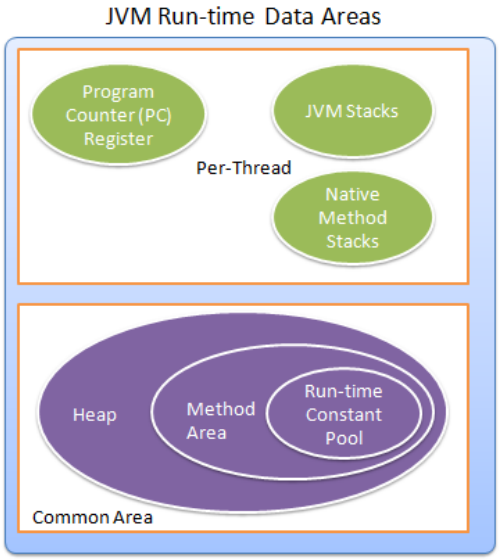
OLGA SMOLYAKOVA

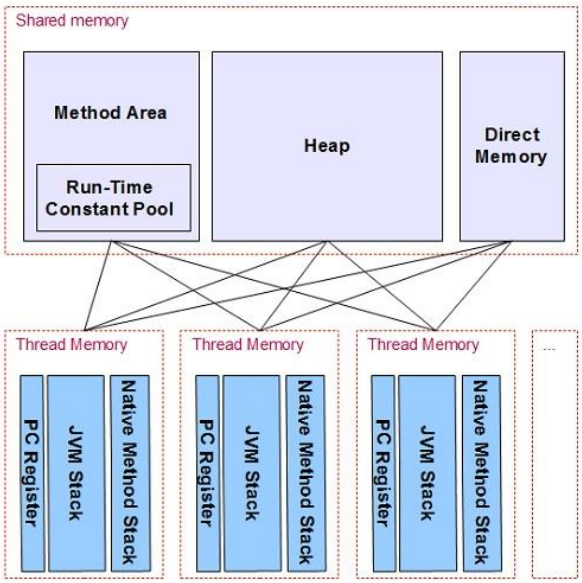
План конспекта

3. Области данных времени выполнения (виды памяти в Java)

3 Области данных времени выполнения (виды памяти в Java)

3.1 При запуске приложения JVM выделяет под ее выполнение различные области памяти: heap, stacks, program counter (pc) register, native method stacks, method area, run-time constant pool. Также отдельно нужно упомянуть об области direct memory.





PC register.

Каждый поток выполняемый виртуальной машиной Java имеет свой собственный pc register. В любой момент каждый поток выполняет код только одного метода (текущего метода для этого потока). Если этот метод не является native, то pc register содержит адрес выполняемой в данный момент инструкции JVM. Если метод, который в настоящее время выполняется потоком, является native, значение pc register JVM не определено. Также pc register также хранит адрес возврата (returnAddress) текущего метода или native-указатель на конкретной платформе (для native методов).

JVM Stacks

Каждый поток JVM имеет собственный стек, созданный одновременно с потоком. Стек используется для размещения и удаления фреймов (память для самих фреймов может быть выделена и в куче).

Память для стека виртуальной машины Java не обязательно должна быть непрерывной. Стеки JVM могут иметь фиксированный размер или динамически расширяться и сжиматься в соответствии с требованиями вычислений. При фиксированном размере стека JVM размер каждого стека

виртуальной машины Java может быть выбран отдельно при создании стека
Реализация JVM позволяет контролировать начальный размер стека (при нефиксированном размере стека), а также, в случае динамического расширения или сжатия стека JVM, предоставляет контроль над максимальным и минимальным размерами.
Если для вычисления в потоке требуется стек JVM большего размера, чем разрешено, виртуальная машина Java выдает ошибку StackOverflowError (при фиксированном стеке).
Если стеки JVM динамически расширяем и выполняется попытка расширения, но недостаточно памяти, или недостаточно памяти при создании нового стека JVM, то виртуальная машина Java генерирует OutOfMemoryError.

Heap

Heap JVM используется всеми потоками виртуальной машины Java. Куча - это область данных времени выполнения, из которой выделяется память для всех экземпляров классов и массивов. Куча создается при запуске виртуальной машины.
Объекты в куче утилизируются автоматической системой управления хранилищем (известной как сборщик мусора, *garbage collector*); объекты никогда не освобождаются явно (вручную).
Метод управления памятью в куче может быть выбран в соответствии с системными требованиями разработчика. Куча может иметь фиксированный или динамичесикй размер (причем память кучи может быть как расширена, так и сокращена).
Память для кучи не обязательно должна быть непрерывной.
Реализация JVM позволяет контролировать начальный размер кучи, а также, если кучу можно динамически расширять или сжимать, в этом случае JVM позволяет контролировать максимальный и минимальный размером кучи.
Если для вычисления требуется больше памяти кучи, чем может предоставить автоматическая система управления хранилищем, виртуальная машина Java сгенерирует ошибку OutOfMemoryError.

Method area

JVM также имеет method area (область методов), которая является общей для всех потоков виртуальной машины Java.
Method area хранит скомпилированный код классов, а также иные данные, такие как run-time constant pool, метаданные классов, статические поля (как часть метаданных класса). Область метода создается при запуске виртуальной машины.

Хотя method area логически является частью кучи, при реализации может быть принято решение не собирать мусор или не сжимать ее. Спецификация JVM не определяет область расположения method area. Method area может иметь фиксированный размер или может быть расширена в соответствии с требованиями вычислений, а может быть и сокращена, если более крупная область метода становится ненужной. Память для области метода не обязательно должна быть непрерывной.
Реализация JVM может предоставить контроль над начальным размером области метода, а также, в случае области метода переменного размера, контроль над максимальным и минимальным размером области метода.
Если память в области метода не может быть предоставлена для удовлетворения запроса на выделение, виртуальная машина Java сгенерирует OutOfMemoryError.

Run-Time Constant Pool

Пул констант времени выполнения - это представление специальной таблицы constant_pool во время выполнения для каждого класса или интерфейса в class-файле. Он содержит несколько видов констант, от числовых литералов, известных во время компиляции, до ссылок на методы и поля, которые должны быть разрешены во время выполнения.

```
class Example {
   public void hello() {
      System.out.println("Hello");
   }
}
```

```
$ javac Example.java

$ javap -c -v Example.class
```

```
public class Example
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #6.#14         // java/lang/Object."<init>":()V
   #2 = Fieldref           #15.#16        // java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #17            // Hello
   #4 = Methodref          #18.#19        // java/io/PrintStream.println:(Ljava/lang/String;)V
   #5 = Class              #20            // Example
   #6 = Class              #21            // java/lang/Object
   #7 = Utf8               <init>
.....
```

Каждый пул констант времени выполнения выделяется из области методов JVM. Пул констант времени выполнения для класса или интерфейса создается, когда создается класс или интерфейс. Если для создания пула констант времени выполнения требуется больше памяти, чем может быть доступно в области методов JVM, JVM генерирует OutOfMemoryError.

Native Method Stacks

Реализация JVM может использовать обычные стеки («стеки C») , для

поддержки native-методов (методов, написанных на языке, отличном от языка программирования Java).
Стеки native-методов также могут использоваться реализацией интерпретатора для набора инструкций JVM на таком языке, как C. Реализации JVM, которые не могут загружать native-методы и которые сами не полагаются на обычные стеки, не должны предоставлять native method stacks.
Стеки native-методов обычно выделяются для каждого потока при создании каждого потока. Спецификация позволяет стекам native-методов либо иметь фиксированный размер, либо динамически расширяться и сжиматься в соответствии с требованиями вычислений. Если стеки native-методов имеют фиксированный размер, размер каждого стека native-методов может быть выбран независимо при создании этого стека. Реализация JVM может предоставить программисту или пользователю контроль над начальным размером стеков собственных методов, а также, в случае стеков native-методов переменного размера, контроль над максимальным и минимальным размерами стека методов. Если для вычисления в потоке требуется больший стек native-методов, чем разрешено, виртуальная машина Java геерирует StackOverflowError. Если стеки native-методов могут быть динамически расширены и выполняется попытка расширения стека native-методов, но может быть предоставлено недостаточно памяти, или если недостаточно памяти может быть доступно для создания начального стека native-методов для нового потока, виртуальная машина Java генерирует OutOfMemoryError .

Frames

Фрейм используется для хранения данных и промежуточных результатов, а также для выполнения динамического связывания, возврата значений для методов и отправки исключений. Новый фрейм создается каждый раз при вызове метода. Кадр уничтожается, когда завершается вызов его метода, независимо от того, является ли это завершение нормальным или внезапным. Фреймы выделяются из стека JVM потока, создающего фрейм. Каждый фрейм имеет свой собственный массив локальных переменных, свой собственный стек операндов и ссылку на пул констант времени выполнения класса текущего метода. Кадр может быть расширен дополнительной информацией, зависящей от реализации, такой как отладочная информация. Размеры массива локальных переменных и стека операндов определяются во время компиляции и предоставляются вместе с кодом для метода, связанного с фреймом. Таким образом, размер структуры данных кадра зависит только от реализации JVM, и память для этих структур может быть выделена одновременно с вызовом метода. Только один фрейм, фрейм для

выполняемого метода, активен в любой точке данного потока управления. Этот кадр называется текущим кадром, а его метод известен как текущий метод. Класс, в котором определен текущий метод, является текущим классом. Операции с локальными переменными и стеком операндов обычно относятся к текущему кадру. Фрейм перестает быть текущим, если его метод вызывает другой метод или его метод завершается. Когда вызывается метод, создается новый фрейм, который становится текущим, когда управление передается новому методу. При возврате метода текущий фрейм передает результат своего вызова метода, если таковой имеется, в предыдущий фрейм. Затем текущий кадр отбрасывается, так как предыдущий кадр становится текущим. Обратите внимание, что кадр, созданный потоком, является локальным для этого потока и не может ссылаться на какой-либо другой поток.

Local Variables

Каждый фрейм содержит массив переменных. Длина массива локальных переменных кадра определяется во время компиляции и предоставляется в двоичном представлении класса или интерфейса вместе с кодом для метода, связанного с кадром. Одна локальная переменная может содержать значение типа boolean, byte, char, short, int, float, reference или returnAddress. Пара локальных переменных может содержать значение типа long или double. Локальные переменные адресуются путем индексации. Индекс первой локальной переменной равен нулю.

Целое число считается индексом в массиве локальных переменных тогда и только тогда, когда это целое число находится между нулем и на единицу меньше, чем размер массива локальных переменных.

Значение типа long или типа double занимает две последовательные локальные переменные. К такому значению можно обращаться только с использованием меньшего индекса. Например, значение типа double, хранящееся в массиве локальных переменных под индексом *n*, фактически занимает локальные переменные с индексами *n* и *n + 1*.

JVM не требует для *n* четности,т.е. значения типов long и double не должны быть выровнены по 64 бита в массиве локальных переменных. Разработчики могут выбрать подходящий способ представления таких значений, используя две локальные переменные.

JVM использует локальные переменные для передачи параметров при вызове метода. При вызове метода класса любые параметры передаются в последовательных локальных переменных, начиная с локальной переменной 0. При вызове метода экземпляра всегда используется локальная переменная

0 для передачи ссылки на объект, для которого вызывается метод экземпляра. Любые параметры впоследствии передаются в последовательные локальные переменные, начиная с локальной переменной 1.

Operand Stacks

Каждый фрейм содержит стек «последним вошел - первым ушел» (LIFO), известный как стек операндов фрейма. Максимальная глубина стека операндов кадра определяется во время компиляции и предоставляется вместе с кодом для метода, связанного с кадром.

Стек операндов пуст при создании содержащего его фрейма. JVM предоставляет инструкции для загрузки констант или значений из локальных переменных или полей в стек операндов. Другие инструкции JVM берут операнды из стека операндов, работают с ними и помещают результат обратно в стек операндов.

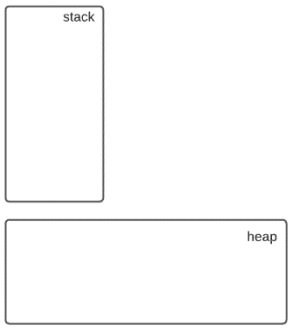
Стек операндов также используется для подготовки параметров для передачи в методы и для получения результатов метода. Например, инструкция iadd складывает два значения int вместе. Это требует, чтобы добавляемые значения int были двумя верхними значениями стека операндов, помещенными туда предыдущими инструкциями. Оба значения int извлекаются из стека операндов. Они складываются, и их сумма возвращается в стек операндов.

Подвычисления могут быть вложены в стек операндов, в результате чего получаются значения, которые могут быть использованы при вычислении охвата.

Каждая запись в стеке операндов может содержать значение любого типа JVM, включая значение типа long или типа double. Значения из стека операндов должны обрабатываться способами, соответствующими их типам. Например, невозможно отправить два значения типа int и впоследствии обработать их как long или отправить два значения с плавающей запятой и затем сложить их с помощью инструкции iadd. [https://docs.oracle.com/javase/specs/]

3.2 С первого раза понять принцип разделения и работы областей данных в Java не так просто. Для анализа материала построим упрощенную модель.

Определим область памяти **heap** и **stack**. **Heap** (куча) используется для хранения объектов, именно в ней выделяется память при создании объекта через оператор **new**. **Stack** используется для данных, необходимых при вызове методов.

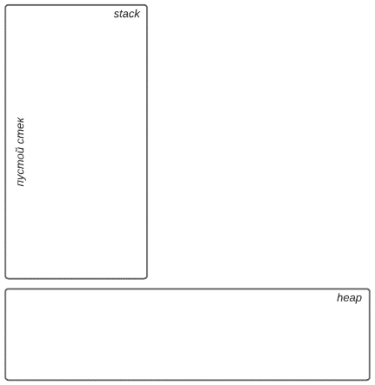
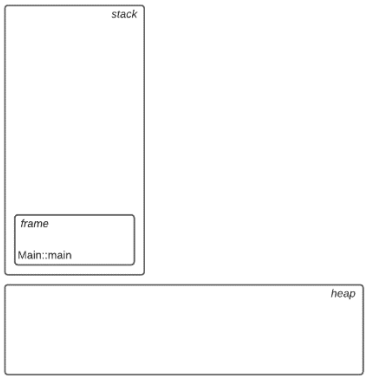
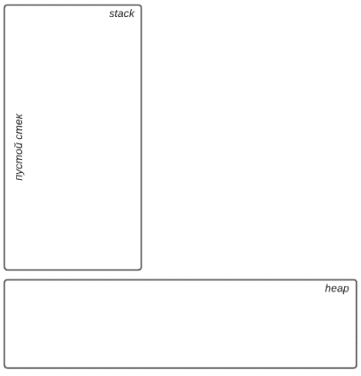


При запуске приложения и вызове метода main в стековой памяти автоматически создается фрейм, после завершения метода фрейм автоматически удаляется.

Визуально, запуск простейшего приложения с одним методом main можно представить следующим образом:

Код:
```
public class Main {
      public static void main(String[] args) {
            System.out.println("my string");
      }
}
```

1) Старт приложения, выделение памяти под стек – стек пустой.
2) Выполнение метода `main` – выделение в стеке фрейма для выполнения метода.
3) Выполнение метода `main` завершено, фрейм из стека автоматически удален – завершение работы приложения.

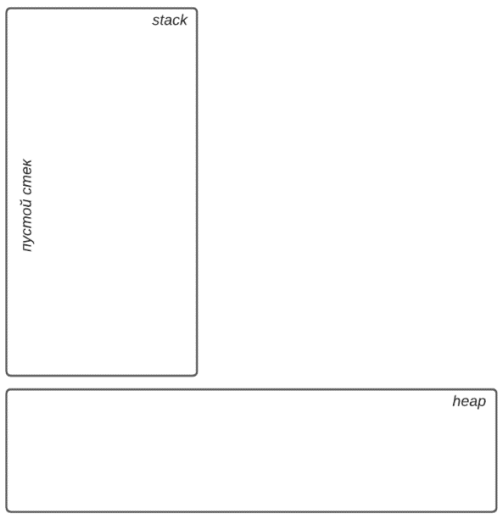
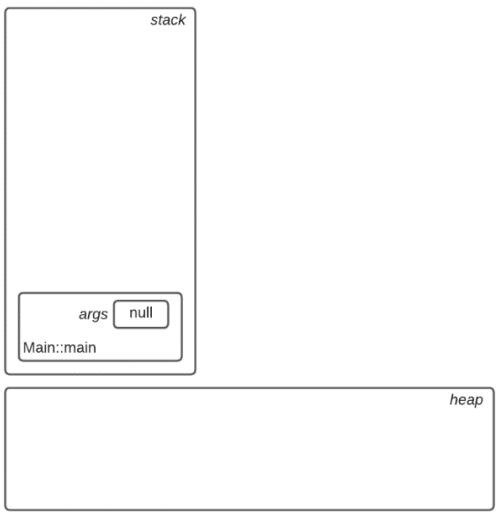
1	2	3
		

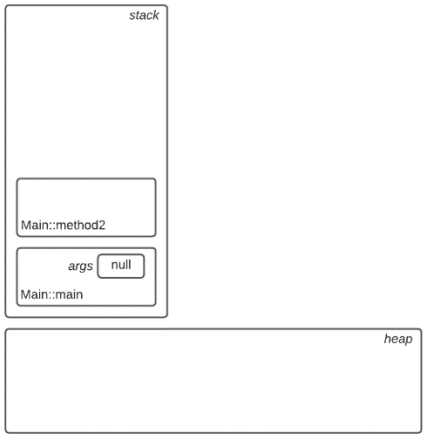
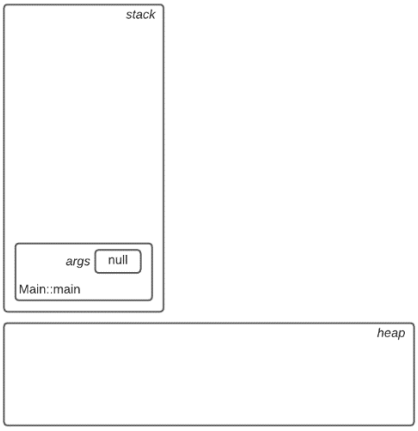
3.3 Рассмотрим, что происходит при вызове одного метода (статического) из другого (также статического).

Код:

```
public class Main {
	public static void main(String[] args) {
		method2();
	}

	public static void method2() {
		System.out.println("call method2");
	}
}
```

1) Старт приложения, выделение памяти под стек – стек пустой
2) Выполнение метода `main` – выделение в стеке фрейма для выполнения метода. В стеке создана локальная переменная `args`, т.к. при запуске не использовались параметры командной строки, то `args` равна `null`.
3) Из метода main вызывается метод `method2` – под него в стеке создается фрейм, выполняется код метода `method2`.
4) `Method2` выполнил все операторы и завершает работу, происходит возврат в точку вызова метода; фрейм, выделенный под `method2` уничтожается автоматически.
   При завершении выполнения метода main, соответствующий фрейм из стека автоматически удалится – приложение завершит свою работу.

1	2
	

3	4
	

3.4 Рассмотрим случай, когда в метод передаются параметры примитивного типа.

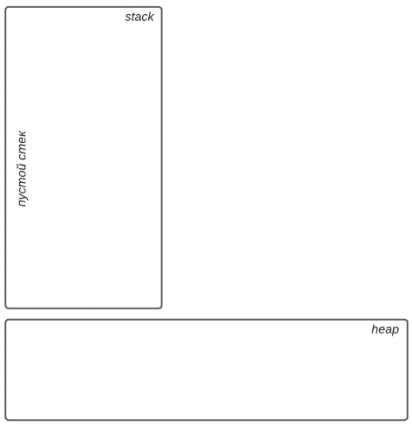
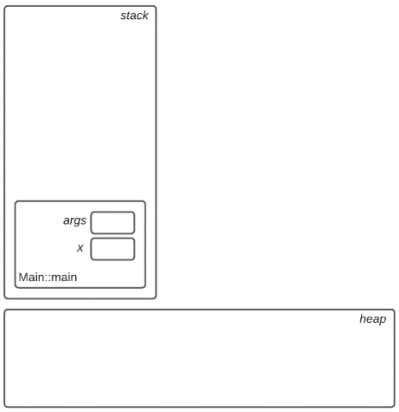
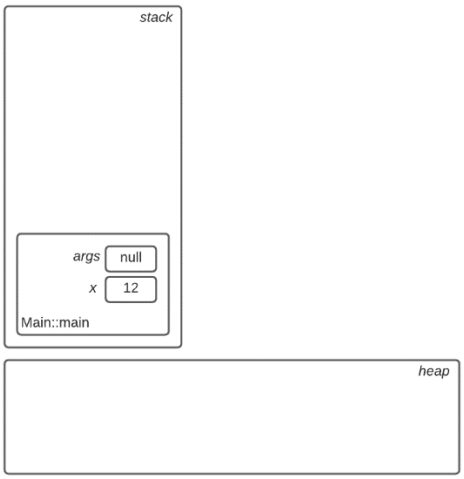
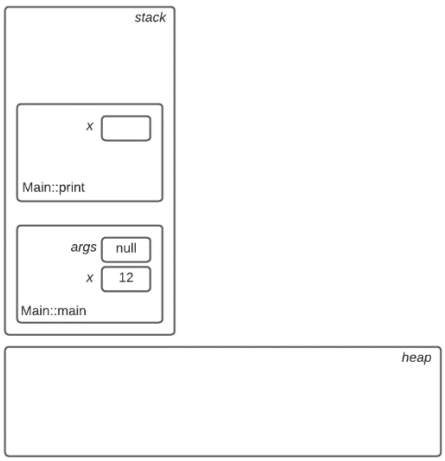
```
public static void main(String[] args) {
      int x = 12;
      print(x);
}


public static void print(int x) {
      System.out.println("x = " + x);
}
```

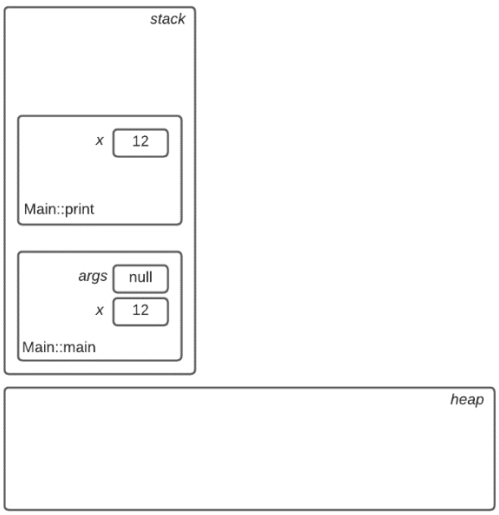
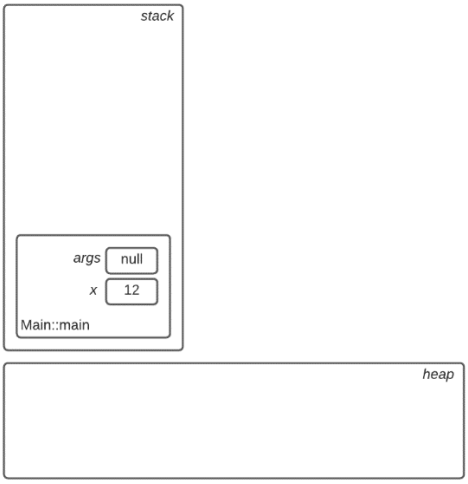
1) Старт приложения, выделение памяти под стек – стек пустой
2) Выполнение метода `main` – выделение в стеке фрейма для выполнения метода. В стеке создана локальная переменная `args`, т.к. при запуске не использовались параметры командной строки, то `args` равна `null`. В стеке создана локальная переменная `x` – при создании локальной переменной (не агрументу метода) начальное значение не присваивается.
3) Выполняется код метода. Локальной переменной `x` присваивается начальное значение 12.
4) Происходит вызов метода `print` – в стеке выделяется фрейм под вызов этого метода и во фрейме создается собственная для метода `print` локальная переменная `x`.
5) Для выполнения метода `print` передаются параметры из метода `main`. Копия значения локальной переменной `x` из метода `main` (фактический параметр) перадается (сохраняется) в аргументе метода в локальной переменной `x` метода `print` (формальный параметр).

Далее выполняется код метода `print` – значение его локальной переменной `x` выводится на консоль, далее происходит завершение работы метода `print`.

6) При завершении выполнения метода `print` соответствующий фрейм из стека удаляется (следовательно, удаляется и локальная переменная `x` этого метода).

   При завершении выполнения метода main, соответствующий фрейм из стека автоматически удалится – приложение завершает свою работу.

1	2
	
3	4
	

5	6
	

3.5 Рассмотрим случай, когда в методе параметры примитивного типа изменяются.

```
public static void main(String[] args) {
            int x = 12;
            print(x);
      }

      public static void print(int x) {
            x = x + 2;
            System.out.println("x = " + x);
      }
}
```

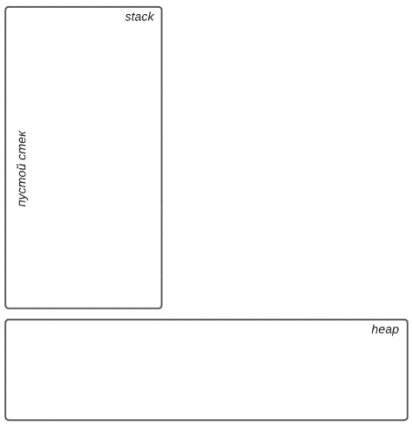
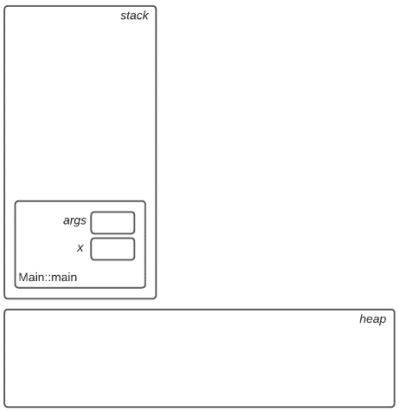
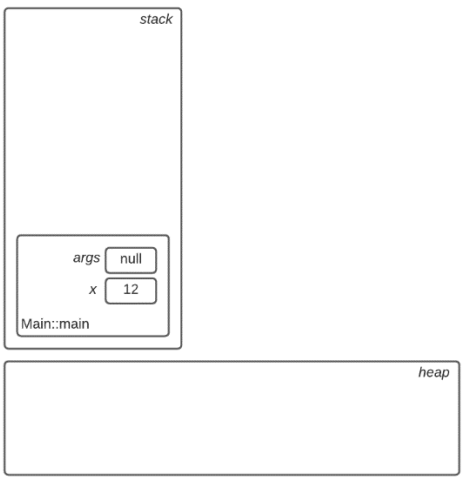
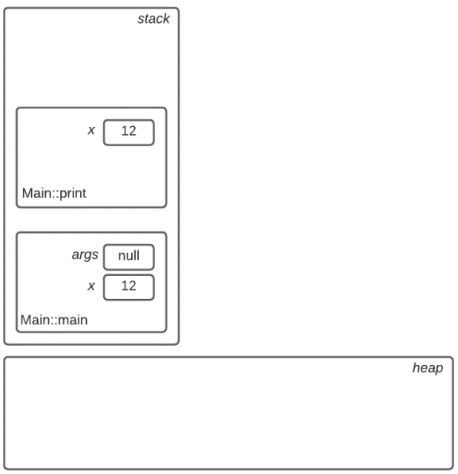
1) Старт приложения, выделение памяти под стек – стек пустой
2) Выполнение метода main – выделение в стеке фрейма для выполнения метода. В стеке создана локальная переменная `args`, т.к. при запуске не использовались параметры командной строки, то `args` равна `null`. В стеке создана локальная переменная `x` – при создании локальной переменной (не агрументу метода) начальное значение не присваивается.
3) Выполняется код метода. Локальной переменной `x` присваивается начальное значение 12.
4) Происходит вызов метода `print` – в стеке выделяется фрейм под вызов этого метода и во фрейме создается собственная для метода `print` локальная переменная `x`. Для выполнения метода `print` передаются параметры из метода `main`. Копия значения локальной переменной `x` из

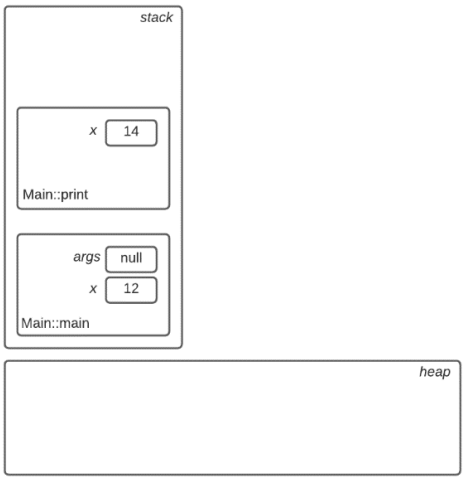
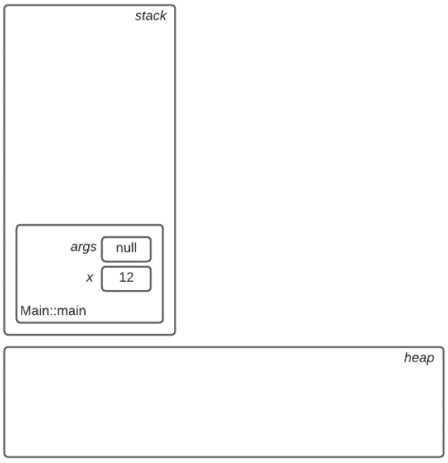
метода `main` (фактический параметр) перадается (сохраняется) в аргументе метода в локальной переменной x метода `print` (формальный параметр).

5) Далее выполняется код метода `print` – значение его локальной переменной x увеличивается на 2 (становится равным 14) и выводится на консоль, затем происходит завершение работы метода print.
6) При завершении выполнения метода `print` соответствующий фрейм из стека удаляется (следовательно, удаляется и локальная переменная x этого метода).

   При возврате в метод `main` становится доступной локальная переменная x метода `main`, которая свое значение не меняла и продолжает быть расно 12-ти.

   При завершении выполнения метода `main`, соответствующий фрейм из стека автоматически удалится – приложение завершает свою работу.

1	2
	
3	**4**
	

5	6
	

3.6 Рассмотрим случай, когда метод возвращает значение своей локальной переменной.

```
public class Main {
	public static void main(String[] args) {
		int x = 12;
		x = print(x);
	}

	public static int print(int x) {
		x = x + 2;
		System.out.println("x = " + x);
		return x;
	}
}
```

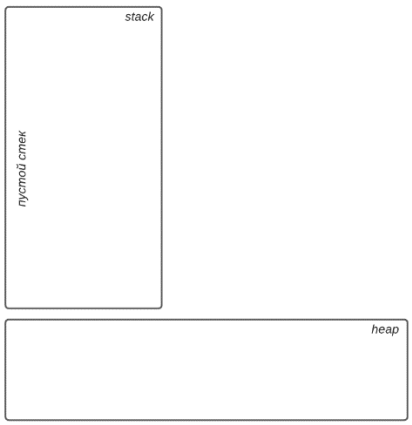
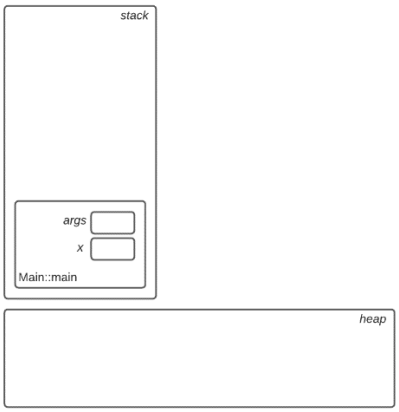
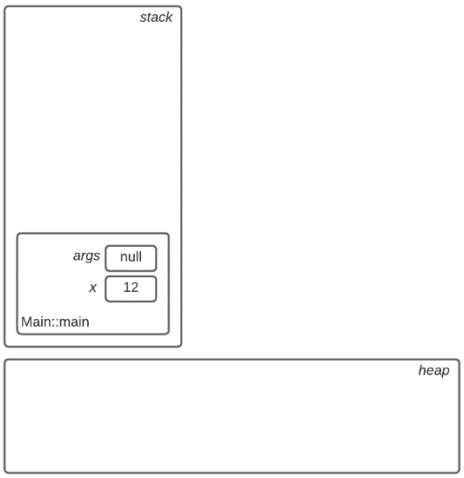
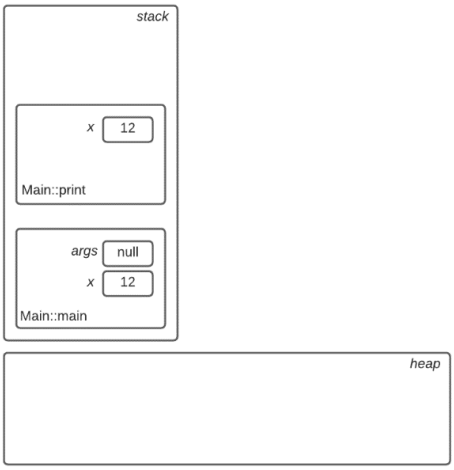
1) Старт приложения, выделение памяти под стек – стек пустой.
2) Выполнение метода main – выделение в стеке фрейма для выполнения метода. В стеке создана локальная переменная `args`, т.к. при запуске не использовались параметры командной строки, то `args` равна `null`. В стеке создана локальная переменная `x` – при создании локальной переменной (не агрументу метода) начальное значение не присваивается.
3) Выполняется код метода `main`. Локальной переменной `x` присваивается начальное значение 12.
4) Происходит вызов метода `print` – в стеке выделяется фрейм под вызов этого метода и во фрейме создается собственная для метода `print` локальная переменная `x`. Для выполнения метода `print` передаются параметры из метода `main`. Копия значения локальной переменной `x` из

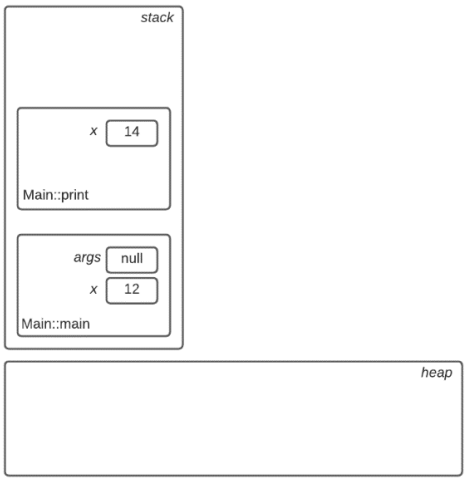
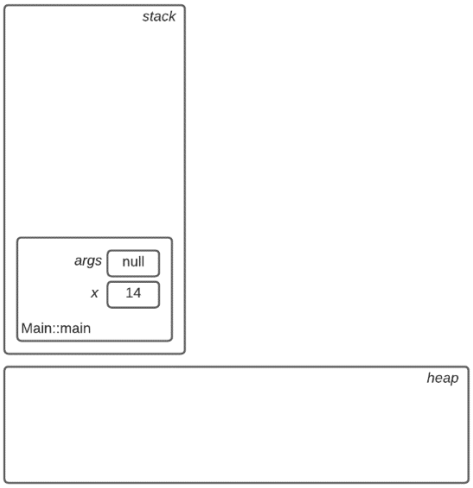
метода `main` (фактический параметр) перадается (сохраняется) в аргументе метода в локальной переменной x метода `print` (формальный параметр).

5) Далее выполняется код метода `print` – значение его локальной переменной x увеличивается на 2 (становится равным 14) и выводится на консоль, затем происходит завершение работы метода `print`.
6) При завершении выполнения метода `print` соответствующий фрейм из стека удаляется, а копия значения локальной переменной x метода `print` возвращается в точку вызова(в оператор метода `main`).

   При возврате в метод `main` становится доступной локальная переменная x метода `main`, которая меняет значение на 14, так как используется в операторе присваивания.

   При завершении выполнения метода `main`, соответствующий фрейм из стека автоматически удалится – приложение завершает свою работу.

1	2
	
3	**4**
	

5	6
	

3.7 Рассмотрим случай передачи в метод параметра ссылочного типа.

```
public class Main {
	public static void main(String[] args) {
		Date d = new Date();

		System.out.println("- " + (d.getYear()+1900));
		method(d);
		System.out.println("- " + (d.getYear()+1900));


	}

	public static void method(Date z) {
		System.out.println("-- " + (z.getYear()+1900));
		z.setYear(2121 - 1900);
		System.out.println("-- " + (z.getYear()+1900));
	}
}
```

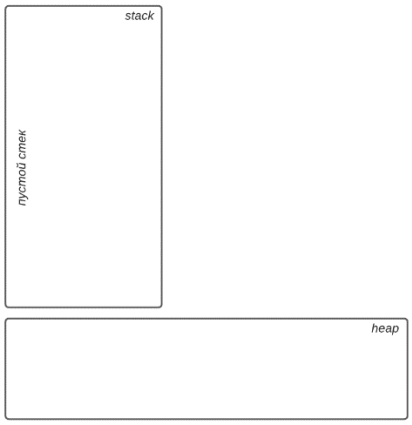
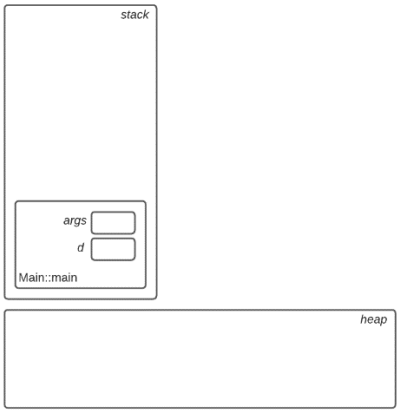
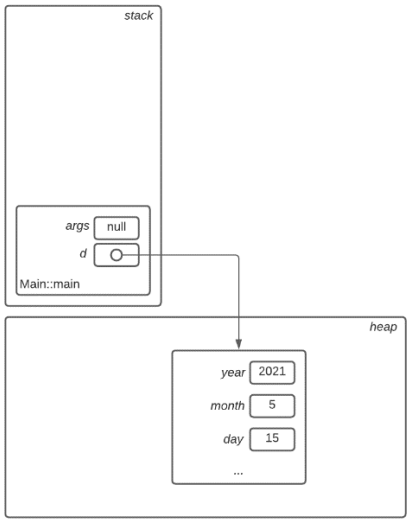
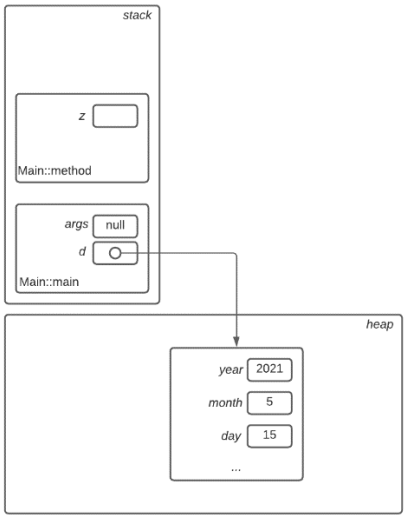
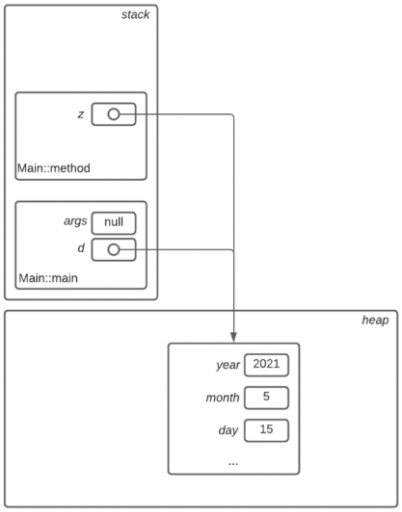
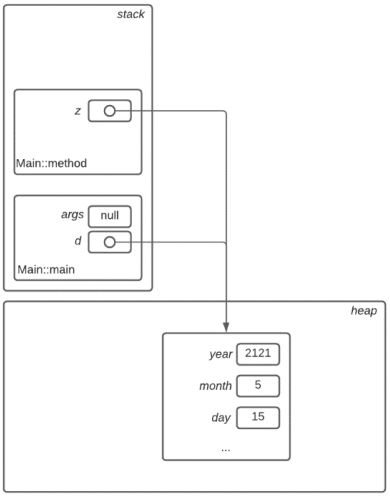
1) Старт приложения, выделение памяти под стек – стек пустой.
2) Выполнение метода `main` – выделение в стеке фрейма для выполнения метода. В стеке создана локальная переменная `args`, т.к. при запуске не использовались параметры командной строки, то `args` равна `null`. В стеке создана локальная ссылочная переменная `d` типа `Date` – при создании локальной переменной (не аргументу метода) начальное значение не присваивается.
3) Выполняется код метода `main`. В куче создается объект класса `Date` и инициализируется текущей (на момент запуска кода) датой и временем.

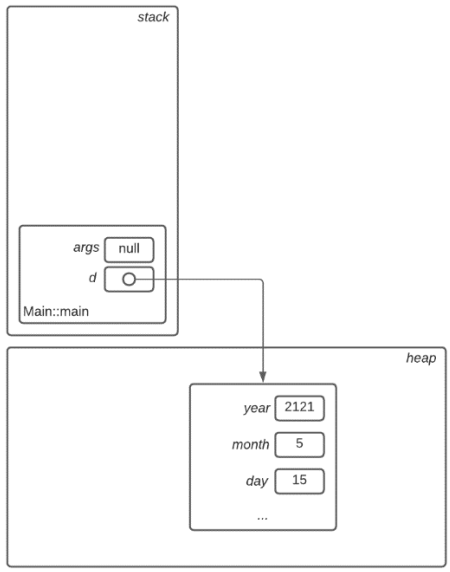
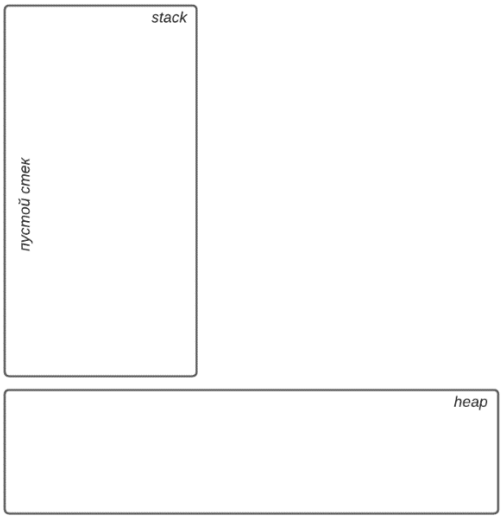
(На самом деле объекта класса Date хранит одно значение типа long – колличество миллисекунд с 1 января 1970 г., 00:00:00 GMT, но мы для наглядности отобразим на рисунке год, месяц и день как отдельные данные). Адрес созданного объекта присваивается локальной ссылке d метода main.

Далее метод main выводит значение года на консоль (в методе getYear() класса Date при возвращении значения года от него отнимается число 1900, поэтому при выводе на консоль мы его добавляем для отображения года в привычном нам виде).

4) Происходит вызов метода method – в стеке выделяется фрейм под вызов этого метода и во фрейме создается собственная для метода method локальная ссылочная переменная z типа Date. Для выполнения метода method передаются параметры из метода main.
5) Копия значения локальной ссылочной переменной d из метода main (фактический параметр) перадается (сохраняется) в аргументе метода в локальной ссылочной переменной z метода method (формальный параметр). Обратите внимание, что значением ссылочной переменной является адрес объекта, именно копия значения адреса передается при вызове метода. Далее выполняется код метода method, вызывается метод getYear() по локальной ссылке z метода method и значение года (2021) выводится на консоль.
6) Затем значение года в объекте переустанавливается с помощью вызова метода setYear() в значение 2121 (метод setYear() при установке значения года автоматически добавляет к нему число 1900, поэтому в коде мы его отнимаем до передачи в метод, таким образом год отображается в привычном виде). Вызов метода getYear() и вывод значение уже обновленного года (2121) на консоль повторяются. Код метода выполнен – происходит завершение метода method.
7) При завершении выполнения метода method соответствующий фрейм из стека удаляется и локальная ссылочная переменная z метода method также удаляется.

   При возврате в метод main становится доступной локальная ссылочная переменная z метода main, которая ссылается на старый объект, однако состояние этого объета изменилось – год остался равен 2121.
8) При завершении выполнения метода main, соответствующий фрейм из стека автоматически удалится – приложение завершает свою работу.

1	2
	
3	4
	
5	6
	

7	8
	

3.8 Рассмотрим случай передачи в метод ссылки на константный объект типа `String`.

```
public class Main {

	public static void main(String[] args) {
		String str;
		str = "Java";

		System.out.println("- " + str);
		change(str);
		System.out.println("- " + str);

	}

	public static void change(String s) {
		System.out.println("-- " + s);
		s = s + " string";
		System.out.println("-- " + s);
	}

}
```

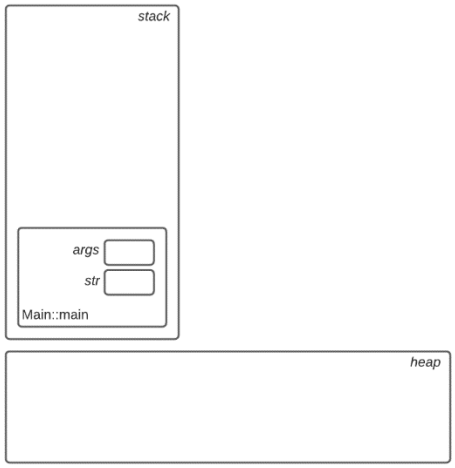
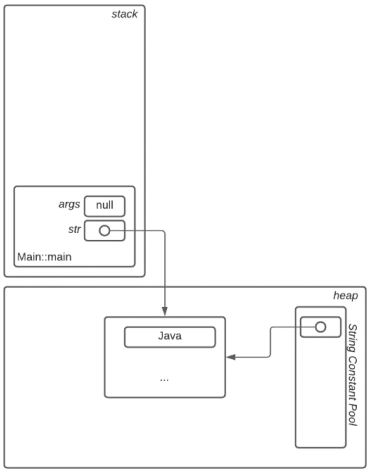
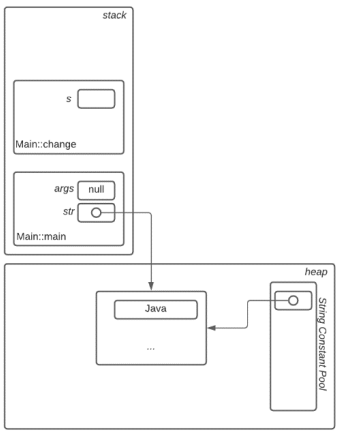
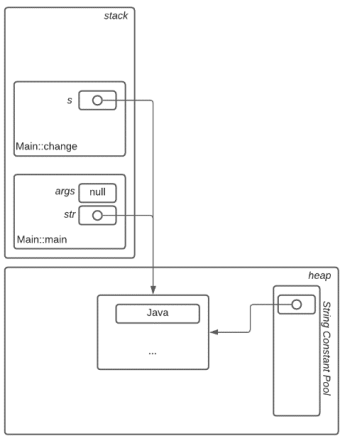
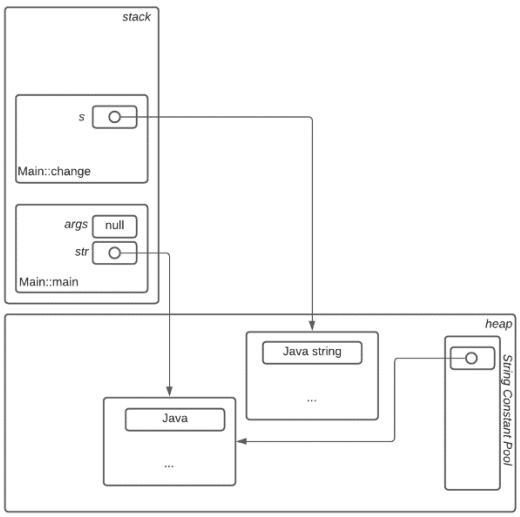
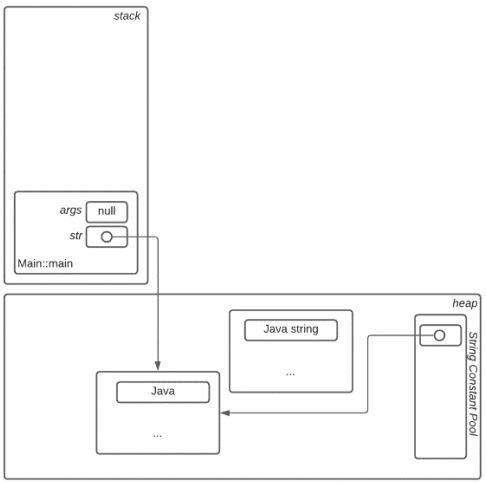
1) Старт приложения, выделение памяти под стек – стек пустой. Выполнение метода main – выделение в стеке фрейма для выполнения метода. В стеке создана локальная переменная `args`, т.к. при запуске не использовались

параметры командной строки, то args равна null. В стеке создана локальная ссылочная переменная str типа String – при создании локальной переменной (не агрументу метода) начальное значение не присваивается.

2) Выполняется код метода main. Переменная str инициализируется адресом строкового объекта со значением “Java” из пула литералов. Адрес объекта присваивается локальной ссылке str метода main.
3) Происходит вызов метода method – в стеке выделяется фрейм под вызов этого метода и во фрейме создается собственная для метода change локальная ссылочная переменная s типа String. Для выполнения метода change передаются параметры из метода main.
4) Копия значения локальной ссылочной переменной str из метода main (фактический параметр) перадается (сохраняется) в аргументе метода в локальной ссылочной переменной s метода change (формальный параметр). Обратите внимание, что значением ссылочной переменной является адрес объекта, именно копия значения адреса передается при вызове метода. Далее выполняется код метода change, значение (“Java”) строкового объекта на который указывает локальная ссылка s метода change выводится на консоль.
5) Затем выполняется оператор конкатенации строк (s + “string”) и создается новый объект с измененным состоянием (“Java string”, вспомним, что при попытке изменить состояние константного строкового объекта автоматически создается новый объект). Адрес измененного объекта сохраняется в локальной ссылочной переменной s метода change. Далее выполняется вывод на консоль значения объекта (“Java string”) на который указывает ссылка s метода change. Все операторы метода change выполнены, он завершает свою работу.
6) При завершении выполнения метода change соответствующий фрейм из стека удаляется и локальная ссылочная переменная s метода change также удаляется.

   При возврате в метод main становится доступной локальная ссылочная переменная str метода main, которая ссылается на старый объект с сотоянием “Java”. Выполняется вовод на консоль значения строкового объекта (“Java”) на который ссылается локальная ссылочная переменная str метода change.

   При завершении выполнения метода main, соответствующий фрейм из стека автоматически удалится – приложение завершает свою работу.

1	2
	
3	4
	
5	6
	

Чтобы сохранить изменения строки, произошедшие в методе change, необходимо вернуть адрес созданного в этом методе объекта.

```
public class Main {
	public static void main(String[] args) {
		String str;
		str = "Java";

		System.out.println("- " + str);

		str = change(str);
		System.out.println("- " + str);

	}

	public static String change(String s) {
		System.out.println("-- " + s);
		s = s + " string";
		System.out.println("-- " + s);
		return s;
	}
}
```

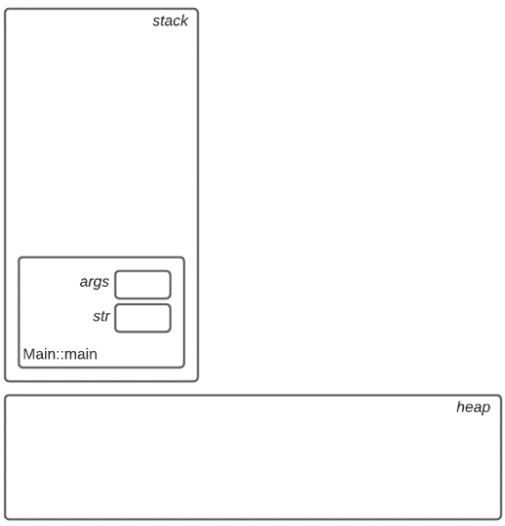
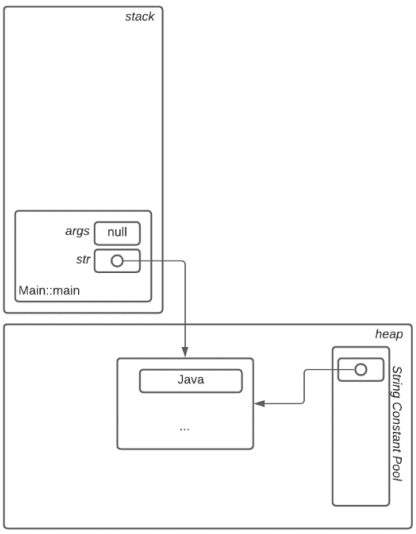
1) Старт приложения, выделение памяти под стек – стек пустой. Выполнение метода main – выделение в стеке фрейма для выполнения метода. В стеке создана локальная переменная `args`, т.к. при запуске не использовались параметры командной строки, то `args` равна `null`. В стеке создана локальная ссылочная переменная `str` типа `String` – при создании локальной переменной (не агрументу метода) начальное значение не присваивается.
2) Выполняется код метода `main`. Переменная `str` инициализируется адресом строкового объекта со значением “Java” из пула литералов. Адрес объекта присваивается локальной ссылке `str` метода `main`.
3) Происходит вызов метода `change` – в стеке выделяется фрейм под вызов этого метода и во фрейме создается собственная для метода `change` локальная ссылочная переменная `s` типа `String`. Для выполнения метода `change` передаются параметры из метода `main`.
4) Копия значения локальной ссылочной переменной `str` из метода `main` (фактический параметр) перадается (сохраняется) в аргументе метода в локальной ссылочной переменной `s` метода `change` (формальный параметр). Обратите внимание, что значением ссылочной переменной является адрес объекта, именно копия значения адреса передается при вызове метода. Далее выполняется код метода `change`, значение (“Java”)

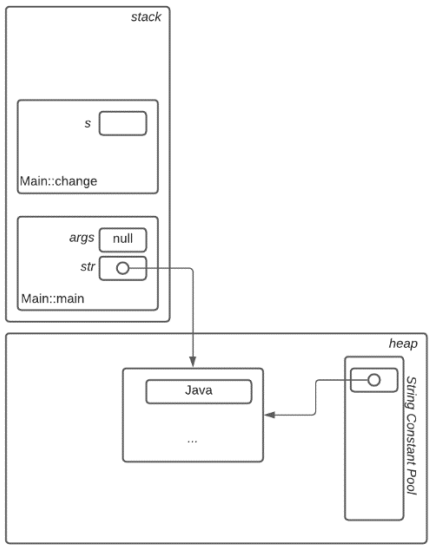
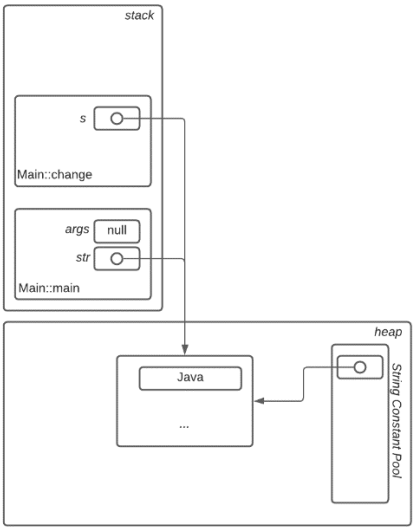
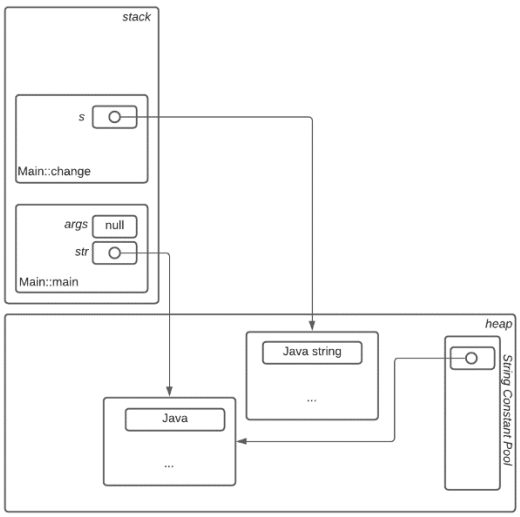
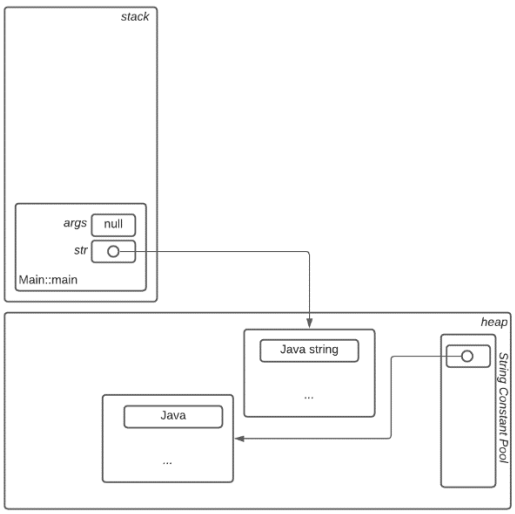
строкового объекта на который указывает локальная ссылка s метода change выводится на консоль.

5) Затем выполняется оператор конкатенации строк (s + “string”) и создается новый объект с измененным состоянием (“Java string”, вспомним, что при попытке изменить состояние константного строкового объекта автоматически создается новый объект). Адрес измененного объекта сохраняется в локальной ссылочной переменной s метода change. Далее выполняется вывод на консоль значения объекта (“Java string”) на который указывает ссылка s метода change. Все операторы метода change выполнены, он завершает свою работу.
6) При завершении выполнения метода change соответствующий фрейм из стека удаляется, значение ссылки s метода change передается в точку вызова этого метода, затем локальная ссылочная переменная s метода change также удаляется.

   При возврате в метод main становится доступной локальная ссылочная переменная str метода main, которая инициализируется возвращенным из метода change адресом измененного строкового объекта. Выполняется вовод на консоль значения строкового объекта (“Java string”) на который ссылается локальная ссылочная переменная str метода change.

   При завершении выполнения метода main, соответствующий фрейм из стека автоматически удалится – приложение завершает свою работу.

1	2
	

3	4
	
5	6
	

3.9 Рассмотрим случай передачи в метод ссылки на массив.

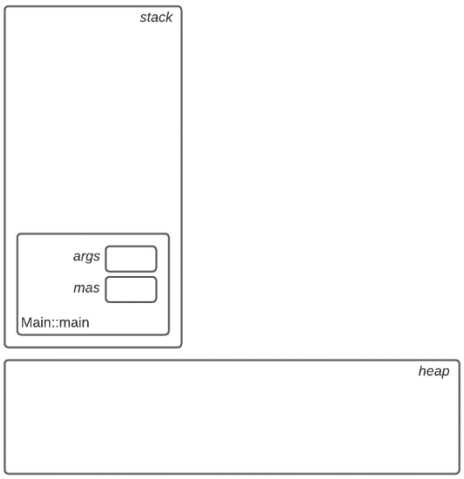
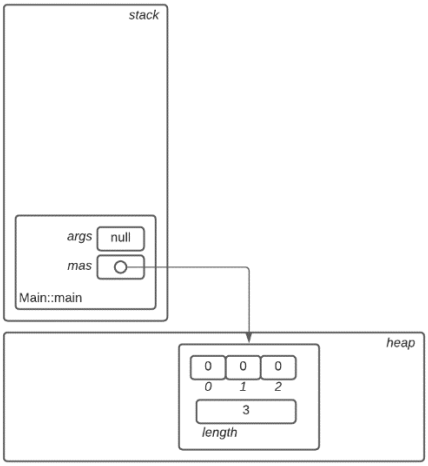
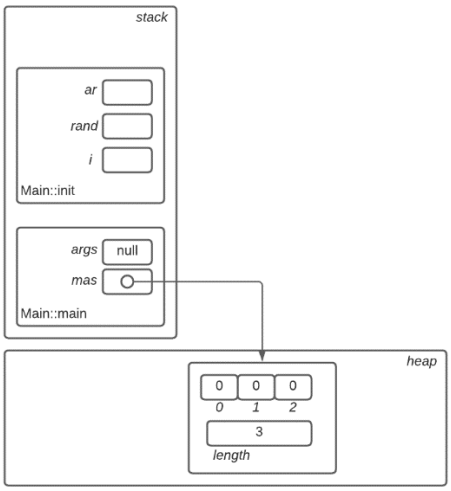
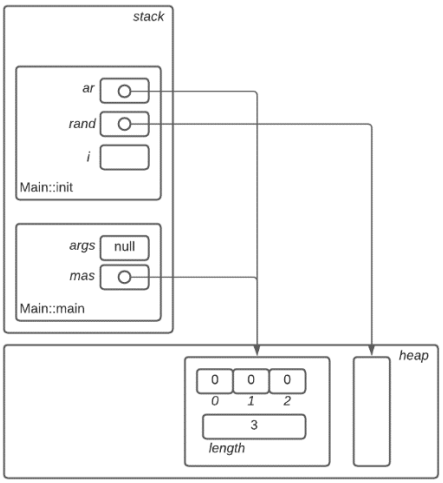
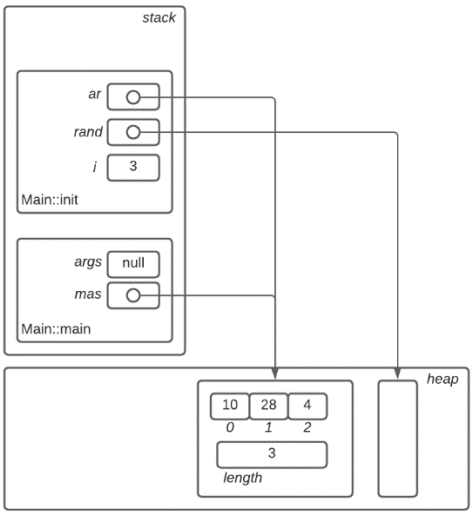
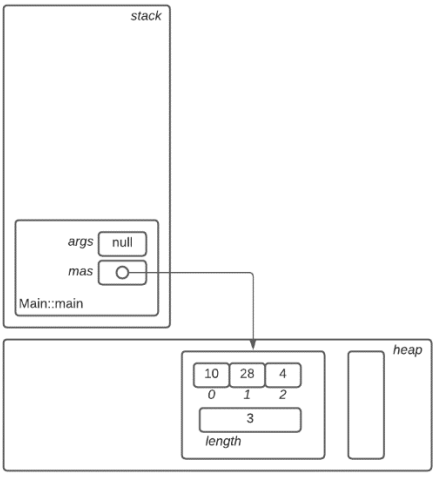
```
public class Main5 {

	public static void main(String[] args) {
		int[] mas = new int[3];
		init(mas);
	}

	public static void init(int[] ar) {
		Random rand = new Random();
		for(int i=0; i<ar.length; i++) {
```

```
                ar[i] = rand.nextInt(100);
            }
        }
}
```

1) Старт приложения, выделение памяти под стек – стек пустой. Выполнение метода main – выделение в стеке фрейма для выполнения метода. В стеке создана локальная переменная `args`, т.к. при запуске не использовались параметры командной строки, то `args` равна `null`. В стеке создана локальная ссылочная переменная `mas` типа `int[]` (ссылка на массив значений типа `int`)– при создании локальной переменной (не агрументу метода) начальное значение не присваивается.
2) Выполняется код метода `main`. В куче выделяется память под объект-массив значений типа int, количество ячеек в массиве равно 3 – в поле length объекта хранится это значение. Ссылка `mas` инициализируется адресом созданного объекта.
3) Происходит вызов метода `init` – в стеке выделяется фрейм под вызов этого метода, стек метода `init` содежит локальные переменные этого метода: две переменные ссылочного типа ar и rand, и одну примитивного типа int – переменную i. Для выполнения метода `init` передаются параметры из метода `main`.
4) Копия значения локальной ссылочной переменной mas из метода `main` (фактический параметр) перадается (сохраняется) в аргументе метода в локальной ссылочной переменной `ar` метода `init` (формальный параметр). Выполняется код метода `init`. В куче создается объект класса Random и локальная ссылочная переменная rand метода `init` инициализируется адресом этого объекта.
5) Далее выполняюся операторы метода `init` по инициализации массива на который ссылается ссылка `ar` метода `init`. Когда все операторы метода `init` выполнены, он завершает свою работу.
6) При завершении выполнения метода `init` соответствующий фрейм из стека удаляется, все локальные переменные этого метода также удаляются. На созданный объект типа Random больше не ссылается ни одна действующая ссылка – и он будет удален при следующем приходе garbage colletor-a.

   При возврате в метод `main` становится доступной локальная ссылочная переменная `mas` метода `main`, она ссылается на объект-массив, состояние которого было изменено в методе `init`.

   При завершении выполнения метода main, соответствующий фрейм из стека автоматически удалится – приложение завершает свою работу.

1	2
	
3	**4**
	
5	**6**
	

3.10 Рассмотрим случай рекурсивного вызова метода.

Рекурсия – это разработка метода таким образом, чтобы он вызывал сам себя. Рекурсивные вызовы метода должны завершаться при достижении некоторого условия. В противном случае произойдет переполнение памяти и программа «зависнет» не достигнув вычисления необходимого результата.

Напишем и разберем приложение, вычисляющее число Фибоначчи.
Последовательность Фибоначчи определяется следующим образом:

$$F_0 = 0,$$
$$F_1 = 1,$$
$$F_n = F_{n-1} + F_{n-2}.$$

Несколько первых её членов:

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ ,\ 55,\ 89,\ \ldots$$

```
public class Main {

	public static void main(String[] args) {
		int n = 3;
		int f;

		f = factorial(n);
		System.out.println("factorial = " + f);
	}

	public static int factorial(int x){

	    if (x == 1){

	        return 1;
	    }
	    x = x * factorial(x - 1);
	    return x;
	}
}
```

1) Старт приложения, выделение памяти под стек – стек пустой. Выполнение метода main – выделение в стеке фрейма для выполнения метода. В стеке создана локальная переменная `args`, т.к. при запуске не использовались параметры командной строки, то `args` равна `null`. В фрейме метода main выделена память по еще две переменные – переменную n типа int и переменную f типа int.
2) Выполняется код метода `main`. Локальная переменная n метода `main`

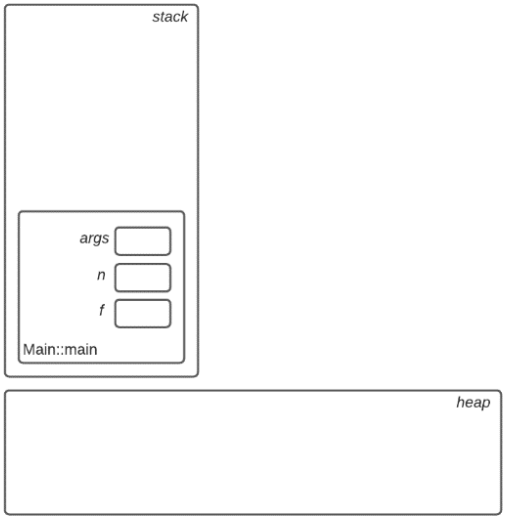
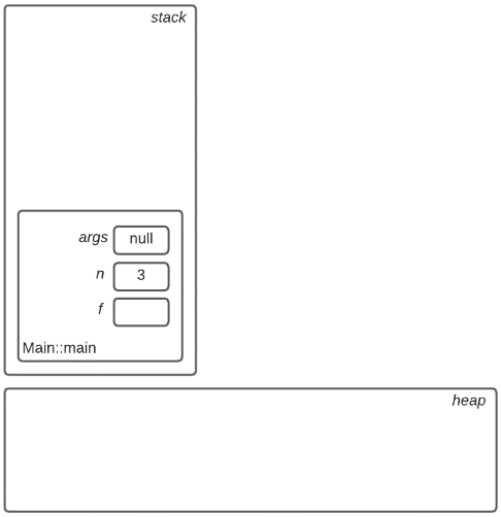
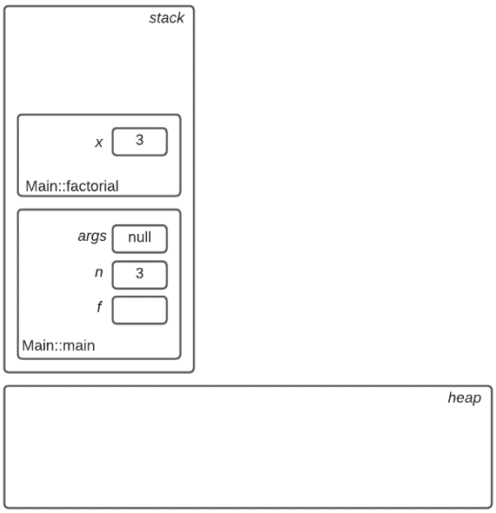
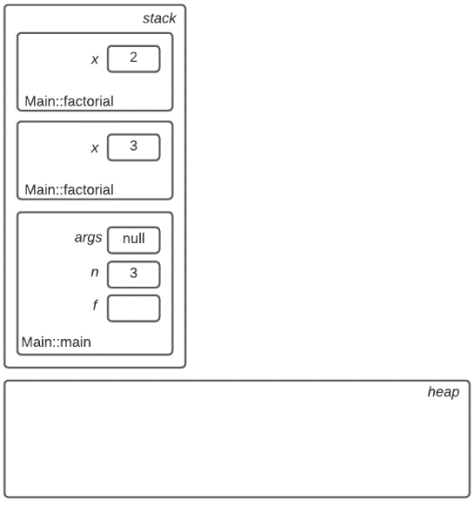
инициализируется значением 3.

3) Происходит вызов метода `factorial` – в стеке выделяется фрейм под вызов этого метода. При вызове метода `factorial` происходит передача параметров в метод – копия значения локальной переменной `n` метода `main` (3) присваивается аргументу метода `factorial` – переменной `x` типа `int`. Значение переменной x не равно 1 – оператор if пропускается, и выполняется оператор, идущий после оператора if.
   Для вычисления нового значения локальной переменной x вызывается снова метод factorial.
4) Происходит вызов метода `factorial` – в стеке выделяется фрейм под вызов этого метода. При вызове метода `factorial` происходит передача параметров в метод – копия значения локальной переменной x из вызывающего метода уменьшенная на единицу (3-1). Выполняется код метода factorial. Значение локальной переменной x текущего вызова метода factorial не равно 1 – оператор if пропускается, и выполняется оператор, идущий после оператора if.
   Для вычисления нового значения локальной переменной x вызывается снова метод factorial.
5) Происходит вызов метода `factorial` – в стеке выделяется фрейм под вызов этого метода. При вызове метода `factorial` происходит передача параметров в метод – копия значения локальной переменной x из вызывающего метода уменьшенная на единицу (2-1). Выполняется код метода `factorial`. Значение локальной переменной x текущего вызова метода `factorial` равно 1 – условие выполнения оператора `if` истинно. Оператор return завершает выполнение метода (фрейм, выделенный под вызов метода с локальными переменными будет удален), и возвращает в точку своего вызова единицу.
6) Происходит продолжение выполнение метода `factorial` в который произошел возврат, и код этого метода продолжает выполнение – высчитывается выражение: значение локальной переменной x текущего метода `factorial` (2) умножается на возвращенную единицу. Новое значение обновляет значение переменной x (x = 2 * 1 = 2). Следующий оператор `return` возвращает значение 2 в точку вызова метода `factorial`. Память, выделенная под фрейм этого метода, удаляется.
7) Происходит продолжение выполнение метода `factorial` в который произошел возврат, и код этого метода продолжает выполнение – высчитывается выражение: значение локальной переменной x текущего метода `factorial` (3) умножается на возвращенную двойку. Новое значение обновляет значение переменной x (x = 3 * 2 = 6). Следующий

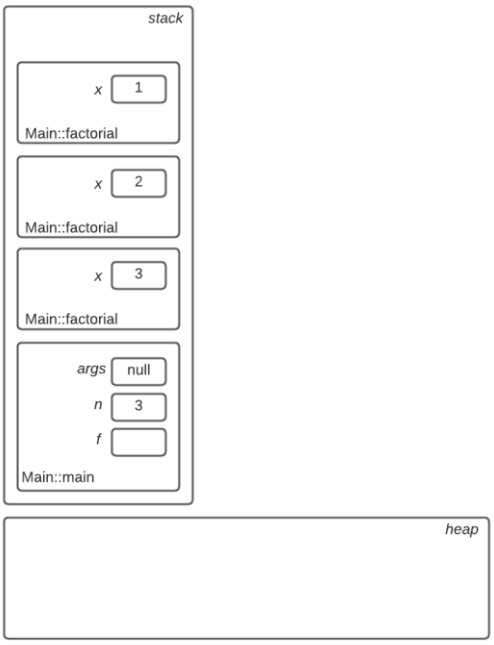
оператор `return` возвращает значение `6` в точку вызова метода `factorial` – в метод `main`. Память, выделенная под фрейм этого метода, удаляется.

8) Продолжает выполняться код метода `main`. Локальная переменная `f` инициализируется возвращенным значением 6 и ее значение выводится на консоль.

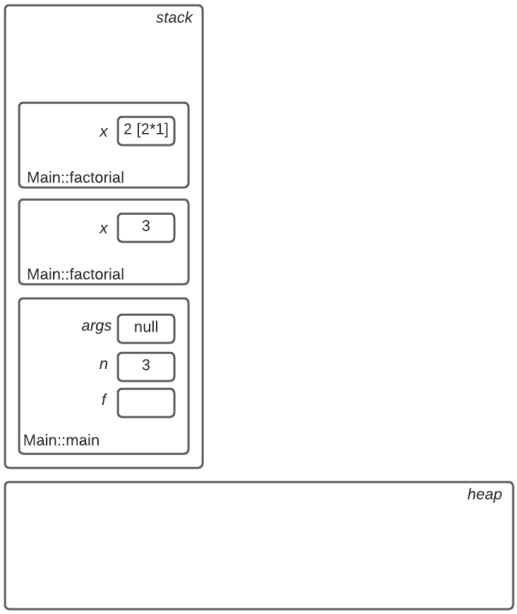
   При завершении выполнения метода main, соответствующий фрейм из стека автоматически удалится – приложение завершает свою работу.

1	2
	
3	4
	

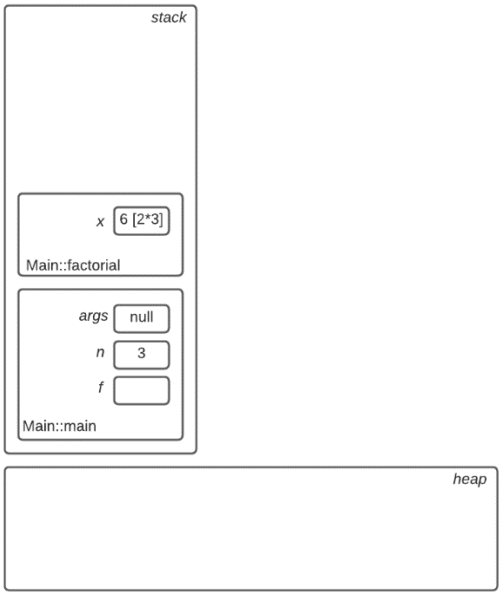
5



6



7



8