Урок 4

Многопоточность
Часть I

Многопоточность в Java. Разделяемая память, управление потоками и вопросы синхронизации. Взаимодействие потоков исполнения, взаимная блокировка.

Общие сведения

Выделяют два вида многозадачности: на основе процессов и потоков. Процесс – это исполняемая программа. Многозадачность на их основе делает возможным одновременное выполнение нескольких программ.

При многозадачности на основе потоков в одной программе могут одновременно выполняться несколько задач. Например, текстовый редактор позволяет форматировать текст во время проверки орфографии. Условие: оба действия должны выполняться в отдельных потоках. В Java программы выполняются в среде, поддерживающей многозадачность на основе процессов, но в самих программах управлять процессами нельзя.

Главное преимущество многопоточной обработки – возможность писать программы с использованием холостых периодов процессора. Это ощутимо повышает эффективность их работы.

Большинство устройств ввода-вывода (подключенных к сетевым портам, дисковых накопителей или клавиатур) работают намного медленнее центрального процессора (ЦП). Программа простаивает в ожидании информации или отправки данных на устройство ввода-вывода. При многопоточной обработке это время можно занять решением другой задачи.

В одноядерной системе параллельно выполняющиеся потоки разделяют ресурсы одного ЦП, получая по очереди квант его времени. Поэтому потоки фактически не выполняются одновременно, а используют время простоя ЦП. В многопроцессорных или многоядерных системах несколько потоков могут в действительности выполняться одновременно.

Поток может находиться в одном из нескольких состояний:

- быть выполняющимся;
- готовым к выполнению, как только он получит время и ресурсы ЦП;
- приостановленным – временно не выполняющимся;
- возобновленным в дальнейшем;
- заблокированным в ожидании ресурсов для своего выполнения;
- завершенным, когда его выполнение закончено и не может быть возобновлено.

При многозадачности на основе потоков необходим особый режим, координирующий выполнение потоков – синхронизация.

Класс Thread и интерфейс Runnable

Класс Thread и интерфейс Runnable – это основа системы многопоточной обработки в Java. Чтобы образовать новый поток, нужно создать подкласс Thread или класс, реализующий интерфейс Runnable.

В классе Thread определен ряд методов для управления потоками:

- **final String getName()** – получает имя потока;
- **final int getPriority()** – получает приоритет потока;

- **final boolean isAlive()** – определяет, выполняется ли поток;
- **final void join()** – ожидает завершения потока;
- **void run()** – определяет точку входа в поток;
- **static void sleep(long мс.)** – приостанавливает выполнение потока на указанное число миллисекунд;
- **void start()** – запускает поток, вызывая его метод **run()**.

В каждом процессе обязательно присутствует основной поток. Он получает управление уже при запуске программы. От него могут быть порождаться подчиненные потоки.

Создание потоков

Чтобы создать поток, нужно построить объект типа **Thread**. Класс **Thread** инкапсулирует объект, который может стать исполняемым. Напомним: в Java пригодные для исполнения объекты можно создавать двумя способами:

- реализуя интерфейс Runnable;
- создавая подкласс класса Thread.

Отличие этих способов только в том, как создается класс, активизирующий поток. Добавление экземпляра потока, доступ к нему и управление осуществляются средствами класса **Thread**.

Интерфейс **Runnable** предоставляет абстрактное описание единицы исполняемого кода. Для формирования потока подходит любой объект, реализующий этот интерфейс. В нем объявлен единственный метод – run():

В теле метода **run()** определяется код, соответствующий новому потоку. Отсюда можно вызывать другие методы, использовать различные классы и объявлять переменные – так же, как в основном потоке. Разница только в том, что метод **run()** создает точку входа в поток, выполняемый в программе параллельно с основным. Он выполняется до тех пор, пока не произойдет возврат из метода **run()**.

После создания класса, реализующего интерфейс **Runnable**, следует добавить экземпляр объекта типа **Thread** на основе объекта данного класса.

Конструктор класса Thread: Thread(Runnable threadOb)

В качестве параметра **threadOb** передается экземпляр класса, реализующего интерфейс **Runnable**. Таким образом устанавливается, откуда начнется выполнение потока. Запуск потока состоится только при вызове метода **start()**, объявленного в классе **Thread**. По сути, метод **start()** предназначен исключительно для вызова метода **run()**.

```
class TestThreadClass irnplernents Runnable {
public void run() {
System.out.println("Begin");
try {
for(int i=0; i < 10; i++) {
Thread.sleep(100);
System.out.println("B " + thrdNarne +", счетчик: " + count);
}
} catch(InterruptedException e) {
System.out.println("Ошибка при выполнении потока");
}
System.out.println("End");
}
}
class MainClass {
public static void main(String args[]) {
System.out.println("Запуск основного потока");
TestThreadClass mpObj = new TestThreadClass();
Thread t1 = new Thread(mpObj);
t1.start();
for(int i=0; i<10; i++) {
System.out.print(i);
try {
Thread.sleep(100);
} catch(InterruptedException exc) {
System.out.println("Прерывание основного потока");
System.out.println("Завершение основного потока");
}
             }
      }
}
```

Рассмотрим эту программу подробно. Класс **TestThreadClass** реализует интерфейс **Runnable**. Значит, объект типа **TestThreadClass** подходит для использования в качестве потока, и его можно передать конструктору класса **Thread**.

В теле метода **run()** есть цикл, счетчик которого принимает значения от 0 до 9. Метод **sleep()** приостанавливает поток, из которого он был вызван, на указанное число миллисекунд. Его нужно вызывать в блоке **try**, так как в нем может быть сгенерировано исключение **InterruptedException**.

В методе **main()** создается новый объект типа **Thread**. Для этого используется такая последовательность операторов:

```
// Сначала создать объект типа TestThreadClass
TestThreadClass mpObj = new TestThreadClass();
// Затем сформировать поток на основе этого объекта
Thread t1 = new Thread(mpObj);
// И начать выполнение потока
t1.start();
```

Как видим, сначала создается объект типа **TestThreadClass**, который затем используется для добавления объекта типа **Thread**. Его можно передать конструктору класса **Thread** в качестве параметра, поскольку класс **TestThreadClass** реализует интерфейс **Runnable**. Для запуска нового потока вызывается метод **start()**, что приводит к вызову **run()** из порожденного потока. После вызова **start()** управление возвращается в метод **main()**, где начинается выполнение цикла **for**. Он повторяется 50 раз, на каждом своем шаге приостанавливая выполнение потока на 100 миллисекунд. Оба потока продолжают выполняться, разделяя ресурсы ЦП до тех пор, пока циклы в них не завершатся.

Выполнение программы продолжается до тех пор, пока все потоки не завершат работу, при этом основной поток лучше завершать последним. В нем удобно выполнять действия по подготовке к завершению программы, закрывать файлы.

Расширение класса Thread

Рассмотрим случай реализации той же программы, только с использованием наследования от класса Thread.

Внесем изменения в код:

```
class MyThread extends Thread {
public void run() {
System.out.println("Begin");
try {
for(int i=0; i < 10; i++) {
Thread.sleep(100);
System.out.println("В " + thrdNarne +", счетчик: " + count);
}
} catch(InterruptedException e) {
System.out.println("Ошибка при выполнении потока");
}
System.out.println("End");
}
}
class MainClass {
public static void main(String args[]) {
System.out.println("Запуск основного потока");
MyThread t1 = new MyThread ();
t1.start();
for(int i=0; i<10; i++) {
System.out.print(i);
try {
Thread.sleep(100);
} catch(InterruptedException exc) {
System.out.println("Прерывание основного потока");
System.out.println("Завершение основного потока");
}
            }
      }
}
```

Какой из двух способов создания потоков предпочтительнее?

В классе **Thread** обязательному переопределению в порожденном классе подлежит только метод **run()**. Он же должен быть определен и при реализации интерфейса **Runnable**. Создавать подкласс от **Thread** следует, если требуется дополнить его новыми функциями. Если переопределять другие методы из класса Thread не нужно, то можно ограничиться только реализацией интерфейса **Runnable**. Это позволяет создаваемому потоку наследовать класс, отличающийся от **Thread**.

Приоритеты потоков

Каждому потоку присваивается приоритет, от которого зависит относительная доля процессорного времени, предоставляемого данному потоку (по сравнению с остальными активными потоками).

Приоритет – не единственный фактор, оказывающий влияние на частоту доступа потока к ЦП. Если высокоприоритетный поток ожидает доступа к ресурсу (например, для ввода с клавиатуры), то он

блокируется и вместо него исполняется низкоприоритетный поток. Но когда доступ получен, низкоприоритетный поток будет прерван.

На планирование работы потоков также влияет то, каким именно образом в операционной системе поддерживается многозадачность. Таким образом, высокий приоритет потока означает, что он ***потенциально может получить*** больше времени ЦП.

При запуске порожденного потока ему устанавливается приоритет родительского. Чтобы его изменить, необходимо вызвать метод **setPriority()** класса **Thread**. Значение параметра «уровень» должно находиться в пределах от **MIN_PRIORITY** до **MAX_PRIORITY**. Актуальное числовое соответствие этих констант – от 1 до 10. Чтобы восстановить приоритет потока, заданный по умолчанию, указываем значение 5 – это **NORM_PRIORITY**. Получить текущий приоритет можно с помощью метода **getPriority()** класса **Thread**.

Синхронизация

Синхронизация координирует выполнение нескольких потоков. Она востребована, когда потоки разделяют ресурс, который не предусматривает одновременный доступ. Когда в одном потоке выполняется запись информации в файл, второму это делать запрещено.

Синхронизация нужна и тогда, когда поток ожидает события, вызываемого другим. Тогда необходимо приостановить один из потоков до тех пор, пока не произойдет определенное событие в другом. После этого ожидающий поток может возобновиться.

Главное понятие для синхронизации в Java – это монитор, контролирующий доступ к объекту и реализующий принцип блокировки. Если объект заблокирован одним потоком, то он оказывается недоступным и для других. Когда объект разблокируется, другие потоки смогут получить к нему доступ.

У каждого объекта в Java есть свой монитор: этот механизм встроен в сам язык. Это значит, что синхронизировать можно любой объект. Для поддержки синхронизации в Java предусмотрено ключевое слово **synchronized** и ряд методов, имеющихся у каждого объекта. Синхронизировать код с помощью ключевого слова можно двумя способами.

Использование синхронизированных методов

Чтобы синхронизировать метод, в его объявлении следует указать ключевое слово **synchronized**. Когда такой метод получает управление, вызывающий поток активизирует монитор, и это приводит к блокированию объекта. Он становится недоступен из другого потока. К тому же, его нельзя вызвать из других синхронизированных методов, определенных в классе данного объекта. Когда выполнение синхронизированного метода завершается, монитор разблокирует объект. Другой поток может использовать этот метод.

Взаимодействие потоков исполнения

В рассмотренных примерах потоки исполнения безусловно блокировались от асинхронного доступа к методам. Чтобы управление стало более точным, можно организовать взаимодействие потоков.

Многопоточность заменяет программирование циклов ожидания событий. Это происходит благодаря разделению задач на дискретные, логически обособленные единицы.

Еще одно преимущество предоставляют потоки исполнения, исключая опрос. Он реализуется в виде цикла, периодически проверяющего заданное условие. Как только оно оказывается истинным,

выполняется определенное действие. Но это отнимает время ЦП.

Рассмотрим классическую задачу организации очереди, когда в одном потоке исполнения данные поставляются, а в другом – потребляются. Добавим условие: поставщик данных должен ожидать завершения работы потребителя, прежде чем сформировать новые данные.

В системах с опросом потребитель данных тратит ресурсов процессора на ожидание данных от поставщика. По завершении его работы необходимо начать опрос, напрасно расходуя лишние циклы ЦП в ожидании окончания работы потребителя данных.

Чтобы избежать опроса, в Java внедрен механизм взаимодействия потоков исполнения, задействующий методы **wait()**, **notify()** и **notifyAll()**. Они реализованы как завершенные в классе **Object** и доступны всем классам. Все три метода могут быть вызваны только из синхронизированного контекста. Правила их применения довольно просты, хотя с точки зрения вычислительной техники они принципиально прогрессивны.

1. Метод **wait()** вынуждает вызывающий поток исполнения уступить монитор. Он переходит в состояние ожидания до тех пор, пока другой поток исполнения не войдет в тот же монитор и не вызовет метод **notify()**;

2. Метод **notify()** возобновляет исполнение потока, из которого был вызван метод **wait()** для того же объекта;

3. Метод **notifyAll()** возобновляет исполнение всех потоков, из которых был вызван метод **wait()** для того же объекта. Одному из этих потоков предоставляется доступ.

Отметим, что метод **wait()** обычно ожидает до тех пор, пока не будет вызван метод **notify()** или **notifyAll()**. Но в редких случаях ожидающий поток исполнения может быть возобновлен из-за ложной активизации. Это происходит без вызова метода **notify()** или **notifyAll()**. Это маловероятно, и все же в компании Oracle рекомендуют вызывать метод **wait()** в цикле, который проверяет условие, по которому поток ожидает возобновления.

Рассмотрим пример программы, неправильно реализующей простую форму поставщика и потребителя данных. Программа состоит из четырех классов:

- **Q** – синхронизируемой очереди;
- **Producer** – поточного объекта, создающего элементы очереди;
- **Consumer** – поточного объекта, принимающего элементы очереди;
- **PC** – класса, в котором создаются объекты вышеперечисленных классов.

```
public class Q {
  int n;
  synchronized int get() {
    System.out.println(" Получено : " + n);
    return n;
  }
  synchronized void put(int n) {
    this.n = n;
    System.out.println(" Отправлено : " + n);
  }
}
public class Producer implements Runnable {
  Q q;
    Producer (Q q) {
    this.q = q;
    new Thread(this, "Поставщик").start();
  }
  public void run() {
    int i = 0;
    while (true) {
      q.put(i++);
    }
  }
}
public class Consumer implements Runnable {
  Q q;
  Consumer (Q q) {
    this.q = q;
    new Thread ( t hi s , " Потребитель " ) . s t a r t ( ) ;
  }

  public void run() {
    while(true) {
    q.get();
  }
}
public class PC {
  public static void main(String[] args) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);
  }
}
```

Методы **put()** и **get()** синхронизированы в классе Q, но ничто не остановит переполнение потребителя данными от поставщика. Кроме того, ничто не помешает потребителю дважды извлечь один и тот же элемент из очереди. В итоге будет выведен неверный результат (он зависит от быстродействия и загрузки ЦП):

Отправлено : 1

Получено : 1
Получено : 1

Получено : 1
Получено : 1
Получено : 1

Отправлено : 2
Отправлено : 3
Отправлено : 4
Отправлено : 5
Отправлено : 6
Отправлено : 7
Получено : 7

Когда поставщик отправит значение 1, запускается потребитель, который получает это значение пять раз подряд. Затем поставщик продолжает работу, подставляя значения от 2 до 7, не давая потребителю возможности получить их. Чтобы правильно реализовать взаимодействие поставщика и потребителя, применим методы **wait()** и **notify()** для передачи уведомлений в обоих направлениях:

```
public class Q {
  int n;
  boolean valueSet = false;
  synchronized int get() {
    while(!valueSet) {
      try {
        wait();
      } catch (InterruptedException e) {
        System.out.println("Exception");
      }
    System.out.println("Получено : " + n);
    valueSet = false;
    notify()
    return n;
  }
  synchronized void put(int n) {
    while(valueSet) {
      try {
        wait();
      } catch (InterruptedException e) {
        System.out.println("Exception");
      }
      this.n = n;
      valueSet = true;
      System.out.println("Отправлено : " + n);
      notify();
  }
```

В методе **get()** вызывается метод **wait()**. В итоге исполнение потока приостанавливается до тех пор, пока объект класса **Producer** не уведомит, что данные прочитаны. Тогда исполнение потока в методе **get()** возобновится.

Как только данные получены, в методе **get()** вызывается метод **notify()**. Так объект класса **Producer** уведомляется о том, что в очереди можно разместить следующий элемент данных. Метод **wait()** приостанавливает исполнение потока в методе **put()** до тех пор, пока объект класса **Consumer** не извлечет элемент из очереди. Когда исполнение потока возобновится, следующий элемент данных размещается в очереди, и вызывается метод **notify()**. Этим объект класса **Consumer** уведомляется, что он может извлечь элемент из очереди.

Результат показывает, что теперь синхронизация потоков исполнения действует корректно:

```
Отправлено : 1
Получено : 1
Отправлено : 2
Получено : 2
Отправлено : 3
Получено : 3
Отправлено : 4
Получено : 4
Отправлено : 5
Получено : 5
```

Взаимная блокировка

При работе с многопоточностью может возникнуть особый тип ошибки – взаимная блокировка (deadlock). Это происходит, когда потоки исполнения имеют циклическую зависимость от пары синхронизированных объектов.

Представим, что один поток исполнения входит в монитор объекта X, а другой – в монитор объекта Y. Если поток исполнения в объекте X попытается вызвать любой синхронизированный метод для объекта Y, он будет блокирован.

Но если поток исполнения в объекте Y попытается вызвать любой синхронизированный метод для объекта X, то этот поток будет ожидать вечно. Ведь для получения доступа к объекту X он должен снять свою блокировку с объекта Y, чтобы первый поток исполнения мог завершиться.

Взаимную блокировку трудно исправить, по двум причинам:

- она возникает очень редко, когда исполнение двух потоков точно совпадает по времени;
- она может возникнуть при участии более двух потоков исполнения и синхронизированных объектов.

Пример кода, выполнение которого приводит к взаимной блокировке потоков:

```
public class SimpleDeadLockDemo {
   public static Object l1 = new Object();
   public static Object l2 = new Object();
   public static void main(String[] a) {
      new Thread(new ThreadOne()).start();
      new Thread(new ThreadTwo()).start();
   }
   private static class ThreadOne implements Runnable {
      public void run() {
         synchronized (l1) {
            System.out.println("Thread 1: Holding lock 1...");
            try { Thread.sleep(100); }
            catch (InterruptedException e) {}
            System.out.println("Thread 1: Waiting for lock 2...");
            synchronized (l2) {
               System.out.println("Thread 2: Holding lock 1 & 2...");
            }
         }
      }
   }
   private static class ThreadTwo implements Runnable {
      public void run() {
         synchronized (l2) {
            System.out.println("Thread 2: Holding lock 2...");
            try { Thread.sleep(100); }
            catch (InterruptedException e) {}
            System.out.println("Thread 2: Waiting for lock 1...");
            synchronized (l1) {
               System.out.println("Thread 2: Holding lock 2 & 1...");
            }
         }
      }
   }
}
```

Изменение состояний потоков исполнения

Иногда исполнение потоков необходимо приостанавливать. Например, когда отдельный поток исполнения служит для отображения времени дня, а пользователю это не нужно.

Механизм остановки потока исполнения (временной или окончательной) и его возобновления был иным в ранних релизах Java. До версии Java 2 для этого использовались методы **suspend()** и **resume()**, определенные в классе **Thread**. Сегодня их применять не рекомендуется, так как:

- Метод **suspend()** из класса **Thread** способен порождать серьезные системные сбои. Представим: поток исполнения получил блокировки для очень важных структур данных. Если в этот момент приостановить его исполнение, блокировки не будут сняты. Другие потоки, ожидающие эти ресурсы, могут оказаться взаимно блокированными;
- Метод **resume()** не вызывает особых осложнений, но возможен в использовании только совместно с **suspend()**.

Кроме того, устарел метод **stop()** из класса **Thread**. Из-за него могут возникнуть сбои такого плана: поток выполняет запись в критически важную структуру данных и успел произвести лишь частичное ее обновление. Если его остановить в этот момент, структура данных может быть повреждена. Метод **stop()** снимает любую блокировку, которую устанавливает вызывающий поток исполнения. Следовательно, поврежденные данные могут быть использованы в другом потоке, ожидающем по той же самой блокировке.

За вычетом методов **suspend(), resume()** или **stop()** оказалось, что нет механизма для приостановки, возобновления или прерывания потока исполнения. Вместо этого код управления должен быть составлен так, чтобы метод **run()** периодически проверял, должно ли исполнение потока быть приостановлено, возобновлено или прервано. Для этого предназначена флаговая переменная, обозначающая состояние потока. Пока она содержит признак «выполняется», метод **run()** должен продолжать выполнение. Если же эта переменная содержит признак «приостановить», поток должен быть приостановлен. А если получен признак «остановить», то поток исполнения должен завершиться.

Рассмотрим пример применения методов **wait()** и **notify()** для управления выполнением потока. Класс **NewThread** содержит переменную **suspendFlag** для управления выполнением потока. Метод **run()** содержит блок оператора **synchronized**, где проверяется состояние переменной **suspendFlag**. Если она принимает логическое значение **true**, то вызывается метод **wait()** для приостановки выполнения потока. В методе **mysuspend()** устанавливается **true** переменной **suspendFlag**, а в методе **myresume()** – **false**. Вызывается метод **notify()**, чтобы активизировать поток исполнения.

В завершение в методе **main()** вызываются оба метода – **mysuspend()** и **myresume()**.

```
public class NewThread implements Runnable {
  String name;
  Thread t;
  boolean suspendFlag;
  NewThread(String name) {
    this.name = name;
    t = new Thread(this.name);
    System.out.println("Новый поток : " + t);
    suspendFlag = false;
    t.start();
  }
  public void run() {
    try {
    for (int i = 15 ; i > 0; i--)
      System.out.println(name + " - " + i) ;
    Thread.sleep(200);
    }
    synchronized(this) {
    }
    while(suspendFlag)
      wait();
    } catch (InterruptedException e) {
      System.out.println("n ame + " прерван . " ) ;
      System.out.println("name + " з а в е ршен . " ) ;
    }
  }
  synchronized void mysuspend() {
    suspendFlag = true;
```

```
  }
  synchronized void myresume() {
    suspendFlag = false;
    notify();
  }
}
public class SuspendResume {
  public static void main(String[] args) {
    NewThread t1 = new NewThread("1");
    NewThread t2 = new NewThread("2");
    try {
      Thread.sleep(1000);
      t1.mysuspend();
      System.out.println("Приостановка потока 1") ;
      Thread.sleep(1000);
      t1.myresume();
      System.out.println("Возобновление потока 1") ;
      t2.mysuspend();
      System.out.println("Приостановка потока 2") ;
      Thread.sleep(1000);
      t2.myresume();
      System.out.println("Возобновление потока 2 ") ;
    } catch (InterruptedException e) {
      System.out.println("Главный поток прерван");
    }
    try {
      System.out.println("Ожидание завершения потоков");
      t1.join();
      t2.join();
    } catch (InterruptedException e) {
      System.out.println("Главный поток прерван" );
      System.out.println("Главный поток завершен");
    }
  }
}
```

На примере выполнения этой программы демонстрируем, как исполнение потоков приостанавливается и возобновляется. Этот механизм сложнее прежнего, но помогает избежать ошибок.

Получение состояния потока исполнения

Поток исполнения может находиться в нескольких состояниях. Чтобы получить его текущее состояние, вызывается метод **getState()**, определенный в классе **Thread**. Он возвращает значение типа **Thread.State**, обозначающее состояние потока исполнения на момент вызова. Перечисление State определено в классе Thread.

- BLOCKED – поток приостановил выполнение, поскольку ожидает получения блокировки;
- NEW – поток еще не начал выполнение;
- RUNNABLE – поток выполняется или начнет выполняться, когда получит доступ к ЦП;
- TERMINATED – поток завершил выполнение;
- TIMED WAITING – поток приостановил выполнение на определенное время (например, после вызова метода **sleep()**, **wait()** или **join()**);

- WAITING – поток приостановил выполнение, ожидая определенного действия (например, вызова версии метода **wait()** или **join()** без заданного времени ожидания).

Состояние потока исполнения может мгновенно измениться после вызова метода **getState()**. По этой и другим причинам он не предназначен для синхронизации потоков исполнения. Он полезен при отладке или профилировании характеристик потока во время выполнения.

Домашнее задание

1. Создать три потока, каждый из которых выводит определенную букву (A, B и C) 5 раз (порядок – ABCABCABC). Используйте wait/notify/notifyAll.

2. Написать небольшой метод, в котором 3 потока построчно пишут данные в файл (по 10 записей с периодом в 20 мс).

3. Написать класс МФУ, на котором возможно одновременно выполнять печать и сканирование документов, но нельзя одновременно печатать или сканировать два документа. При печати в консоль выводится сообщения «Отпечатано 1, 2, 3,... страницы», при сканировании – аналогично «Отсканировано...». Вывод в консоль с периодом в 50 мс.

Дополнительные материалы

1. Кей С. Хорстманн, Гари Корнелл. Java. Библиотека профессионала. Том 1. Основы;

2. Стив Макконнелл. Совершенный код;

3. Брюс Эккель. Философия Java;

4. Герберт Шилдт. Java 8: Полное руководство;

5. Герберт Шилдт. Java 8: Руководство для начинающих.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Герберт Шилдт. Java. Полное руководство // 8-е изд.: Пер. с англ. – М.: Вильямс, 2012. – 1 376 с.
2. Герберт Шилдт. Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. – М.: Вильямс, 2015. – 720 с.