МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

И. В. Курбатова, А. В. Печкуров

ЯЗЫК ПРОГРАММИРОВАНИЯ JAVA

Учебное пособие

Ч а с т ь 4

Особенности языка и платформы

Воронеж
Издательский дом ВГУ
2020

УДК 004.432.2
ББК 32.973.2
К93

Р е ц е н з е н т ы:
доктор физико-математических наук, профессор *И. П. Половинкин*,
доктор физико-математических наук, профессор *Ю. А. Юраков*

Курбатова И. В.

К93 Язык программирования Java : учебное пособие/ И. В. Курбатова, А. В. Печкуров ; Воронежский государственный университет. — Воронеж : Издательский дом ВГУ, 2020.

ISBN 978-5-9273-2790-4

Ч. 4 : Особенности языка и платформы. — 99 с.

ISBN 978-5-9273-3034-8

Учебное пособие подготовлено на кафедре программного обеспечения и администрирования информационных систем факультета прикладной математики, механики и информатики Воронежского государственного университета.

Рекомендовано для студентов 3 курса факультета прикладной математики, механики и информатики Воронежского государственного университета.

УДК 004.432.2
ББК 32.973.2

ISBN 978-5-9273-3034-8 (ч.4)
ISBN 978-5-9273-2790-4

Оглавление

Введение

Сложно переоценить важность объектно-ориентированного программирования (ООП) для мира разработки программного обеспечения. Объекты и коммуникация, происходящая между ними, — это выразительный и интуитивно понятный способ моделирования систем и процессов. В их терминах можно описать программы любой сложности. Парадигма ООП сформировалась много десятилетий назад и существенно повлияла на развитие языков программирования. Такие языки, как C^{++}, C$^{\#}$ и Java, являются наиболее яркими представителями парадигмы ООП.

Данное пособие посвящено языку Java, появившемуся в 1995 году и уже третье десятилетие остающемуся актуальным. Не даром этот язык находится в глобальной десятке наиболее популярных языков программирования. На Java написано огромное количество разнообразных приложений, от банковского ПО до встраиваемых систем. Благодаря лаконичному синтаксису и гибкости Java позволяет моделировать и решать разнообразные математические задачи (см., например, [4]). Богатейшая экосистема и производительная стабильная платформа — это то, за что разработчики выбирают данный язык для своих проектов. Именно поэтому Java в качестве основного языка программирования — это отличный выбор для студентов и начинающих разработчиков.

Языку Java посвящено множество книг [1; 7–14], но, как правило, эти книги требуют достаточно высокого уровня подготовки и содержат избыточный для начинающих материал. Экосистема Java включает великое множество библиотек и технологий, что усложняет задачу формирования минимального набора знаний для начинающего программиста. В этом учебном курсе содержится такой необходимый набор знаний. Курс преследует задачу научить читателей основам языка Java и его стандартным библиотекам, а также наиболее популярным технологиям. Изложение дополнено множеством практических рекомендаций от опытных разработчиков. Подразумевается, что читатели уже знакомы с одним из процедурных языков программирования, например, C или Pascal, и знают основные понятия этих языков, такие как оператор, выражение, блок, цикл и т. д.

Не следует забывать, что содержательной стороной любого программирования являются данные и алгоритмы их обработки [2]. Оптимальный выбор структур данных и связанных с ними алгоритмов существенно вли-

яет на такие свойства программ, как производительность и потребление памяти. Освещаемые в настоящей части пособия широкие возможности стандартной библиотеки Java во многих случаях помогут читателю сделать такой выбор.

В настоящей четвертой части пособия рассматриваются углубленные особенности и возможности языка Java и платформы в целом. Повествование начинается с шаблонов проектирования, описывающих подходы к решению типовых проблем объектно-ориентированного проектирования. Затем рассматриваются такие особенности работы JVM, как сборка мусора и загрузчики классов. Наконец, освещается одна из наиболее сложных тем программирования на Java — многопоточное программирование. Материал дополнен примерами и рекомендациями, которые помогут студентам избежать основных ошибок начинающих разработчиков.

Глава 1

Шаблоны проектирования

Шаблоны проектирования (“software design patterns”) — повторяемая архитектурная конструкция, представляющая собой решение какой-либо проблемы проектирования программного обеспечения, в рамках некоторого распространенного контекста. Иными словами — это описание проблем, которые встречаются при написании объектно-ориентированного кода, и их решение.

Крайне важно понимать, что шаблоны проектирования не являются решением всех проблем и не стоит пытаться использовать их в обязательном порядке. В большинстве случаев хорошо продуманный код, написанный без явного использования шаблонов проектирования, предпочтительнее кода, написанного с их обилием. Не стоит забывать, что шаблоны — это всего лишь проработанные и описанные пути решения типовых проблем. Но стоит также отметить, что шаблоны, к месту примененные в коде программы, зачастую хорошо работают в том смысле, что другие разработчики, знакомые с примененным шаблоном, быстрее смогут понять предназначение кода при первом знакомстве с ним.

Шаблоны проектирования отличаются по уровню сложности, детализации и охвату проектируемой системы. Существуют простые шаблоны, которые не являются универсальными и применяются в рамках конкретного языка программирования. Кроме того, бывают и сложные архитектурные шаблоны, подходящие практически для любого языка, которые обычно применяются для проектирования всей программы, а не отдельных ее частей.

Шаблоны проектирования можно разделить по предназначению:

- структурные — помогают построить связи (зависимости) между объектами;

- поведенческие — помогают организовать коммуникацию между объектами;
- порождающие — нужны для гибкого создания объектов без внесения в программу лишних зависимостей.

В этой главе не ставится задача рассмотрения всех известных шаблонов проектирования. Вместо этого рассмотрим несколько популярных шаблонов с целью показать, как, пользуясь средствами ООП, можно решать те или иные проблемы. Начнем с уже знакомых по предыдущим главам шаблонов.

1. Структурные шаблоны

1.1. Посредник (proxy)

Мы уже рассматривали шаблон “посредник”, когда говорили о механизме рефлексии. В общем случае *посредник* — это объект, который является промежуточным звеном между двумя другими объектами и реализует/ограничивает доступ к объекту, к которому обращаются через него. Пример использования шаблона “посредника” можно найти в соответствующей главе третьей части пособия.

1.2. Декоратор (decorator)

Шаблон “декоратор” уже обсуждался во время рассмотрения механизма ввода/вывода. Напомним, что декоратор предоставляет гибкую альтернативу созданию подклассов с целью расширения функциональности. Простейший пример реализации этого шаблона на Java может быть следующим:

```
public interface SimplePrinter {
    void print();
}

public class MainPrinter implements SimplePrinter {

    @Override
    public void print() {
        System.out.print("Hello world!");
```

```
    }
}

public abstract class DecoratorPrinter
        implements SimplePrinter {

    protected SimplePrinter decorated;

    public DecoratorPrinter(SimplePrinter p) {
        decorated = p;
    }

    @Override
    public void print() {
        decorated.print();
    }
}

public class PrinterWithNewLine
        extends DecoratorPrinter {

    public PrinterWithNewLine(SimplePrinter p) {
        super(p);
    }

    @Override
    public void print() {
        super.print();
        System.out.println();
    }
}


public class Main {

    public static void main(String[] args) {
        DecoratorPrinter printer =
```

```
                new PrinterWithNewLine(new MainPrinter());
        printer.print();
        System.out.print("Программа закончена.");
    }
}
```

2. Поведенческие шаблоны

2.1. Итератор (iterator)

С итераторами мы уже неоднократно работали в предыдущих главах. Идея шаблона "итератор" состоит в том, чтобы вынести поведение обхода коллекции в отдельный класс. Для полноты изложения отметим, что схожая концепция в базах данных, которая позволяет обойти результат выполнения запроса, называется курсором.

Итак, *итератор* — это помощник для корректной работы цикла (обхода), своеобразный указатель, который в каждый момент времени указывает на один из элементов и может передвигаться к следующему элементу. Все реализации интерфейса `Iterable`[1] в Java поддерживают итераторы. Итератор отслеживает состояние обхода, текущую позицию в коллекции и количество элементов, которые еще осталось обойти. Одну и ту же коллекцию могут одновременно обходить различные итераторы.

Напомним использование итераторов в Java на простом примере:

```
List myList = new ArrayList();
// заполняем список
// ...
Iterator it = myList.iterator();
while (it.hasNext()) {
    Object obj = it.next();
    // производим действие над элементом списка
    // ...
}
```

[1]Различные коллекции, речь о которых идет в первой главе третьей части пособия.

2.2. Стратегия (strategy)

Стратегия — это шаблон проектирования, используемый для определения семейства алгоритмов, сокрытия реализации каждого из них и обеспечения их взаимозаменяемости. Данный шаблон решает задачу выбора по типу обрабатываемых данных (или по типу клиента) подходящего алгоритма, который требуется запустить.

Стратегия не относится к категории наиболее популярных шаблонов, однако мы рассматриваем ее в качестве иллюстрации преимуществ использования шаблонов при написании программ и, в целом, разумного подхода к объектно-ориентированному проектированию. Одним из бесспорных плюсов данного шаблона является то, что он позволяет менять выбранный алгоритм независимо от клиентского кода, использующего его.

В качестве описания и одновременной демонстрации шаблона стратегия рассмотрим следующую программу:

```
// базовый интерфейс для всех стратегий
public interface Strategy {
    int calc(int a, int b);
}

// конкретная реализация - подсчет суммы
public class SumStrategy implements Strategy {
    public int calc(int a, int b) {
        return a + b;
    }
}

// конкретная реализация - подсчет разности
public class SubStrategy implements Strategy {
    public int calc(int a, int b) {
        return a - b;
    }
}

// объект-контекст, использующий интерфейс стратегии
public class ArithmeticContext {
    private Strategy strategy;
```

```
    public void setStrategy(Strategy strategy) {
        this.strategy = strategy;
    }

    public int execute(int a, int b) {
        return strategy.calc(a, b);
    }
}

public class Main {

    public static void main(String[] args) {
        ArithmeticContext context = new ArithmeticContext();

        context.setStrategy(new SumStrategy());
        int sum = context.execute(40, 2);
        System.out.println("Результат 1: " + sum);

        context.setStrategy(new SubStrategy());
        int sub = context.execute(44, 2);
        System.out.println("Результат 2: " + sub);
    }
}
```

3. Порождающие шаблоны

3.1. Одиночка (singleton)

Одиночка — это шаблон проектирования, гарантирующий, что в приложении будет существовать не более одного экземпляра класса с глобальной точкой доступа. Чаще всего такой шаблон используют при организации доступа к некоторому ресурсу, например, конфигурации приложения.

Идея реализации этого шаблона проста и состоит в том, чтобы скрыть конструктор по умолчанию (с помощью модификатора `private`) и описать статический метод, который и будет контролировать жизненный цикл объекта-одиночки. Этот метод должен всегда возвращать один и тот же объект.

Продемонстрируем одну из возможных реализаций данного шаблона:

```
public class Singleton {

    private static final Singleton instance = new Singleton();

    // конструктор скрыт
    private Singleton() {
    }

    public static Singleton getInstance() {
        return instance;
    }

}

public class Main {

    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();
        boolean sameObject = (s1 == s2);
        System.out.println("Тот же объект: " + sameObject);
    }

}
```

Заметим, что данная реализация обеспечивает корректную работу при доступе к методу `getInstance()` из нескольких потоков, поскольку модификатор `static final` дает гарантии корректной инициализации поля средствами JVM в момент загрузки класса. О многопоточном программировании в Java мы поговорим в одной из следующих глав.

Также следует отметить, что данный шаблон нередко подвергается критике. В качестве его минусов можно указать на то, что глобальные объекты в некоторых случаях приводят к созданию немасштабируемой программы. Кроме того, использование данного шаблона усложняет написание модульных тестов (“unit test”).

3.2. Фабричный метод (factory method)

Фабричный метод — это шаблон проектирования, предоставляющий подклассам (дочерним классам) интерфейс для создания экземпляров некоторого класса. Идея фабричного метода состоит в том, чтобы создавать объекты не напрямую, используя оператор `new` и конструктор, а через вызов фабричного метода, внутри которого уже вызывается подходящий конструктор. Смысл состоит в том, что в дочернем классе можно переопределить фабричный метод, вызвав другой конструктор, тем самым изменяя тип создаваемого продукта. Дочерние классы могут производить объекты различных классов, но все они должны реализовывать один и тот же интерфейс. Зачастую фабричный метод в родительском классе объявляют абстрактным, чтобы заставить все дочерние классы реализовать его по-своему, однако это необязательное требование. Фабричный метод не обязан все время создавать новые объекты, иногда он возвращает существующие объекты из какого-то хранилища.

Данный шаблон используется в случаях, когда классу заранее не известно, объекты каких подклассов ему требуется создавать, или когда класс задуман так, чтобы объекты, которые он создает, определялись подклассами.

Заметим, что существует еще шаблон “фабрика” (или абстрактная фабрика), схожая по идеям с данным шаблоном. Этот шаблон мы обсуждать не будем.

Продемонстрируем на примере простую реализацию шаблона “фабричный метод”:

```
public interface Dog {
}

public class Dalmatian implements Dog {
}

public class Wolfhound implements Dog {
}

// класс-создатель, описывающий фабричный метод
public abstract class Creator {
    public abstract Dog factoryMethod();
}
```

```
// конкретный класс, реализующий фабричный метод
public class DalmatianCreator extends Creator {
    @Override
    public Dog factoryMethod() {
        return new Dalmatian();
    }
}

public class WolfhoundCreator extends Creator {
    @Override
    public Dog factoryMethod() {
        return new Wolfhound();
    }
}

public class Main {

    public static void main(String[] args) {
        // список объектов-создателей
        List<Creator> creators = Arrays.asList(
                new DalmatianCreator(),
                new WolfhoundCreator()
        );
        // обход списка и создание объектов
        for (Creator creator : creators) {
            Dog dog = creator.factoryMethod();
            System.out.println("Собака: " + dog);
        }
    }

}
```

3.3. Строитель (builder)

Строитель — шаблон проектирования, который дает более простой способ создания составного объекта, нежели конструктор со множеством аргументов. Шаблон разбивает процесс конструирования объекта на от-

дельные шаги. Для создания объекта нужно по очереди вызывать методы объекта-строителя. При этом зачастую можно пропускать часть шагов и выполнять только те, что нужны для производства объекта с определенной конфигурацией.

Продемонстрируем шаблон “строитель” на примере email-письма:

```
public class EmailMessage {

    private String to;
    private String from;
    private String subject;
    private String content;

    public EmailMessage(
            String to, String from,
            String subject, String content
    ) {
        this.to = to;
        this.from = from;
        this.subject = subject;
        this.content = content;
    }

    public String getTo() {
        return to;
    }

    public String getFrom() {
        return from;
    }

    public String getSubject() {
        return subject;
    }

    public String getContent() {
        return content;
    }
```

```
@Override
public String toString() {
    return "EmailMessage{" +
            "to='" + to + '\'' +
            ", from='" + from + '\'' +
            ", subject='" + subject + '\'' +
            ", content='" + content + '\'' +
            '}';
}

public static Builder newBuilder() {
    return new Builder();
}

// класс-строитель
public static class Builder {

    private String to;
    private String from;
    private String subject;
    private String content;

    private Builder() {
    }

    public Builder to(String to) {
        this.to = to;
        // возвращаем сам объект-строитель
        // чтобы можно было строить цепочки вызовов
        return this;
    }

    public Builder from(String from) {
        this.from = from;
        return this;
    }
```

```
        public Builder subject(String subject) {
            this.subject = subject;
            return this;
        }

        public Builder content(String content) {
            this.content = content;
            return this;
        }

        public EmailMessage build() {
            return new EmailMessage(
                    to, from,
                    subject, content
            );
        }
    }

}

public class Main {

    public static void main(String[] args) {
        // пример использования объекта-строителя
        EmailMessage msg =
                EmailMessage.newBuilder()
                        .to("Турецкий султан")
                        .from("Казаки")
                        .subject("Запорізькі козаки " +
                                "турецькому султану!")
                        .content("Ти, султан, ч...")
                        .build();
        System.out.println("Сообщение: " + msg);
    }

}
```

Отметим, что иногда этот шаблон реализуют через абстрактный класс-строитель и дочерние классы, конструирующие объект с различными значениями полей. Тогда этот шаблон отделяет создание сложного объекта от его представления тем, что в результате одного и того же процесса конструирования могут получаться различные представления.

В данной главе мы рассмотрели некоторые хорошо известные шаблоны проектирования и сформировали определенное представление о предмете. Любознательный читатель может продолжить знакомство с шаблонами проектирования, например, прочитав книгу [3].

Глава 2

Сборка мусора

Сборка мусора (“garbage collection” или “GC”) является краеугольной особенностью архитектуры Java (и JVM). И хотя непосредственно на процесс сборки мусора в Java программист повлиять не может, понимать его особенности крайне важно для написания предсказуемого и достаточно производительного кода. В большинстве учебников авторы никак не освещают эту тему в силу ее большого объема и насыщенности. В этой главе мы поговорим об основных концепциях и нюансах сборки мусора в Java. После прочтения данной главы заинтересовавшимся данной темой читателям советуем почитать более детальную информацию, доступную в Интернете. В качестве начального материала советуем ознакомиться со следующей статьей: `https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html`.

1. Базовые концепции

В относительно низкоуровневых языках программирования, таких как C и C++, требуется осуществлять ручное управление памятью, т. е. аллоцировать и деаллоцировать объекты в памяти путем вызова системных функций (или использовать немного более высокоуровневые средства, например, умные указатели, “smart pointers” в C++). Кроме того, в этих языках есть возможность работать с указателями на адрес в виртуальном адресном пространстве и даже выполнять арифметические операции над указателями. Все эти инструменты безусловно являются очень мощными и позволяют писать максимально эффективный код. Однако написание и отладка такого кода требует особой дисциплины и занимает довольно много времени.

Java принципиально отличается от более низкоуровневых языков программирования, таких как C и C++, наличием сборки мусора. От программиста не требуется аллоцировать и деаллоцировать объекты в памяти, а также заботиться об актуальности участка памяти, на который ссылается указатель. В Java просто-напросто нет подобных конструкций на уровне языка. Все, что требуется от программиста, это конструировать объекты, затем работать со ссылками, а в момент, когда объект или примитивное значение уже не нужны, позаботиться о том, чтобы никакие переменные и другие объекты, которые еще используются в программе, не ссылались на этот объект или значение. Впрочем, последнее действие в большинстве случаев даже не требует особого внимания и дополнительных действий. Когда какой-то объект или цепочка ссылающихся друг на друга объектов, или примитивное значение становятся недостижимыми из программы, специальный модуль JVM, называемый сборщиком мусора ("garbage collector"), во время одной из последующих сборок мусора освободит память, зарезервированную под них.

Поговорим немного подробнее о понятии достижимости. Объект считается достижимым, а значит, не попадает под действие сборщика мусора, если до него можно добраться по цепочке ссылок, начиная с какой-либо корневой ссылки, т. е. ссылки, используемой в выполняемом коде. Очевидно, что объект считается недостижимым, т. е. попадает под действие сборщика мусора, в обратном случае, когда до него нельзя добраться через какую-либо корневую ссылку.

Корневыми ссылками в Java-программе считаются следующие:

- локальные переменные, которые содержатся в стеке одного из потоков программы;
- активные потоки ("thread") программы и связанные с ними данные (например, "thread local variables");
- статические переменные, ссылки на которые хранятся в объектах `Class`;
- ссылки JNI[2] ("Java Native Interface"), т. е. ссылки на объекты, созданные в процессе JNI вызовов.

Для расположения объектов в памяти JVM использует концепцию *кучи* ("heap"). Это область памяти, которая резервируется в момент запуска

[2]Вкратце этот механизм позволяет делать вызовы функций C/C++ из Java-программы, и наоборот.

JVM. Во время выполнения программы куча может быть увеличена или уменьшена самой JVM (но не из кода программы) при наступлении определенных условий, например, в момент, когда памяти в куче становится недостаточно и не удается ее освободить путем сборки мусора[3]. Кроме того, при определенной степени заполненности и фрагментированности кучи происходит сборка мусора[4]. Обратим внимание на то, что помимо самой кучи JVM отдельно хранит в памяти дополнительные структуры данных, такие как стеки потоков, кеш для кода методов, полученного после JIT-компиляции, и т. д.

Применительно к работе с памятью, сборка мусора — это процесс освобождения места в куче, что позволяет в дальнейшем добавлять на освободившееся место в куче новые объекты.

С точки зрения программиста, наличие сборки мусора в Java обеспечивает полностью автоматическое и безопасное[5] управление памятью. В следующем параграфе процесс сборки мусора рассматривается подробнее.

2. Сборщик мусора

Как уже говорилось, *сборщик мусора* — это модуль JVM, отвечающий за процесс сборки мусора, т. е. за определение моментов времени, когда необходимо освободить место в куче, само освобождение места, а также за сбор статистической и прочей информации, необходимой для перечисленных действий. Реализация сборщика мусора зависит от конкретной реализации JVM и ее версии. Мы обсудим реализацию сборки мусора на виртуальной машине HotSpot, одной из наиболее популярных реализаций JVM.

В ранних версиях Java сборщик мусора был относительно примитивным. Он работал в одном потоке и периодически осуществлял сборку мусора, во время которой наступала так называемая “stop the world” пауза, что означало полную остановку выполнения всех потоков программы

[3]Задать начальный и максимальный размер кучи можно при запуске JVM с помощью параметров `-Xms` и `-Xmx` командной строки. Описание этих и многих других параметров JVM доступно в официальной документации.

[4]Механизм определения наступления сборки и сама логика сборки зависят от типа сборщика мусора. Позже мы поговорим об этом подробней.

[5]До определенных пределов, разумеется. Наличие сборки мусора не защищает от ”утечек памяти” по причине некорректно написанного кода, порождающего и хранящего ссылки на все новые и новые объекты.

до момента завершения сборки мусора. В последующем создавались более совершенные сборщики мусора, в которых, например, добавили промежуточные этапы сборки мусора, задействовали несколько ядер процессора на многоядерных процессорах и ввели другие усовершенствования.

В целом, по мере развития платформы сборщики мусора в HotSpot постоянно усовершенствуются. В новых версиях Java зачастую добавляются новые типы сборщиков мусора, сначала в качестве экспериментальной функции, а затем они могут быть перенесены в статус сборщика мусора по умолчанию. Разные типы сборщиков мусора могут хорошо подходить для специфических сценариев использования, например, для минимизации продолжительности пауз для сборки мусора или для работы с большими размерами кучи.

Поговорим теперь о некоторых особенностях различных реализаций сборщиков мусора. На первый взгляд кажется, что в идеале сборщик мусора должен как можно быстрее избавляться от недостижимых объектов, не затрачивая при этом лишние ресурсы компьютера. Но важно понимать, что на сборку мусора тратятся те же ресурсы компьютера, что и на саму программу, а значит, она не "бесплатна". Как уже говорилось, не существует сборщика мусора, одинаково хорошо справляющегося со всеми сценариями работы.

При оценке эффективности работы сборщика мусора учитывают следующие аспекты:

- максимальное время задержки — это максимальное время "stop the world" паузы;
- пропускная способность — это отношение общего времени работы программы к суммарному времени простоя во время сборки мусора, причем оценка делается на длительном промежутке времени;
- потребляемые ресурсы — это объем вычислительных ресурсов процессора и/или памяти, потребляемые сборщиком.

К сожалению, достичь максимальной эффективности по всем этим показателям одновременно практически невозможно.

Отметим, что независимо от типа сборщика мусора общей рекомендацией является выделение под кучу заведомо бóльшего объема памяти, чем требуется программе. В противном случае свободной памяти в куче будет мало, что усложнит задачу сборщика мусора и приведет к деградации производительности программы из-за частых сборок мусора.

Как уже говорилось, у Java-программиста нет возможности непосредственно из кода повлиять на сборку мусора после запуска программы, например, вызвать сборщик мусора в нужный момент времени. Многих вводит в заблуждение наличие метода `gc()`, находящегося в системном классе `System`. В действительности вызов `System.gc()` не приводит к мгновенной сборке мусора, а уведомляет JVM, что необходимо произвести сборку мусора. В свою очередь момент сборки мусора отдается на усмотрение конкретной реализации JVM, и данный метод не требует от виртуальной машины никаких гарантий по времени начала сборки.

Мы намеренно не будем перечислять и рассматривать все типы сборщиков мусора в HotSpot, так как этот список со временем меняется. Вместо этого рассмотрим основную логику работы сборщиков мусора в виртуальной машине HotSpot.

3. Основная логика сборщиков мусора в JVM

В данном параграфе обсудим общие особенности и логику работы сборщиков мусора в JVM (HotSpot).

Базовый процесс сборки мусора может быть описан следующими шагами:

- шаг 1 (“marking”): отмечаются недостижимые объекты; этот шаг может быть очень продолжительным по времени, если сканируются сразу все объекты, имеющиеся в приложении;
- шаг 2 (“deletion”): удаляются из памяти объекты, отмеченные на шаге 1 как недостижимые; ссылки на освобожденные блоки памяти при этом сохраняются средствами JVM;
- шаг 2a (“deletion with compacting”): повторяет логику шага 2, но помимо удаления недостижимых объектов из памяти, оставшиеся объекты переносятся на смежные участки памяти как можно компактней; благодаря этим дополнительным действиям в дальнейшем аллокация новых объектов в памяти происходит гораздо быстрее.

Работа сборщика мусора на каждом из описанных выше трех шагов иллюстрируется на рис. 2.1.

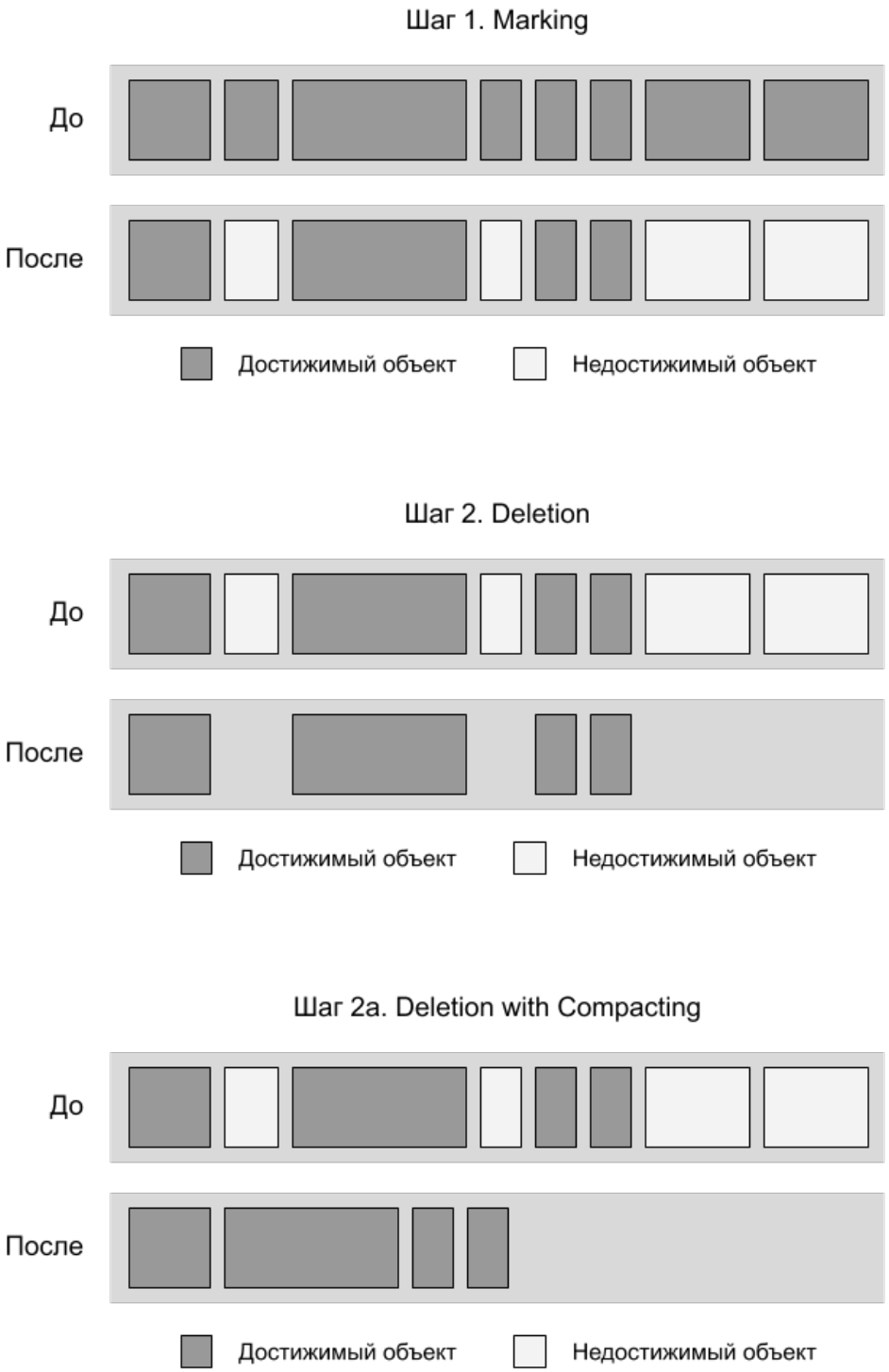

Рис. 2.1. Три последовательных шага, выполняемые сборщиком мусора

Практика показывает, что объекты, созданные недавно, становятся недостижимыми чаще, чем уже существующие долгое время[6]. Поэтому сборщики мусора в JVM разделяют все объекты в программе на области, называемые поколениями, т. е. группы объектов с близким временем создания. Если память, выделенная под одно из поколений, заканчивается, в нем и во всех более молодых поколениях производится сборка мусора. При этом неиспользуемые объекты в поколениях удаляются, а оставшиеся объекты перемещаются в старшее поколение. Такой подход сокращает время, необходимое на фазу сборки мусора, поскольку в ходе сборки требуется просматривать меньшее число объектов. Однако есть и минус — этот подход требует отслеживания ссылок между разными поколениями.

В JVM куча разделяется на следующие области:

- младшее поколение ("young generation") — содержит недавно сконструированные объекты; состоит из нескольких подразделов ("eden", в который помещаются объекты изначально, и "survivor space", в который перемещаются объекты из "eden", пережившие сборку мусора);
- старшее поколение ("old generation") — используется для хранения объектов, переживших несколько сборок мусора;
- постоянное поколение ("permanent generation" или "PermGen")[7] — содержит условно постоянные данные, например, метаданные JVM, т. е. объекты `Class`.

Рисунок 2.2 иллюстрирует структуру кучи.

В соответствии с этим разделением кучи на области сборщик осуществляет разные типы сборки мусора:

- промежуточная сборка ("minor GC") — выполняется при заполнении области младшего поколения;
- полная сборка ("full GC/major GC") — выполняется при заполнении кучи; сначала запускается сборка для младшего поколения, затем для старшего;
- сборка в старшем поколении ("old generation GC") — промежуточная сборка мусора в области старшего поколения; осуществляется некоторыми видами сборщиков мусора (например, G1 и CMS).

[6]Это утверждение часто называют "слабой гипотезой о поколениях" ("weak generational hypothesis").

[7]В последних версиях Java эту область заменили на другую, называемую "metaspace". Отличия этой области от "PermGen" мы рассматривать не будем.

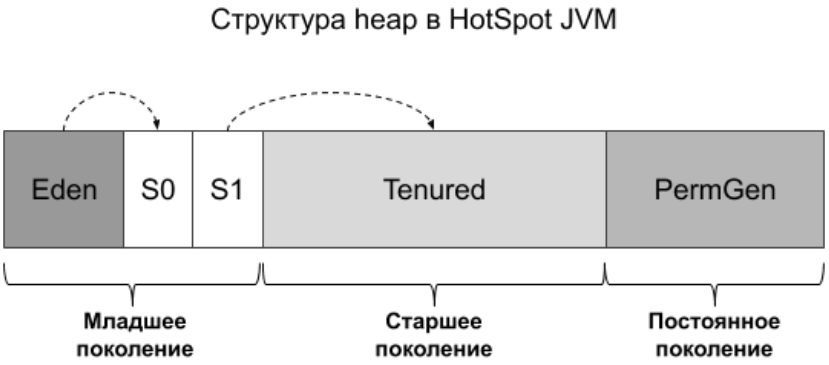

Рис. 2.2. Структура кучи

Очевидно, что промежуточная сборка занимает существенно меньше времени, чем полная. Поэтому все сборщики мусора нацелены на то, чтобы осуществлять промежуточную сборку вместо полной, если это возможно.

Кроме того, с целью уменьшения задержек некоторые сборщики мусора, нацеленные на минимизацию времени задержки, выполняют некоторые (но далеко не все) операции по сборке мусора параллельно с выполнением программы. Благодаря этому “stop the world” пауза, сопровождающая освобождение памяти, длится меньше времени.

4. Специальные виды ссылок (weak, soft, phantom)

В Java имеются классы, отвечающие за специальные виды ссылок, которые находятся в пакете `java.lang.ref` и позволяют влиять на сборку связанных с ними объектов. Эти классы предназначены для особых типов задач.

Рассмотрим эти классы подробнее:

- `WeakReference` — хранит так называемую ”слабую” ссылку на объект. Эта ссылка не препятствует сборке мусора, и в момент сборки мусора сборщик сначала освобождает все слабые ссылки и добавляет ссылку на объект в специальную очередь класса `ReferenceQueue` (если она была указана при инициализации слабой ссылки). При следующей сборке мусора объект будет стерт из памяти. Этот вид ссы-

лок подходит для решения проблемы утечки памяти из-за классов-слушателей (“listener”);

- `SoftReference` — хранит так называемую ”мягкую” ссылку на объект. Эти ссылки и объекты, на которые они ссылаются, удаляются при нехватке памяти. Это означает, что сборщик мусора гарантированно освободит память, связанную с мягкими ссылками, прежде чем будет выброшено исключение `OutOfMemoryError`. Этот вид ссылок подходит для реализации различных кешей данных, когда требуется решить проблему переполнения кеша данными;

- `PhantomReference` — хранит так называемую ”фантомную” ссылку на объект. Сборщик мусора не освобождает память, связанную с фантомными ссылками. Вместо этого он добавляет ссылку на объект в `ReferenceQueue` в случаях, когда объект доступен только через фантомную ссылку. Этот вид ссылок подходит для очень редких сценариев, когда требуется освободить ресурсы более гибким способом, чем стандартный механизм финализации или освобождение ресурсов ”вручную”.

На практике все эти классы используются довольно редко. Но, пожалуй, среди этих трех классов `PhantomReference` применяется реже всего.

Для демонстрации работы с подобными ссылками рассмотрим пример работы с мягкими ссылками. Для начала запустим следующую программу с ключом `-Xmx10m`[8]:

```
public class Main {

    public static void main(String[] args) {
        final int size = 170_000;

        List<String> numbers = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            numbers.add(String.valueOf(i));
        }
    }

}
```

[8]Этим ключом мы ограничиваем максимальный размер кучи значением 10 Мбайт.

При запуске программы она не выполнится успешно по причине недостатка памяти, и будет выброшено исключение `OutOfMemoryError`. Покажем, как эту ситуацию можно исправить при помощи слабых ссылок:

```
public class Main {

    public static void main(String[] args) {
        final int size = 170_000;

        List<SoftReference<String>> numRefs =
                new ArrayList<>();
        for (int i = 0; i < size; i++) {
            SoftReference<String> numRef =
                    new SoftReference<>(String.valueOf(i));
            numRefs.add(numRef);
        }

        // получим первый элемент
        SoftReference<String> firstRef =
                numRefs.get(0);
        System.out.printf("Первый элемент: %s\n",
                firstRef.get());
        // получим последний элемент
        SoftReference<String> lastRef =
                numRefs.get(size - 1);
        System.out.printf("Последний элемент: %s\n",
                lastRef.get());
    }

}
```

Эта программа успешно завершится и выведет в консоль следующее:

```
Первый элемент: null
Последний элемент: 169999
```

Приведенный пример со слабыми ссылками демонстрирует основную идею: при нехватке памяти в куче сборщик мусора выгрузил из памяти некоторые объекты (строки), на которые указывали мягкие ссылки.

5. Обзор стандартных утилит

В некоторых ситуациях, когда требуется анализ работы сборщика мусора HotSpot JVM, на помощь приходят утилиты из набора JDK и опции JVM, активируемые через флаги запуска.

В данном разделе мы бегло перечислим наиболее важные из них:

- флаг запуска JVM `-verbose:gc` — включает режим логирования сборок мусора в стандартный поток вывода;
- флаг запуска JVM `-Xloggc:filename` — указывает файл, в который должна логироваться информация о сборках мусора. Имеет более высокий приоритет, нежели флаг `-verbose:gc`;
- флаг запуска JVM `-XX:+PrintGCTimeStamps` — добавляет в информацию о сборках мусора временные метки;
- флаг запуска JVM `-XX:+PrintGCDetails` — включает расширенный режим логирования информации о сборках мусора;
- флаг запуска JVM `-XX:+PrintFlagsFinal` — при старте JVM выводит в стандартный поток вывода значения всех опций;
- утилита VisualVM — позволяет подключиться к запущенной локально JVM и визуализирует состояние виртуальной машины, в том числе с точки зрения памяти. Исполняемый файл `jvisualvm` находится в подпапке `bin` папки JDK;
- утилита jmap — позволяет подключиться к запущенной JVM и выводит статистику состояния памяти. В том числе умеет получать и сохранять в файл "снимок" текущего состояния кучи ("heap dump"), который в дальнейшем можно проанализировать стандартной утилитой jhat.

Предлагаем читателю самостоятельно более обстоятельно ознакомиться с этими опциями и утилитами, чтобы в дальнейшем иметь навыки нахождения причин некорректной работы программы, связанной с выделением памяти и сборкой мусора.

Глава 3

Загрузка классов в Java

Механизм загрузки классов уже неоднократно упоминался. В этой главе мы поговорим об этом безусловно важном механизме виртуальной машины Java подробнее. Конечно, все особенности и возможности этого достаточно сложного механизма рассматриваться не будут, поскольку наша основная цель — познакомить читателя с Java на достаточном для понимания концепций и принципов этой платформы и языка уровне.

Мы уже знаем, что загрузка классов в Java всегда происходит динамически, в ленивом режиме. Интересным следствием является то, что программа не бывает полностью загружена до момента своего выполнения. Чтобы проиллюстрировать это, рассмотрим простой пример:

```
public class A {

    static {
        System.out.println("Класс A загружается");
    }

}

public class B {

    static {
        System.out.println("Класс B загружается");
    }

}

public class Main {
```

```
    public static void main(String[] args) {
        System.out.println("Старт программы");
        A a = new A(); // класс загрузится
        B b; // класс загружен не будет
        System.out.println("Конец программы");
    }

}
```

При запуске этой программы в консоль будет выведено следующее:

```
Старт программы
Класс A загружается
Конец программы
```

Обратите внимание на тот факт, что класс `A` был загружен в процессе выполнения метода `main()`, в момент обращения к его конструктору. При этом класс `B` вовсе не был загружен, поскольку в программе не было обращений к его статическим полям, методам и к конструктору. Таким образом, JVM откладывает загрузку класса до момента, когда к информации о классе действительно нужно обратиться.

Будучи один раз загруженным, класс[9] хранится в памяти программы и используется при дальнейших обращениях к нему. Это означает, что загрузка класса происходит один раз, а затем используются закешированные данные.

Поговорим теперь про загрузчики классов, а затем рассмотрим подробно, что происходит в процессе загрузки класса.

1. Загрузчики классов

Загрузчик классов — это модуль JRE, отвечающий в первую очередь за получение байт-кода класса. При этом сам байт-код может быть получен из различных источников, таких как диск, сеть и даже может быть сгенерирован динамически.

Каждый из загрузчиков является объектом класса, унаследованного от класса `java.lang.ClassLoader`[10]. Этот абстрактный класс содержит

[9] А точнее, метаданные класса, т. е. объект `Class`.

[10] Исключением является только базовый загрузчик, но об этом мы поговорим позже.

множество различных методов, задающих поведение загрузчиков. Рассмотрим наиболее показательные методы этого класса:

- `Class loadClass(String name)` — публичный метод, который ожидает на вход полное имя класса, включая пакет, и возвращает объект `Class` для указанного класса. Этот метод вызывается JVM при необходимости загрузки класса;
- `Class<?> findClass(String name)` — этот `protected` метод вызывается из метода `loadClass()` после проверки наличия класса в родительском загрузчике. Он должен возвращать объект `Class` по имени класса. Реализация этого метода по умолчанию выбрасывает исключение и должна быть переопределена в классе-наследнике;
- `Class<?> defineClass(String name, byte[] b, int off, int len)` — `protected final` метод, который обрабатывает массив байтов, содержащий байт-код класса, и формирует из него объект `Class` средствами JVM. Этот метод обычно используется внутри переопределенного метода `findClass()` для преобразования загруженного байт-кода в объект `Class`;
- `ClassLoader getParent()` — публичный метод, который возвращает родительский загрузчик. Родительский загрузчик нужен для построения цепочки делегирования;
- `InputStream getResourceAsStream(String name)` — публичный метод, который позволяет находить файлы с различными ресурсами (в основном используется для поиска конфигурационных файлов в `XML` и `properties` форматах) при помощи механизма делегирования загрузчиков, независимо от текущего расположения файлов с байт-кодом. Возвращает входной поток для чтения ресурса. Кроме этого метода, в классе `ClassLoader` существует несколько других методов для работы с ресурсами, однако мы не будем рассматривать эту функцию загрузчиков классов в рамках данной главы и предлагаем читателю самостоятельно ознакомиться с ней.

Загрузчики классов связаны между собой через цепочку делегирования, т. е. у каждого загрузчика есть родительский загрузчик. При старте любой Java-программы создается три основных загрузчика классов, представляющие собой начало цепочки загрузчиков:

- базовый загрузчик (“bootstrap” или “primordial”) — это доверенный загрузчик JRE. Он реализован на уровне кода самой JVM, поэтому не представлен Java-классом. Этот загрузчик загружает все системные классы, например, `java.lang.Object`;
- загрузчик расширений (“extension”) — это загрузчик, отвечающий за загрузку расширений стандартной библиотеки, таких как платформа для создания приложений с графическим интерфейсом JavaFX или JavaScript-движок Nashorn. Его родительским загрузчиком[11] является базовый загрузчик;
- системный загрузчик, или загрузчик приложений (“system”, или “application”) — этот объект загружает файлы с байт-кодом классов приложения из `CLASSPATH`. Его родительским загрузчиком является загрузчик расширений.

Продемонстрируем работу всех этих трех загрузчиков на простом примере:

```
public class Main {

    public static void main(String[] args) {
        System.out.println("Базовый загрузчик: "
                + ClassLoader.class.getClassLoader());

        ClassLoader extClassLoader =
                com.sun.nio.zipfs.
                        ZipFileSystem.class
                        .getClassLoader();
        System.out.println("Загрузчик расширений: "
                + extClassLoader);
        System.out.println("Его родитель: "
                + extClassLoader.getParent());

        ClassLoader appClassLoader =
                Main.class.getClassLoader();
        System.out.println("Системный загрузчик: "
                + appClassLoader);
```

[11]Заметим, что слово ”родительский” здесь означает связь между объектами загрузчиков, а не цепочку наследования их классов.

```
        System.out.println("Его родитель: "
                + appClassLoader.getParent());
    }

}
```

После запуска эта программа выведет в консоль примерно следующее:

```
Базовый загрузчик: null
Загрузчик расширений: sun.misc.Launcher$ExtClassLoader@45ee12a7
Его родитель: null
Системный загрузчик: sun.misc.Launcher$AppClassLoader@18b4aac2
Его родитель: sun.misc.Launcher$ExtClassLoader@45ee12a7
```

Отметим, что поскольку базовый загрузчик не представлен классом, при попытке его получить возвращается значение `null`.

2. Порядок загрузки классов

Как мы уже отмечали, загрузчики классов — это часть JRE. В тот момент, когда JVM запрашивает определение класса, загрузчик классов пытается найти класс по заданному полному имени класса и затем загрузить его байт-код. При использовании конфигурации по умолчанию этим загрузчиком является системный загрузчик. При этом JVM вызывает у загрузчика знакомый нам метод `loadClass()`. Рассмотрим логику работы этого метода подробней.

В случае когда класс еще не был загружен, запрос на загрузку класса делегируется родительскому загрузчику классов. Этот процесс происходит рекурсивно, пока не дойдет до базового загрузчика. В этом процессе делегирования заключается смысл наличия цепочки загрузчиков классов. Такой подход позволяет загружать классы загрузчиком, максимально близко находящимся к базовому.

В случае когда родительский загрузчик (и все его родительские загрузчики соответственно) не может найти запрошенный класс, дочерний класс должен осуществить попытку поиска и загрузки класса самостоятельно. В случае если ни один из загрузчиков не обнаружит класс, будет выброшено исключение `ClassNotFoundException`.

В качестве иллюстрации логики загрузки класса рассмотрим ситуацию, когда происходит загрузка пользовательского класса `Dog`. При этом

выполняются следующие шаги:

1) системный загрузчик осуществляет поиск класса `Dog` среди уже загруженных им классов;

1.1) если класс найден, загрузка завершается;

1.2) если класс не найден, загрузка делегируется загрузчику расширений (шаг 2);

2) загрузчик расширений осуществляет поиск класса `Dog` среди уже загруженных им классов;

2.1) если класс найден, загрузка завершается;

2.2) если класс не найден, загрузка делегируется базовому загрузчику (шаг 3);

3) базовый загрузчик осуществляет поиск класса `Dog` среди уже загруженных им классов;

3.1) если класс найден, загрузка завершается;

3.2) если класс не найден, базовый загрузчик пытается его загрузить (шаг 4);

4) базовый загрузчик пытается загрузить класс `Dog`;

4.1) если загрузка прошла успешно, загрузка завершается;

4.2) если загрузку осуществить не удалось, загрузка передается загрузчику расширений (шаг 5);

5) загрузчик расширений пытается загрузить класс `Dog`;

5.1) если загрузка прошла успешно, загрузка завершается;

5.2) если загрузку осуществить не удалось, загрузка передается системному загрузчику (шаг 6);

6) системный загрузчик пытается загрузить класс `Dog`;

6.1) если загрузка прошла успешно, загрузка завершается;

6.2) если загрузку осуществить не удалось, выбрасывается исключение `ClassNotFoundException`.

Каждый загрузчик ведет учет классов, загруженных им. Это множество классов называют областью видимости (“namespace”). Один и тот же класс может войти в область видимости конкретного загрузчика только однократно. Области видимости в первую очередь используются в целях обеспечения безопасности выполняемого кода. Благодаря этой особенности и модели делегирования загрузчиков код злоумышленника, оказавшийся в одной из библиотек, используемых в вашей программе, не сможет подменить системные классы, такие как `java.lang.Object`.

Заметим также, что если по каким-то причинам (очевидно, выходящим за рамки поведения по умолчанию) один и тот же класс был загружен двумя разными загрузчиками, то с точки зрения JVM эти две копии считается двумя разными классами, поскольку эти загрузчики классов ассоциированы с двумя разными областями видимости. Другими словами, в качестве идентификатора класса с точки зрения JVM используется пара ”загрузчик (или область видимости) — класс”.

3. Явная и неявная загрузка классов

Как можно было заметить, классы в Java могут загружаться двумя способами — явным или неявным.

Явная загрузка класса происходит в случае, когда класс загружается посредством вызова одного из следующих методов:

- метод `loadClass()` загрузчика классов;
- статический метод `forName()` класса `Class`. В этом случае загрузку класса начинает загрузчик, ответственный за загрузку текущего класса, в котором осуществлен вызов данного метода.

Неявная загрузка классов происходит во всех ситуациях, когда какой-либо класс используется в коде, иными словами, при использовании класса: в качестве типа переменных и полей, при конструировании объектов, наследовании и т. д. Как и в случае явной загрузки, если класс уже был загружен, загрузчик вернет ссылку на него из своего кеша.

Заметим также, что цепочка классов может загружаться посредством комбинированного способа загрузки. Например, первый класс в цепочке может быть загружен явным образом, а классы, на которые он ссылается, — уже неявным способом.

4. Пример нестандартного загрузчика

Нестандартные загрузчики классов могут понадобиться в следующих случаях:

- при необходимости загрузки классов из специальных источников, например, из сети;
- при необходимости изолирования пользовательского кода. Этот прием в основном используется на различных серверах приложений (“application server”) и контейнерах сервлетов (“servlet container”);
- при необходимости выгрузки загруженных классов. Это может понадобиться в случаях, когда большое количество классов используется ограниченный период времени, а затем эти классы не используются. Поскольку загрузчики классов кешируют загруженные в их область видимости классы, эти загруженные классы не будут выгружены из памяти до тех пор, пока приложение содержит ссылку на сам загрузчик. Нестандартные системные загрузчики могут быть использованы для этих целей, так как они сами, а также загруженные ими классы, могут быть выгружены из памяти.

В этом разделе рассмотрим пример определения приложением собственного загрузчика классов. Этот загрузчик классов будет дочерним по отношению к системному загрузчику. В качестве его основной роли возьмем искусственный пример с загрузкой существующего класса при попытке загрузить несуществующий класс. Заметим, что вместо такого поведения, удобного нам в качестве демонстрации, наш загрузчик мог бы читать `.class` файлы из какого-либо источника, например, из директории или из сети, или даже генерировать байт-код ”на лету”.

Ниже приведен код нашего примера:

```
public class CustomClassLoader extends ClassLoader {

    public CustomClassLoader(ClassLoader parent) {
        super(parent);
    }

    @Override
    public Class<?> loadClass(String name)
```

```
        throws ClassNotFoundException {
    System.out.printf("Начинается загрузка класса %s\n",
            name);
    return super.loadClass(name);
}

@Override
public Class findClass(String name)
        throws ClassNotFoundException {
    System.out.printf("Начинается поиск класса %s\n",
            name);
    // в случае имени класса "HiddenA",
    // заменяем его на "A"
    if (name.equals("HiddenA")) {
        name = "A";
    }
    byte[] b = loadClassFromFile(name);
    return defineClass(name, b, 0, b.length);
}

private byte[] loadClassFromFile(String fileName)
        throws ClassNotFoundException {
    ByteArrayOutputStream byteStream =
            new ByteArrayOutputStream();
    int nextValue;
    try (InputStream inputStream = getClass()
            .getClassLoader()
            .getResourceAsStream(fileName + ".class")) {
        while ((nextValue = inputStream.read()) != -1) {
            byteStream.write(nextValue);
        }
    } catch (IOException e) {
        System.out.printf("Класс %s не загружен\n",
                fileName);
        throw new ClassNotFoundException();
    }
    return byteStream.toByteArray();
```

```
    }

}

public class Dog {
}

public class Main {

    public static void main(String[] args) {
        CustomClassLoader loader = new CustomClassLoader(
                Main.class.getClassLoader()
        );
        Class clazz;
        try {
            // загрузка "специального" класса
            clazz = loader.loadClass("HiddenDog");
            System.out.printf("0: класс %s загружен\n",
                    clazz);

            // загрузка обычного класса
            clazz = loader.loadClass("Dog");
            System.out.printf("1: класс %s загружен\n",
                    clazz);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

}
```

При запуске этот пример выведет в консоль следующее:

```
Начинается загрузка класса HiddenDog
Начинается поиск класса HiddenDog
Начинается загрузка класса java.lang.Object
0: класс class Dog загружен
Начинается загрузка класса Dog
1: класс class Dog загружен
```

Класс `CustomClassLoader` использует модели делегирования загрузки классов, поэтому он не изменяет поведение метода `loadClass()`, а вместо этого переопределяет метод `findClass()`. Из этого правила следует наблюдаемое нами поведение программы. Нашему загрузчику поручается только поиск и загрузка несуществующего класса `HiddenDog`, в то время как существующий класс `Dog` загружается системным загрузчиком.

Отметим также, что JVM предоставляет возможность переопределить системный загрузчик при помощи опции `-Djava.system.class.loader`, указав в качестве значения полное имя класса для нестандартного загрузчика. Кроме этой опции, существует еще ряд опций, важных с точки зрения изменения настроек загрузки классов и диагностики нештатных ситуаций. Мы не будем подробно рассматривать эти опции в данной главе и рекомендуем читателям ознакомиться с соответствующей документацией.

Напоследок еще раз подчеркнем, что в подавляющем большинстве программ изменять штатное поведение загрузчиков классов не требуется и не рекомендуется.

Глава 4

Многопоточное программирование

Вместе с бурным ростом производительности, а затем и количества ядер компьютеров, произошедшим за последние несколько десятилетий, проблема оптимального использования программами доступных вычислительных ресурсов стала актуальной, как никогда. За счет оптимальной загрузки вычислительных ядер, посредством распараллеливания задач, можно сократить время сложных вычислений в несколько раз или, например, увеличить количество запросов, обрабатываемых сетевым приложением за единицу времени. Для этих целей Java предлагает возможности кросс-платформенного многопоточного программирования.

Но прежде чем перейти к основному содержанию главы, поговорим о параллелизме и конкурентности, двух важных понятиях, связанных с многопоточным програмированием.

Параллелизм (“parallelism”) — это *свойство* систем, при котором вычисления выполняются одновременно и при этом, возможно, взаимодействуют друг с другом. Подчеркнем, что параллелизм подразумевает одновременное (параллельное) выполнение вычислений.

Конкурентность (“concurrency”) — это *возможность* построения программ из неких независимых блоков (единиц вычислений), способных выполняться в произвольном или частично определенном порядке, без влияния на конечный результат. Подчеркнем, что конкурентность допускает (но не требует) пареллельное выполнение некоторых вычислений. С точки зрения программирования конкурентность определяется некой моделью. Примерами таких моделей могут быть многопоточное программирование, модель взаимодействующих последовательных процессов (“communicating sequential processes (CSP)”), модель акторов (“actors”) и другие модели.

В русскоязычной технической литературе эти термины нередко путают друг с другом. Действительно, параллелизм и конкурентность являют-

ся смежными понятиями и подразумевают возможность одновременного выполнения вычислений. Однако конкурентность является парадигмой проектирования программ и систем, т. е. в некотором смысле более общим, логическим понятием, поскольку конкурентные программы и системы обеспечивают параллелизм при условии поддержки параллелизма на физическом уровне, например, при наличии у процессора нескольких вычислительных ядер. При этом возможна и конкурентность без параллелизма, например, многозадачность на одноядерном процессоре за счет разделения времени ("time-sharing") между задачами.

В данной главе мы сосредоточимся на модели многопоточного программирования и поговорим о возможностях многопоточного программирования в Java. Безусловно, написание производительных и, что еще более важно, надежных и предсказуемых многопоточных программ — это весьма сложная и обширная тема. В рамках этого курса мы ставим себе задачу познакомить читателя с базовыми понятиями, сформировав "фундамент" для дальнейшего самостоятельного изучения темы. Начнем с базовых для многопоточных программ понятий — процесс и поток.

1. Процессы

Процесс ("process") — это экземпляр программы, выполняемый в текущий момент в рамках операционной системы (ОС). В то время как компьютерная программа — это статический набор машинных инструкций, процесс подразумевает выполнение этих инструкций. Поэтому процесс содержит программный код и его состояние. Кроме того, с одной и той же программой могут быть ассоциированы несколько процессов. Например, при запуске программы несколько раз ОС стартует и инициирует несколько процессов.

Благодаря операционной системе процессы изолированы друг от друга, т. е. прямой доступ к памяти другого процесса невозможен. Это достигается за счет виртуального адресного пространства, индивидуально выделяемого каждому процессу. В момент запуска программы операционная система создает процесс, выделяет память и загружает в адресное пространство код и данные программы, затем запускает главный поток нового процесса. При необходимости ОС может изменить размер памяти, выделенной процессу.

Процессы всех программ выполняются в операционной системе в рамках многозадачности, т. е. конкурентно. Все они конкурируют за ресурсы компьютера: вычислительные ядра, память, устройства ввода/вывода. В современных ОС многозадачность, как правило, реализуется посредством разделения времени между задачами, заключающегося в выделении определенной квоты на выполнение очередного процесса, после исчерпания которой происходит переключение контекста, и процессор или его ядро начинает выполнять следующий процесс. За счет такой конкурентной многозадачности у пользователей ОС возникает ощущение, что даже на процессоре с одним ядром процессы выполняются одновременно. Иногда такой принцип выполнения вычислительных задач называют "псевдопараллелизмом".

Каждый процесс с точки зрения операционной системы характеризуется следующими ресурсами:

- образ машинных инструкций, соответствующий программе, иначе говоря, машинный код программы;
- память, выделенная процессу (виртуальное адресное пространство). Содержит машинный код, данные процесса (например, стандартные потоки ввода/вывода), стек вызовов ("call stack") и кучу ("heap");
- дескрипторы ресурсов, выделенных процессу. Примером таких ресурсов в Linux являются файловые дескрипторы, т. е. абстрактные идентификаторы, используемые процессом для доступа к файлу или иному ресурсу ввода/вывода, такому как сетевой сокет;
- атрибуты безопасности, например, идентификатор пользователя, являющегося владельцем процесса, и набор операций, разрешенных процессу;
- контекст процесса. Включает в себя содержимое пользовательского адресного пространства, аппаратных регистров процессора и структуры данных ядра, связанные с этим процессом.

Операционная система отслеживает ресурсы процессов, обеспечивая их изолированность и, при необходимости, выделяя дополнительные ресурсы. Кроме того, ОС также предоставляет средства межпроцессного взаимодействия ("inter-process communication" или "IPC"), позволяющие процессам обмениваться данными.

Отметим, что в большинстве операционных систем процесс может содержать в себе несколько потоков, в рамках которых независимо выполняются инструкции. Платформа Java подразумевает поддержку именно таких ОС.

2. Потоки

Поток (“thread”) — это минимальная последовательность программных инструкций, выполняемая операционной системой при помощи планировщика процессов (“process scheduler”). Любой процесс состоит как минимум из одного потока, который часто называют главным. Процесс, т. е. программа, будучи запущенным, может создавать дополнительные потоки.

В отличие от процессов, все потоки одного процесса имеют совместный доступ к ресурсам процесса, в первую очередь, к памяти, т. е. они выполняются в одном адресном пространстве. Каждому потоку при его создании операционной системой выделяется отдельная область в памяти, в которой содержится стек вызовов потока. Его зачастую называют просто стеком потока (“thread stack”). В Java стек потока используется виртуальной машиной для хранения параметров, локальных переменных, результатов промежуточных вычислений и других данных.

Как и в случае с процессами, операционная система конкурентно выполняет потоки текущего процесса. Кроме того, во многих программах потоки конкурируют за получение доступа к определенным участкам памяти для чтения и/или записи, например, для получения текущего значения и последующего увеличения какого-либо счетчика. Обеспечение детерменированного и безошибочного доступа к оперативной памяти из нескольких потоков является основной сложностью многопоточного программирования. Для корректной работы многопоточной программы используют механизмы синхронизации памяти, которые рассмотрим в следующих параграфах.

3. Модель памяти Java

Прежде чем перейти к рассмотрению возможностей многопоточного программирования, поговорим об основной спецификации, гарантирующей корректность и безопасность многопоточных программ в Java.

Модель памяти Java (“Java Memory Model” или “JMM”) — это спецификация, описывающая взаимодействие потоков с памятью в рамках JVM. Помимо освещения особенностей работы программы в рамках одного потока, модель памяти задает семантику языка Java при работе с памятью. Эта семантика описывает гарантии, на которые должен рассчитывать Java-программист, и то, на что рассчитывать не стоит.

Оригинальная модель памяти Java, разработанная в 1995 году, являлась одной из первых попыток описания кросс-платформенной модели памяти и оказалась не самой удачной по ряду причин. В 2004 году, в рамках Java 5, ее существенно переработали. Мы не будем описывать все особенности данной спецификации, однако не упомянуть ее в данной главе нельзя. Любознательному читателю предлагаем ознакомиться с обзорной веб-страницей, отвечающей на основные вопросы про JMM: `http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html`. Кроме того, в качестве дополнительного чтения рекомендуем познакомиться с главой ”Threads and Locks” спецификации ”Java Language Specification, Java SE 8”.

Перейдем, наконец, к обсуждению основного синтаксиса работы с потоками и синхронизации памяти.

4. Запуск потоков

Как мы уже знаем, у каждого процесса есть как минимум один поток, называемый главным (“main thread”). Помимо автоматически создаваемого главного потока, программа может запускать дополнительные потоки. Все потоки в Java представлены классом `java.lang.Thread`, реализующим интерфейс `java.lang.Runnable`. Объекты класса `Thread` являются абстракцией или объектным представлением системных потоков на уровне операционной системы.

Вначале обсудим интерфейс `Runnable`. Этот интерфейс служит для реализации классами, которые предназначены для выполнения в отдельном потоке. Он крайне прост и выглядит следующим образом:

```
public interface Runnable {

    void run();

}
```

Запустить поток в Java можно двумя способами. Первый способ предполагает создание объекта класса `Thread` через вызов конструктора, принимающего на вход реализацию интерфейса `Runnable`. Затем у объекта `Thread` потребуется вызвать метод `start()`. После вызова этого метода у созданного объекта-потока главный поток продолжает свое выполнение, т. е. они оба продолжают работу параллельно. Приведем простой пример, демонстрирующий этот способ:

```
public class Main {

    public static void main(String[] args) {
        System.out.println("Главный поток: "
                + Thread.currentThread());
        // определение потока с Runnable
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("Поток запущен: "
                        + Thread.currentThread());
            }
        });
        // теперь запускаем поток
        thread.start();
    }

}
```

Данная программа выведет в консоль примерно следующее:

```
Главный поток: Thread[main,5,main]
Поток запущен: Thread[Thread-0,5,main]
```

В этом примере для реализации интерфейса `Runnable` используется анонимный класс, но, конечно, ничто не мешает определить реализацию в отдельном классе. Кроме того, данный пример демонстрирует использование статического метода `currentThread()`, доступного в классе `Thread`. Этот метод возвращает объект `Thread`, соответствующий текущему потоку выполнения, из которого он вызван. Поэтому в примере при выводе в консоль мы видим главный поток, запущенный при старте программы, и дополнительный поток, который стартовали мы сами.

Опишем теперь второй способ запуска потока. Он заключается в создании потомка класса `Thread` и переопределении метода `run()`. Демонстрация этого способа, аналогичная предыдущему примеру по логике выполнения, выглядит так:

```
public class Main {

    public static void main(String[] args) {
        System.out.println("Главный поток: "
                + Thread.currentThread());
        // определение потока без Runnable
        Thread thread = new Thread() {
            @Override
            public void run() {
                System.out.println("Поток запущен: "
                        + Thread.currentThread());
            }
        };
        // теперь запускаем поток
        thread.start();
    }

}
```

Снова обратим внимание на использование анонимного класса, примененное из соображений лаконичности.

Заметим, что аналогичным образом потоки можно запускать не только из главного, но и из любого другого потока. Например, в наших примерах мы легко могли бы запустить еще один поток из нашего дополнительного потока.

Обсудим теперь важные особенности поведения потоков. Дополнительный поток будет завершен, как только метод `run()` завершит свое выполнение. Запустить один и тот же поток (объект `Thread`) несколько раз параллельно или последовательно невозможно, поскольку у каждого потока есть определенный жизненный цикл. Java отслеживает текущее состояние потока, значения которого задаются `enum State` (о нем поговорим далее), и при попытке повторного запуска выбрасывает исключение `java.lang.IllegalThreadStateException`.

5. Методы получения информации о потоке

Класс `Thread` содержит ряд методов, необходимых для осуществления действий с самим потоком и взаимодействия данного потока с другими. С некоторыми методами, такими как `start()` и `currentThread()`, мы уже познакомились. В данном параграфе рассмотрим методы, связанные с получением информации о потоке, а также заданием его поведения.

Рассмотрим некоторые соответствующие методы класса `Thread`:

- `static Thread currentThread()` — возвращает текущий поток;
- `String getName()` — возвращает имя потока;
- `void setName(String name)` — задает имя потока. Альтернативно имя потока можно задать через один из конструкторов класса. Если имя потока не было задано, оно будет сформировано автоматически по шаблону вида `Thread-<порядковый-номер>`;
- `long getId()` — возвращает идентификатор потока. Это положительное число, генерируемое при конструировании потока. Идентификатор является уникальным в рамках потока и остается неизменным до момента завершения потока. Ранее использованные значения идентификатора, соответствующие завершенным потокам, могут быть повторно использованы для новых потоков;
- `void setPriority(int newPriority)` — задает приоритет потока. В случае когда поток относится к группе потоков[12], имеющей ограничение на максимальное значение приоритета, будет выбрано наименьшее из двух значений — `newPriority` и максимального значения для группы. В случае если текущий поток, в рамках которого вызван метод, не имеет прав для изменения приоритета данного потока, будет выброшено исключение `java.lang.SecurityException`;
- `int getPriority()` — возвращает приоритет потока. Любой поток имеет приоритет. Потоки с более высоким приоритетом выбираются

[12]Группы потоков представлены классом `java.lang.ThreadGroup`. Каждая группа потоков задает набор потоков и позволяет изменять общие характеристики для всех потоков, входящих в группу. При этом группы могут быть связаны иерархией. Поток, находящийся в группе, может получить информацию о своей группе и влиять только на потоки своей группы, но к ее родительской группе у него уже не будет доступа. Таким образом, группы потоков используются для управления набором потоков и обеспечения безопасности доступа. Мы не будем подробно рассматривать эту возможность стандартной библиотеки и предлагаем читателю изучить ее самостоятельно.

планировщиком для выполнения с большим предпочтением, нежели потоки с меньшим приоритетом. По умолчанию приоритет потока равен приоритету потока, в рамках которого он был создан;

- `StackTraceElement[] getStackTrace()` — возвращает массив, содержащий снимок стека вызовов потока. Если поток еще не запущен, то массив будет пустым. В случае если метод вызван в потоке, отличном от данного, будут выполнены проверки безопасности доступа;

- `State getState()` — возвращает состояние потока. Данный метод предназначен для мониторинга состояния потоков, а не для их синхронизации. Состояние описывается `enum State` со следующими возможными значениями: `NEW` — поток еще не был запущен; `RUNNABLE` — поток запущен; `BLOCKED` — поток ожидает освобождения монитора[13]; `WAITING` — поток ожидает какого-то действия от другого потока (например, у потока был вызван метод `join()`); `TIMED_WAITING` — поток ожидает какого-то действия от другого потока, и при этом задано ограничение по времени ожидания; `TERMINATED` — поток завершил выполнение.

Рассмотрим пример использования некоторых из этих методов для получения информации о потоке:

```
public class Main {

    public static void main(String[] args) {
        System.out.println("Главный поток");
        printInfo(Thread.currentThread());

        // определение доп-го потока
        Thread thread = new Thread() {
            @Override
            public void run() {
                System.out.println("Доп-й поток: внутри");
                printInfo(Thread.currentThread());
            }
        };
        System.out.println("Доп-й поток: до");
```

[13] О синхронизации потоков, и в частности о блокировках и мониторах, поговорим в параграфе 10.

```
        printInfo(thread);

        // изменяем и стартуем поток
        thread.setName("MyThread");
        thread.setPriority(Thread.MAX_PRIORITY);
        thread.start();

        try {
            // дожидаемся завершения работы доп-го потока
            thread.join();
            System.out.println("Доп-й поток: после");
            printInfo(thread);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private static void printInfo(Thread thread) {
        System.out.printf(
                "Имя: %s, id: %d, " +
                        "приоритет: %d, состояние: %s\n",
                thread.getName(),
                thread.getId(),
                thread.getPriority(),
                thread.getState()
        );
    }

}
```

После запуска данная программа выведет в консоль следующее:

```
Главный поток
Имя: main, id: 1, приоритет: 5, состояние: RUNNABLE
Доп-й поток: до
Имя: Thread-0, id: 11, приоритет: 5, состояние: NEW
Доп-й поток: внутри
Имя: MyThread, id: 11, приоритет: 10, состояние: RUNNABLE
Доп-й поток: после
```

```
Имя: MyThread, id: 11, приоритет: 10, состояние: TERMINATED
```

Этот пример достаточно прямолинеен и наглядно демонстрирует получение базовой информации о потоке, а также изменение имени и приоритета потока. Отметим лишь использование метода `join()` для целей демонстрации статуса завершенного потока. Мы обсудим этот метод в следующих параграфах.

6. Потоки-демоны

Как мы уже знаем, каждый поток имеет определенное значение приоритета, которое системный планировщик учитывает при выборе потоков для исполнения. Кроме того, поток может быть отмечен как поток-демон ("daemon thread"). Этот признак учитывается при определении условий завершения работы JVM. Виртуальная машина продолжает выполнять потоки программы до наступления одного из следующих условий:

- один из потоков вызвал метод `exit()` класса `Runtime`. При этом менеджер безопасности ("security manager") — специальный класс, задающий политики безопасности — подтвердил права на вызов;
- все потоки, не являющиеся демонами, завершили выполнение.

Таким образом, потоки, отмеченные как демоны, не препятствуют завершению Java-программы. Эти потоки могут быть удобны для выполнения побочных, необязательных задач в отдельном потоке. Примером таких задач может быть отправка записей журнала приложения по сети или сбор и обработка информации, необходимой для мониторинга состояния приложения.

Перечислим методы класса `Thread`, связанные с потоками-демонами:

- `boolean isDaemon()` — возвращает признак того, что текущий поток является потоком-демоном. По умолчанию значение признака равно значению для потока, в рамках которого он был создан;
- `void setDaemon(boolean on)` — переопределяет значение признака того, что текущий поток является потоком-демоном. Должен быть вызван до запуска потока. В противном случае выбрасывает исключение `java.lang.IllegalThreadStateException`.

Еще раз отметим, что при создании новый поток по умолчанию получает такое же значение приоритета и признака потока-демона, как у текущего потока.

Рассмотрим пример использования потоков-демонов:

```
public class Main {

    public static void main(String[] args) {
        Thread thread = new Thread() {
            @Override
            public void run() {
                while (true) {
                  System.out.println("Доп-й поток: " +
                            "проводим вычисления");
                  // некие вычисления
                  int fib100 = fib(5);
                  System.out.printf("Доп-й поток: результат " +
                            "вычислений: %d\n", fib100);
                }
            }
        };
        System.out.println("Стартуем доп-й поток-демон");
        thread.setDaemon(true);
        thread.start();

        // некие вычисления
        // (чуть более длительные, чем в доп-м потоке)
        fib(15);

        System.out.println("Программа завершилась");
    }

    // вычисление чисел Фибоначчи через рекурсию
    // (имеет экспоненциальное время выполнения)
    private static int fib(int num) {
        if (num == 0) return 0;
        if (num == 1) return 1;
        return fib(num - 2) + fib(num - 1);
```

```
    }

}
```

Эта программа выведет в консоль примерно следующее (результат зависит от многих факторов и может варьироваться):

```
Стартуем доп-й поток-демон
Доп-й поток: проводим вычисления
Доп-й поток: результат вычислений: 5
Доп-й поток: проводим вычисления
Программа завершилась
Доп-й поток: результат вычислений: 5
Доп-й поток: проводим вычисления
Доп-й поток: результат вычислений: 5
Доп-й поток: проводим вычисления
```

В нашем примере был создан дополнительный поток-демон, в котором происходят некие условно бесконечные вычисления. Пример демонстрирует то, что выполнение потока-демона не препятствует завершению программы в тот момент, когда все обычные потоки (в нашем примере это только главный поток) завершают выполнение.

7. Методы управления потоком

Рассмотрим теперь методы управления потоком. Эти методы позволяют запускать, временно приостанавливать и при определенных условиях даже полностью останавливать выполнение потока. Большинство из этих методов, например, метод `sleep()`, рассчитаны на вызов внутри кода потока, т. е. в теле метода `run()`. Некоторые методы, такие как `start()` или `interrupt()`, рассчитаны на вызов в рамках другого потока.

Перечислим основные методы, позволяющие управлять потоком:

- `void start()` — запускает выполнение потока. В случае попытки повторного запуска этого же потока будет выброшено исключение `java.lang.IllegalThreadStateException`;
- `static void sleep(long millis)` — приостанавливает выполнение потока на указанное время (в миллисекундах). При этом системный планировщик приостанавливает выполнение текущего потока и

переходит к выполнению других потоков или процессов. Этот метод и некоторые из перечисленных далее выбрасывают исключение `java.lang.InterruptedException` в случае, если поток был прерван ("interrupted"). Точность соответствия желаемого и фактического периода бездействия потока зависит от точности системного планировщика и механизма определения времени. О прерывании работы потоков поговорим далее;

- `void join(long millis)` — приводит к ожиданию завершения выполнения потока в течение указанного времени. Время, указанное в миллисекундах, обозначает максимальное время ожидания. Если выполнение потока не завершится в течение обозначенного интервала, то текущий поток продолжит выполнение. Этот метод полезен для случаев, когда в одном потоке требуется дождаться завершения работы другого потока. Выбрасывает исключение `InterruptedException` в случае, если поток был прерван ("interrupted");

- `void join()` — аналогичен предыдущему методу, однако приводит к неограниченному по времени ожиданию;

- `static void yield()` — дает системному планировщику подсказку о том, что текущий поток завершил значимую часть работы, и можно переключиться на выполнение какого-либо другого потока или процесса. Планировщик может проигнорировать эту подсказку и продолжить выполнение текущего потока. Данный метод полезен в тех случаях, когда требуется дать возможность выполниться другому потоку. Примером такой ситуации может быть выполнение вычислительно сложной задачи. Вычисления при этом можно разбить на части и вызывать метод `yield()` перед продолжением вычислений. Это позволит другим потокам или процессам выполняться периодически. Заметим, что это узкоспециализированный метод, поэтому его использование должно быть обоснованным и сопровождаться соответствующим тестированием и профилированием программы;

- `void interrupt()` — отправляет потоку "сигнал" прерывания. При этом в случае, когда прерываемый поток был приостановлен (заблокирован) вызовом таких методов, как `sleep()`, `join()` или вариации метода `wait()`, его выполнение будет прервано выбросом исключения `InterruptedException`. При попытке неразрешенного доступа к потоку выбрасывается исключение `java.lang.SecurityException`;

- `static boolean interrupted()` — возвращает признак того, что текущий поток был прерван. В отличие от метода `isInterrupted()` в результате вызова данного метода в объекте потока признак сбрасывается на значение `false`;
- `boolean isInterrupted()` — возвращает признак того, что текущий поток был прерван. Значение признака при этом не сбрасывается.

Эти и другие рассмотренные ранее методы класса `Thread` — далеко не полный список методов, доступных в данном классе. Но даже этот неполный список сложно запомнить и понять принцип работы только по описанию. Поэтому рассмотрим последовательно использование методов управления потоком на примерах.

7.1. Методы sleep() и join()

Начнем с рассмотрения примеров использования методов `sleep()` и `join()`. Метод `sleep()` приостанавливает выполнение потока на заданное время. Метод `join()` позволяет приостановить выполнение текущего потока до момента завершения выполнения заданного потока.

В нашем примере основной поток будет запускать дополнительный поток, в котором мы будем приостанавливать выполнение через вызов `sleep()`. После чего основной поток будет дожидаться завершения дополнительного потока при помощи вызова `join()`. Соответствующая программа выглядит следующим образом:

```
public class Main {

    public static void main(String[] args) {
        Thread thread = new Thread() {
            @Override
            public void run() {
                System.out.println("Доп-й поток: стартовал");
                // ждем 1 секунду
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
```

```
                System.out.println("Доп-й поток: завершился");
            }
        };
        System.out.println("Стартуем доп-й поток");
        thread.start();

        try {
            // дожидаемся завершения работы доп-го потока
            thread.join();
            System.out.println("Программа завершилась");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

}
```

Эта программа выведет в консоль следующее:

```
Стартуем доп-й поток
Доп-й поток: стартовал
Доп-й поток: завершился
Программа завершилась
```

7.2. Метод yield()

Рассмотрим теперь метод `yield()`. Как уже отмечалось, это узкоспециализированный метод, который дает системному планировщику подсказку о том, что текущий поток завершил значимую часть работы, и можно переключиться на выполнение какого-либо другого потока или процесса. Поэтому продемонстрировать его работу относительно сложно.

Рассмотрим программу, эмулирующую вычислительно сложную задачу, выполняемую в дополнительном потоке. При этом дополнительный поток будет периодически вызывать метод `yield()` для того, чтобы планировщик мог передать управление другому потоку или процессу. Заметим, что ощутимые отличия в работе этой программы без вызова `yield()` и с ним с большей вероятностью удастся увидеть на компьютерах, где программе доступно только одно ядро процессора.

Приведем код нашего примера:

```
public class Main {

    public static void main(String[] args) {
        Thread thread = new Thread() {
            @Override
            public void run() {
                System.out.println("Доп-й поток: стартовал");
                // делаем 10 итераций с вычислением
                // 40го числа Фибоначчи
                heavyCalc(10, 40);
                System.out.println("Доп-й поток: завершился");
            }
        };
        System.out.println("Стартуем доп-й поток");
        thread.start();

        // делаем такое же вычисление, что и в доп-м потоке
        heavyCalc(10, 40);
        System.out.println("Осн-й поток: завершился");
    }

    // вычисление чисел Фибоначчи через рекурсию
    // (имеет экспоненциальное время выполнения)
    private static int fib(int num) {
        if (num == 0) return 0;
        if (num == 1) return 1;
        return fib(num - 2) + fib(num - 1);
    }

    private static void heavyCalc(int iterations, int fibNum) {
        for (int it = 0; it < iterations; it++) {
            long time = System.currentTimeMillis();
            fib(fibNum);
            System.out.printf("Поток %s: вычисление " +
                              "заняло %d мс\n",
                    Thread.currentThread(),
                    (System.currentTimeMillis() - time));
```

```
            // предлагаем передать выполнение
            // другому потоку/процессу
            Thread.yield();
        }
    }
}
```

Эта программа выведет в консоль следующее (вывод немного сокращен):

```
Стартуем доп-й поток
Доп-й поток: стартовал
Поток Thread[Thread-0,5,main]: вычисление заняло 1037 мс
Поток Thread[main,5,main]: вычисление заняло 1063 мс
Поток Thread[Thread-0,5,main]: вычисление заняло 944 мс
Поток Thread[main,5,main]: вычисление заняло 945 мс
...
Доп-й поток: завершился
Поток Thread[main,5,main]: вычисление заняло 851 мс
Осн-й поток: завершился
```

В консольном выводе мы видим периодическое переключение между потоками. Подчеркнем еще раз, что `yield()` на практике используется редко и только при осмысленной и обоснованной необходимости. Однако не обсудить этот метод и его предназначение мы не могли.

7.3. Методы interrupt(), interrupted() и isInterrupted()

Рассмотрим методы `interrupt()`, `interrupted()` и `isInterrupted()`. Эти методы предназначены для прерывания работы запущенного потока. При этом ничто не мешает реализовать поток так, чтобы он игнорировал полученный сигнал о прерывании и продолжал работу[14]. Другими словами, для прерывания работы потока требуются согласованные действия со стороны обоих потоков, прерывающего и прерываемого. Такой подход к прерыванию работы потоков называют кооперативным. Помимо кооперативного способа, в Java отсутствуют другие способы принудительной остановки потока[15].

[14]Такая реализация считается плохим тоном. Потоки должны корректно обрабатывать сигналы прерывания работы.

[15]В первых версиях Java в классе `Thread` имелся метод `stop()`, принудительно останавливающий поток. Однако применение этого метода часто приводило к неопределенному состоянию данных в

Напомним, что метод `interrupt()` отправляет в поток сигнал прерывания, что приводит, во-первых, к присвоению значения `true` признаку того, что текущий поток был прерван, и во-вторых, к возможному выбросу исключения `InterruptedException` из ряда методов класса `Thread`. Методы `interrupted()` и `isInterrupted()` возвращают значение признака того, что текущий поток был прерван. При этом вызов метода `interrupted()` сбрасывает значение признака на `false`.

Для демонстрации этих методов рассмотрим пример, где основной поток будет запускать два дополнительных потока и пытаться завершить их работу. Код программы приведен ниже:

```
public class LooperThread extends Thread {

    @Override
    public void run() {
        while (true) {
            // проверка сигнала прерывания
            if (interrupted()) {
                System.out.println("Доп-й поток " +
                        "LooperThread: получен сигнал");
                break;
            }
            // здесь могли бы быть какие-то вычисления
        }
        System.out.println("Доп-й поток " +
                "LooperThread: завершился");
    }

}

public class SleeperThread extends Thread {

    @Override
    public void run() {
        while (true) {
            try {
```

памяти работающего приложения. Поэтому позднее метод `stop()` объявили устаревшим и добавили средства кооперативного прерывания работы потока.

```
                // ждем 10 сек
                Thread.sleep(10_000);
            } catch (InterruptedException e) {
                System.out.println("Доп-й поток " +
                        "SleeperThread: " +
                        "получен сигнал-исключение");
                break;
            }
            // проверка сигнала нужна
            if (interrupted()) {
                System.out.println("Доп-й поток " +
                        "SleeperThread: получен сигнал");
                break;
            }
        }
        System.out.println("Доп-й поток " +
                "SleeperThread: завершился");
    }

}

public class Main {

    public static void main(String[] args)
            throws InterruptedException {
        Thread lt = new LooperThread();
        lt.start();
        // немного ждем и пытаемся остановить поток
        Thread.sleep(100);
        lt.interrupt();
        lt.join();
        System.out.println("Осн-й поток: дождался " +
                "остановки первого доп-го потока");

        Thread st = new SleeperThread();
        st.start();
        // немного ждем и пытаемся остановить поток
```

```
        Thread.sleep(100);
        st.interrupt();
        st.join();
        System.out.println("Осн-й поток: дождался " +
                "остановки второго доп-го потока");

        System.out.println("Осн-й поток: завершился");
    }

}
```

Эта программа выведет в консоль следующее:

```
Доп-й поток LooperThread: получен сигнал
Доп-й поток LooperThread: завершился
Осн-й поток: дождался остановки первого доп-го потока
Доп-й поток SleeperThread: получен сигнал-исключение
Доп-й поток SleeperThread: завершился
Осн-й поток: дождался остановки второго доп-го потока
Осн-й поток: завершился
```

Отметим, что в методе `run()` класса `SleeperThread` обрабатываются как исключение `InterruptedException`, так и признак (или сигнал) прерывания работы потока, возвращаемый методом `interrupted()`. Но одной обработки исключения недостаточно, поскольку нет никаких гарантий, что выполнение потока будет находиться внутри метода `sleep()` или аналогичного метода, выбрасывающего `InterruptedException`, во время получения сигнала прерывания. Кроме того, при выбросе исключения `InterruptedException` признак прерывания, возвращаемый методами `interrupted()` и `isInterrupted()`, будет иметь значение `false`, поскольку в этом случае таким признаком является само выброшенное исключение.

8. Обработка исключений в дополнительных потоках

Обработка исключений в дополнительных потоках имеет свои особенности по сравнению с обработкой исключений в главном потоке. Ключевая особенность состоит в том, что в дополнительных потоках выброс

необработанных исключений, относящихся к непроверяемым исключениям, не приводит к завершению работы программы, в то время как в случае главного потока работа программы завершается.

Следующий пример демонстрирует эту особенность:

```
public class Main {

    public static void main(String[] args)
            throws InterruptedException {
        Thread thread = new Thread() {
            @Override
            public void run() {
                System.out.println("Доп-й поток: " +
                        "старт");
                throw new RuntimeException();
            }
        };
        System.out.println("Стартуем доп-й поток");
        thread.start();

        // ждем 3 секунды до момента завершения программы
        Thread.sleep(3000);
        System.out.println("Программа завершилась");
    }

}
```

Эта программа выведет в консоль следующее:

```
Стартуем доп-й поток
Доп-й поток: старт
Exception in thread "Thread-0" java.lang.RuntimeException
at Main$1.run(Main.java:10)
Программа завершилась
```

Из консольного вывода видно, что выброс необработанного исключения `RuntimeException` в дополнительном потоке привел только к записи информации (причем эта запись ушла в стандартный поток для ошибок). Программа при этом продолжила выполнение.

Конечно, для более контролируемого выполнения программы можно использовать блок `try/catch` на верхнем уровне метода `run()` потока. Однако в Java есть и более удобные способы описания обработчиков исключений для дополнительных потоков. Перечислим соответствующие методы класса `Thread`:

- `void setUncaughtExceptionHandler(UncaughtExceptionHandler eh)` — задает обработчик исключений, вызываемый при выбросе необработанных исключений в данном потоке. В случае когда обработчик для потока не задан, используется обработчик группы потоков (`ThreadGroup`), к которой относится данный поток, при ее наличии);
- `UncaughtExceptionHandler getUncaughtExceptionHandler()` — возвращает обработчик, заданный для данного потока. В случае отсутствия обработчика возвращается обработчик группы потоков;
- `static void setDefaultUncaughtExceptionHandler(UncaughtExceptionHandler eh)` — задает обработчик исключений по умолчанию. В случае выброса необработанного исключения сначала проверяется наличие обработчика уровня потока, затем — уровня группы потоков, после чего обработчика по умолчанию. Только первый найденный в этой цепочке обработчик будет вызван.

Эти методы используют интерфейс `UncaughtExceptionHandler`, являющийся вложенным интерфейсом класса `Thread`. Он содержит один метод и выглядит следующим образом:

```
public interface UncaughtExceptionHandler {

    void uncaughtException(Thread t, Throwable e);

}
```

Метод `uncaughtException()` обработчика вызывается перед завершением работы потока, в котором было выброшено необработанное исключение. При этом любые необработанные исключения, выброшенные в ходе выполнения самого метода `uncaughtException()`, будут проигнорированы. Отметим также, что класс `ThreadGroup` реализует интерфейс `UncaughtExceptionHandler` и, таким образом, сам по себе является обработчиком исключений.

Приведем пример использования разных видов обработчиков исключений:

```
public class Main {

    public static void main(String[] args)
            throws InterruptedException {
        Thread.setDefaultUncaughtExceptionHandler(
                new Handler("По умолчанию"));

        Thread t1 = createThread(1);
        t1.setUncaughtExceptionHandler(
                new Handler("Обычный"));
        System.out.printf("Стартуем доп-й поток 1: %s\n", t1);
        t1.start();

        Thread t2 = createThread(2);
        System.out.printf("Стартуем доп-й поток 2: %s\n", t2);
        t2.start();

        // ждем завершения обоих потоков
        t1.join();
        t2.join();
        System.out.println("Программа завершилась");
    }

    private static Thread createThread(int num) {
        return new Thread() {
            @Override
            public void run() {
                System.out.printf("Доп-й поток %d: " +
                        "старт\n", num);
                throw new RuntimeException();
            }
        };
    }

}
```

```
public class Handler
        implements Thread.UncaughtExceptionHandler {

    private final String name;

    public Handler(String name) {
        this.name = name;
    }

    @Override
    public void uncaughtException(Thread t,
                                  Throwable e) {
        System.out.printf("[%s] В потоке %s\n  " +
                "выброшено исключение %s\n", name, t, e);
    }

}
```

Этот пример наглядно показывает использование обработчиков исключений в дополнительных потоках. В результате работы он выведет в консоль следующее:

```
Стартуем доп-й поток 1: Thread[Thread-0,5,main]
Стартуем доп-й поток 2: Thread[Thread-1,5,main]
Доп-й поток 1: старт
Доп-й поток 2: старт
[Обычный] В потоке Thread[Thread-0,5,main]
  выброшено исключение java.lang.RuntimeException
[По умолчанию] В потоке Thread[Thread-1,5,main]
  выброшено исключение java.lang.RuntimeException
Программа завершилась
```

Исключение из потока `t2`, где не было без заданного обработчика исключений, было перехвачено обработчиком по умолчанию, тогда как исключение из потока `t1` попало в его собственный обработчик.

9. Пулы потоков

Пулы потоков (“thread pool”) — это объекты (и классы), предоставляющие абстракцию для определенного набора потоков и позволяющие отправлять вычислительные задачи на выполнение в одном из потоков (“worker thread”), входящих в пул. В случае когда все потоки пула оказываются заняты при отправлении очередной задачи на выполнение, эта задача добавляется во внутреннюю очередь пула и выполнится позднее, когда освободится один из потоков. Размер пула, в зависимости от конкретного класса и его конфигурации, может быть как фиксированным, так и изменяющимся динамически.

Пулы потоков имеют ряд преимуществ в сравнении с ”ручным” порождением и запуском потоков, выполняющих какую-то задачу. Во-первых, обычно пул потоков реализован так, чтобы он переиспользовал уже имеющиеся потоки, а не порождал новые перед выполнением каждой задачи. Это позволяет не тратить процессорное время и память на выделение ресурсов для потока. Пулы также позволяют отложить создание потоков до появления задач и указать время, после истечения которого неиспользуемые потоки будут уничтожены при условии отсутствия новых задач. Во-вторых, для пула потоков можно задать максимально возможное количество потоков. Это позволяет ограничить максимальный объем ресурсов, потребляемых программой. Например, для вычислительно сложных задач часто используют пул фиксированного размера, совпадающего с количеством ядер процессора. При написании программ всегда полезно продумывать поведение программы в случае роста размера входных данных и ограничивать максимальное потребляемое количество ресурсов, будь то количество потоков или используемая память.

В этом разделе мы в первую очередь хотим познакомить читателя с понятием ”пул потоков”. Поэтому мы не будем описывать все возможности стандартной библиотеки Java, связанные с пулами потоков, а лишь ограничимся малой их частью, необходимой для демонстрационных целей.

Рассмотрим использование пула потоков на примере библиотечного класса `java.util.concurrent.ThreadPoolExecutor`. Этот класс позволяет гибко конфигурировать размер пула, задавать правила создания и уничтожения потоков, входящих в пул, а также задавать очередь, в которую будут попадать задачи в случае, когда все потоки пула заняты. В качестве задач в этом пуле выступают экземпляры уже знакомого нам

интерфейса Runnable. Код примера представлен ниже:

```
public class Main {

    public static void main(String[] args)
            throws InterruptedException {
        // определяем кол-во ядер процессора
        int cores = Runtime.getRuntime().availableProcessors();
        // создаем очередь с максимальным размером 100 задач
        BlockingQueue<Runnable> queue =
                new LinkedBlockingQueue<>(100);
        // создаем пул
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
                // мин и макс размеры пула:
                cores, cores * 2,
                // время ожидания до уничтожения
                // неиспользуемых потоков
                60, TimeUnit.SECONDS,
                // указываем очередь
                queue,
                // указываем объект, который будет вызван,
                // если превышен размер очереди
                new RejectedHandler()
        );

        // по умолчанию пул будет создавать потоки
        // в "ленивом" режиме, но таким образом мы могли
        // бы "прогреть" пул до мин кол-ва потоков:
        // executor.prestartAllCoreThreads();

        System.out.printf("Начальный размер пула %d потоков\n",
                executor.getPoolSize());

        // добавляем в пул задачи
        int tasks = 100 + cores * 4;
        for (int i = 0; i < tasks; i++) {
            executor.execute(new FibRunnable(i));
        }
```

```
        // переключаем пул в режим завершения работы,
        // после чего новые задачи добавлять в него нельзя
        executor.shutdown();
        // дожидаемся завершения выполнения всех задач
        while (!executor.awaitTermination(10,
                TimeUnit.SECONDS)) {
            System.out.println("Ожидаем завершения");
        }
        System.out.println("Все задачи завершены");
    }

}

public class FibRunnable implements Runnable {

    private final int indx;

    public FibRunnable(int indx) {
        this.indx = indx;
    }

    @Override
    public void run() {
        System.out.printf("Стартуем задачу %d " +
                "в потоке %s\n", indx, Thread.currentThread());
        // вычисляем 40е число Фибоначчи
        fib(40);
        System.out.printf("Задача %d в потоке %s " +
                "завершилась\n", indx, Thread.currentThread());
    }

    // вычисление чисел Фибоначчи через рекурсию
    // (имеет экспоненциальное время выполнения)
    private int fib(int num) {
        if (num == 0) return 0;
        if (num == 1) return 1;
```

```
        return fib(num - 2) + fib(num - 1);
    }

    @Override
    public String toString() {
        return "Задача " + indx;
    }

}

public class RejectedHandler
        implements RejectedExecutionHandler {

    @Override
    public void rejectedExecution(Runnable r,
                                  ThreadPoolExecutor e) {
        System.out.printf("Задача \"%s\" отклонена\n", r);
    }
}
```

После запуска эта программа выведет в консоль примерно следующее (содержимое сокращено для упрощения восприятия):

```
Начальный размер пула 0 потоков
Стартуем задачу 1 в потоке Thread[pool-1-thread-2,5,main]
Стартуем задачу 4 в потоке Thread[pool-1-thread-5,5,main]
Стартуем задачу 0 в потоке Thread[pool-1-thread-1,5,main]
...
Стартуем задачу 109 в потоке Thread[pool-1-thread-10,5,main]
Стартуем задачу 2 в потоке Thread[pool-1-thread-3,5,main]
Задача "Задача 112" отклонена
...
Задача "Задача 123" отклонена
Стартуем задачу 111 в потоке Thread[pool-1-thread-12,5,main]
...
Задача 80 в потоке Thread[pool-1-thread-8,5,main] завершилась
Стартуем задачу 104 в потоке Thread[pool-1-thread-8,5,main]
Ожидаем завершения
...
```

```
Задача 105 в потоке Thread[pool-1-thread-7,5,main] завершилась
Задача 104 в потоке Thread[pool-1-thread-3,5,main] завершилась
Все задачи завершены
```

Обратите внимание, что последовательность запуска и завершения задач в пуле может не совпадать с порядком добавления задач (в примере задача №2 начинает выполняться после задачи №109). Это обусловлено заданным поведением пула. В нашем случае при добавлении новых задач пул создает дополнительные потоки в случае, если начальному количеству потоков, равному в этом примере количеству ядер процессора, уже присвоены задачи.

Пулы потоков — очень мощный инструмент многопоточного программирования. Они прекрасно подходят для задач, в которых требуется распараллелить вычисления и/или обработку данных, при этом ограничив максимально возможное потребление ресурсов компьютера. С другой стороны, в случаях когда вычислительные задачи в процессе выполнения зависят от данных друг друга, т. е. требуют синхронной обработки, пулы потоков могут оказаться не лучшим выбором, поскольку при неконтролируемом порядке выполнения задач может возникнуть риск блокировки.

10. Синхронизация потоков

Ранее мы рассматривали примеры многопоточных программ, где не требовалось одновременно читать и модифицировать одни и те же данные, находящиеся в памяти, из разных потоков. Однако, как только в программе появляется такая необходимость, оказывается, что для ее корректного поведения требуется использовать средства синхронизации, предоставляемые Java. Без использования средств синхронизации код, работающий корректно в рамках однопоточной программы, будет приводить к непредсказуемым результатам при использовании нескольких потоков, одновременно читающих и изменяющих одни и те же значения.

Практически все, что мы рассмотрим в дальнейших разделах, относится к так называемой пессимистичной или блокирующей синхронизации[16]. Именно ее в большинстве случаев подразумевают, когда говорят о синхронизации. Суть такой синхронизации заключается в том, что когда

[16]Отметим, что ключевое слово `volatile` не обеспечивает синхронизацию потоков, но об этом мы поговорим позже.

поток пытается получить управление каким-либо ресурсом, например, получить монитор, связанный с объектом, для входа в блок `synchronized`[17], поток блокируется, т. е. приостанавливает свое выполнение, до момента получения контроля над запрошенным ресурсом. Только один поток может захватить ресурс и выполнить определенную часть кода. Другими словами, использование пессимистичной синхронизации можно охарактеризовать фразой "все или ничего".

На уровне ОС инструменты синхронизации Java обеспечиваются известными примитивами, такими как мьютексы, семафоры и критические секции. Читатели, знакомые с этими примитивами, должны быстрее освоить дальнейший материал.

Поскольку блокирующая синхронизация подразумевает определенные накладные расходы, то логично, что традиционная проблема высокопроизводительных многопоточных программ — это сведение синхронизации до минимально необходимого уровня, позволяющего оптимально загрузить все доступные ресурсы компьютера.

Отметим также понятие оптимистичной или неблокирующей сихронизации. Под ней подразумевают синхронизацию, основанную на попытке изменения общего ресурса, т. е. значения, находящегося в памяти. Поток при этом не блокируется, и в случае если изменить значения не удалось, поток продолжает выполнение и имеет возможность осуществить повторную попытку изменения ресурса или же выполнить какие-то другие действия. Стандартная библиотека Java предоставляет некоторые средства для неблокирующей синхронизации, например, набор классов `Atomic*` из пакета `java.util.concurrent`. Но, поскольку оптимистичная синхронизация выходит за рамки данной главы, мы предлагаем читателю самостоятельно ознакомиться этой темой.

В качестве демонстрации необходимости наличия синхронизации рассмотрим простой пример с целочисленным счетчиком, значение которого изменяется из нескольких потоков:

```
public class Main {

    private static final int ITERATIONS = 10_000;
    private static int cnt;
```

[17]Блоки и методы, отмеченные ключевым словом `synchronized`, и их особенности мы обсудим немного позже.

```
    public static void main(String[] args)
            throws InterruptedException {
        for (int i = 0; i < ITERATIONS; i++) {
            cnt = 0;
            inc();
            if (cnt != 200) {
                System.out.printf("Ошибка. " +
                        "Счетчик оказался равен %d\n", cnt);
            }
        }

        System.out.println("Программа завершена");
    }

    public static void inc()
            throws InterruptedException {
        // оба потока увеличивают счетчик на 100
        Runnable incrementer = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 100; i++) {
                    cnt = cnt + 1;
                }
            }
        };
        Thread t1 = new Thread(incrementer);
        Thread t2 = new Thread(incrementer);
        // стартуем оба потока
        t1.start();
        t2.start();
        // дожидаемся их завершения
        t1.join();
        t2.join();
    }

}
```

Эта программа сбрасывает значение счетчика до нуля, а затем увели-

чивает его на 100 параллельно из двух потоков. Этот процесс выполняется много раз, и в случае когда итоговое значение счетчика оказывается отличным от ожидаемого значения 200, программа выводит предупреждение в консоль.

На первый взгляд эта программа выглядит безопасно, по крайней мере с точки зрения действия каждого потока в отдельности. Однако если запустить такую программу, вы с высокой долей вероятности увидите одно или несколько предупреждений в ее консольном выводе. При этом не исключен шанс, что программа завершится без предупреждений, т. е. для всех итераций значения счетчика будут совпадать. И это лишний раз показывает, насколько неожиданным и проблемным может быть воспроизведение ошибок в многопоточных программах.

Конкретная проблема в данной программе заключается в том, как именно происходит увеличение счетчика. Дело в том, что в каждом из потоков счетчик увеличивается в два шага. На первом шаге считывается текущее значение счетчика `cnt`. На втором — счетчику присваивается только что считанное значение, увеличенное на 1. Сама проблема в том, что с некоторой вероятностью один из потоков может считать значение счетчика сразу после того, как другой поток считал то же самое значение. В результате оба потока запишут в поле `cnt` одно и то же значение, что и приведет к некорректным итоговым значениям. В рамках дальнейших параграфов рассмотрим то, как эту программу можно превратить в корректную с точки зрения многопоточности.

Приведем пример проблемной последовательности действий.

1. Поток `t1` читает значение счетчика `cnt` из памяти. Значение `cnt` равно 11.

2. Поток `t2` читает значение счетчика `cnt` из памяти. Значение `cnt` равно 11.

3. Поток `t1` вычисляет увеличенное значение счетчика, равное 12. Значение `cnt` равно 11.

4. Поток `t2` вычисляет увеличенное значение счетчика, равное 12. Значение `cnt` равно 11.

5. Поток `t1` записывает увеличенное значение счетчика в память. Значение `cnt` равно 12.

6. Поток `t2` записывает увеличенное значение счетчика в память. Значение `cnt` равно 12.

Отметим также, что этот пример показывает одну из наиболее тривиальных проблем из класса всех подобных проблем многопоточного кода. При более сложном и притом также небезопасном с точки зрения многопоточности коде проблемы могут оказаться гораздо менее наглядными и ожидаемыми.

В общем, наиболее частые проблемы, встречающиеся в некорректных с точки зрения многопоточности программах, сводятся к двум категориям. Первая категория — это состояние гонки ("race condition"), т. е. ситуация, когда данные приходят в некорректное состояние в результате несинхронизированных изменений. Рассмотренный нами пример иллюстрирует как раз эту категорию. Вторая категория — это проблемы в программах, где потоки синхронизированы, но во время взаимодействия потоков возникают ситуации, препятствующие продолжению корректной работы программы. Наиболее часто встречающаяся проблема из второй категории — это взаимная блокировка ("deadlock") — два или более потока оказываются взаимно заблокированы в ожидании освобождения ресурсов, принадлежащих обоим. В этом случае программа или некоторая ее часть прекращает свое выполнение. Имеются и другие проблемы подобного сорта, но они встречаются не настолько часто, поэтому освещать их мы не будем.

Введем одно важное определение и опишем основные средства синхронизации, предоставляемые Java.

10.1. Отношение happens-before

Прежде чем приступить к рассмотрению средств синхронизации, нам нужно ввести одно важное для многопоточного программирования на языке Java понятие. Речь о так называемом отношении "happens-before" ("происходит-до"), описываемом в "Java Memory Model" и спецификации "Java Language Specification" (JLS) и упоминаемом в стандартной документации JDK. Это отношение между некоторыми действиями, выполняемыми в разных потоках программы. При наличии отношения "happens-before" между некоторыми действиями А и Б все операции и данные, производимые и изменяемые действием А, будут гарантированно выполнены и видны до начала выполнения действия Б.

Поясним это на нашем предыдущем примере с целочисленным счетчиком. Напомним, что в нашей программе было два потока, выполняющих одновременно увеличение счетчика. Каждый из потоков читал счетчик и записывал в него увеличенное на 1 значение. При наличии отношения “happens-before” между этими действиями потоков счетчик будет корректно увеличиваться и его конечное значение всегда будет равно 200. Без этого отношения операция чтения в одном из потоков может произойти между операциями чтения и записи в другом потоке, что мы и наблюдали в примере без корректной синхронизации.

На самом деле синхронизация потоков всегда подразумевает отношение “happens-before”[18] и обеспечивается за счет того или иного набора низкоуровневых механизмов: запрета переупорядочивания инструкций JIT-компилятором, инструкций для барьеров памяти (“memory barrier” или “memory fence”), механизмов синхронизации уровня операционной системы (мьютексы, семафоры и критические секции) или атомарных машинных инструкций для той или иной архитектуры процессора (например, CAS-инструкций[19] для процессоров x86, Itanium и Sparc). В рамках этой главы мы не будем углубляться в эти детали. Однако поскольку само понятие носит несколько абстрактный характер, мы будем рассматривать отношение “happens-before” в свете конкретных (высокоуровневых) механизмов синхронизации, предоставляемых Java. Начнем с наиболее известного из них — блоков `synchronized`.

Все инструменты языка Java, которые мы будем рассматривать в рамках этого раздела, обеспечивают отношение “happens-before”. Для этих инструментов мы поясним, к чему конкретно относится это отношение. Подчеркнем, что отношение “happens-before” в Java возникает и в некоторых других случаях, но поскольку эта глава — обзорное введение в многопоточное программирование, мы не будем вдаваться во все детали и перечислять все такие случаи. Любопытному читателю советуем обратиться к спецификации ”Java Language Specification”.

Теперь рассмотрим базовые средства синхронизации в Java.

[18]Однако отношение “happens-before” может иметь место и без пессимистичной (блокирующей) синхронизации. Мы убедимся в этом, когда будем рассматривать ключевое слово `volatile`.

[19]Аббревиатура CAS означает “compare-and-swap”, т. е. ”сравнить-и-заменить”. Это разновидность процессорных инструкций, атомарно выполняющих изменение значения в памяти при условии равенства значения указанному. Как правило, такие инструкции используют для реализации неблокирующей синхронизации, речь о которой пойдет далее.

10.2. Синхронизированные методы и блоки (synchronized)

Мы начнем рассматривать синхронизированные методы и блоки с их синтаксиса, а затем поговорим об особенностях реализации и рассмотрим их работу на примере.

Для объявления синхронизированного метода, обычного или статического, достаточно пометить его ключевым словом `synchronized`. Ниже показан пример этого синтаксиса:

```
public class A {

    public synchronized void syncMethod() {
        // тело метода
    }

    public static synchronized void syncStaticMethod() {
        // тело метода
    }

}
```

Синхронизированные блоки по синтаксису схожи с обычными блоками кода, но как и синхронизированные методы, требуют ключевого слова `synchronized`. Для объявления синхронизированного блока потребуется какой-либо объект, в неявном виде предоставляющий монитор для синхронизации блока[20]. Рассмотрим синтаксис таких блоков на простом примере:

```
public class A {

    public Object o = new Object();

    public void methodWithSyncBlock() {
        // в скобках указывается объект для синхронизации
        synchronized (o) {
            // тело блока
        }
    }
```

[20]Подробнее о мониторах и их роли в синхронизированных блоках и методах поговорим немного позже.

```
}
```

Синхронизированные методы и блоки гарантируют уже знакомое нам отношение “happens-before”. Применительно к ним оно означает, что только один поток в одно и то же время может находиться внутри синхронизированного метода или блока и выполнять код из его тела. Все остальные потоки, пытающиеся выполнить данный метод или блок, ожидают, когда первый поток закончит выполнение тела метода или блока, при любом исходе этого выполнения — в штатном режиме или с выбросом исключения.

Продемонстрируем это на нашем примере со счетчиком, в который добавим синхронизированный блок. Код модифицированного примера выглядит так:

```
public class Main {

    private static final int ITERATIONS = 10_000;
    private static Object lock = new Object();
    private static int cnt;

    public static void main(String[] args)
            throws InterruptedException {
        for (int i = 0; i < ITERATIONS; i++) {
            cnt = 0;
            inc();
            if (cnt != 200) {
                System.out.printf("Ошибка. " +
                        "Счетчик оказался равен %d\n", cnt);
            }
        }

        System.out.println("Программа завершена");
    }

    public static void inc()
            throws InterruptedException {
        // оба потока увеличивают счетчик на 100
        Runnable incrementer = new Runnable() {
```

```
            @Override
            public void run() {
                for (int i = 0; i < 100; i++) {
                    // синхронизация изменений счетчика
                    synchronized (lock) {
                        cnt = cnt + 1;
                    }
                }
            }
        };
        Thread t1 = new Thread(incrementer);
        Thread t2 = new Thread(incrementer);
        // стартуем оба потока
        t1.start();
        t2.start();
        // дожидаемся их завершения
        t1.join();
        t2.join();
    }

}
```

В этом примере синхронизированный блок использует объект из поля `lock`, который добавлен исключительно для целей синхронизации. Но поскольку в синхронизированных блоках можно использовать любой объект, то в этой программе вместо объекта `lock` можно было бы использовать объект-класс `Main.class` с точно таким же эффектом.

Эта программа является корректной с точки зрения синхронизации памяти. Оба потока захватывают синхронизированный блок (вернее, монитор объекта, но об этом позже) прежде, чем прочитать и увеличить текущее значение счетчика. Эту последовательность действий, чтение и изменение значения, единовременно осуществляет только один поток. Поэтому можно было бы изменить общее количество итераций на существенно большее, и конечное значение счетчика всегда было бы равно 200.

Поговорим теперь о синхронизированных методах и блоках более детально. Как уже отмечалось ранее, их реализация основана на концепции мониторов. Каждый объект в Java неявным образом ассоциирован с *монитором* (“monitor”), т. е. ресурсом, который может быть захвачен

или освобожден каким-либо потоком. Для лучшего понимания принципа работы мониторов можно считать, что монитор эквивалентен ссылке на объект, т. е. разные объекты имеют разные ссылки в памяти, а значит, и мониторы.

Только один поток может захватить монитор в один и тот же момент времени. Все остальные потоки, пытающиеся захватить монитор, будут заблокированы до момента, пока монитор не будет освобожден. В момент, когда монитор будет освобожден, один из заблокированных потоков будет уведомлен о событии и, в свою очередь, захватит монитор. Выбор этого потока осуществляется произвольным образом, т. е. Java не дает никаких гарантий порядка с точки зрения выбора следующего потока в момент освобождения монитора.

Интересной особенностью реализации является то, что один и тот же поток может захватить один и тот же монитор несколько раз. В такой ситуации каждое освобождение монитора будет отменять эффект одного действия по его захвату, т. е. в такой ситуации N захватов потребуют N освобождений. Эта логика обеспечивает корректную работу вложенных вызовов синхронизированных методов и вложенных синхронизированных блоков.

При входе в синхронизированный блок или вызове синхронизированного метода текущий поток делает попытку захвата монитора для объекта, ассоциированного с блоком или методом. В этот момент выполнение потока блокируется до успешного захвата монитора. После захвата монитора тело блока или метода выполняется. И наконец, после завершения выполнения тела блока или метода, независимо от результата этого выполнения — штатного или с выбросом исключения, монитор автоматически освобождается.

Мы уже знаем, что синхронизированные блоки требуют указания объекта для использования его монитора в явном виде. Синхронизированные методы неявным образом используют в качестве такого объекта экземпляр класса, у которого вызван метод. Синхронизированные статические методы используют объект-класс[21] для класса, в котором они объявлены.

10.3. Методы wait(), notify() и notifyAll() класса Object

В базовом классе `java.lang.Object` имеется ряд методов, обеспечивающих синхронизацию потоков. Это методы `wait()`, `notify()`, а также

[21]Это знакомые нам по главе 4 третьей части пособия объекты класса `Class`.

`notifyAll()`. Как и синхронизированные методы и блоки, они используют монитор, ассоциированный с объектом.

Опишем логику работы этих методов:

- `void wait(long timeout)` — ставит поток на паузу на заданное время или до вызова одного из методов `notify()`, `notifyAll()` или `interrupt()` у потока. Для вызова этого метода текущий поток должен являться владельцем монитора, соответствующего данному объекту. Как следствие, вызов должен находиться внутри синхронизированного метода или блока. В противном случае будет выброшено исключение `java.lang.IllegalMonitorStateException`. При вызове метода `wait()` текущий поток освобождает монитор объекта и начинает ожидание. Этот метод также может выбросить знакомое нам исключение `InterruptedException`;

- `void wait()` — аналогичен предыдущему, но максимальное время ожидания при этом никак не ограничивается;

- `void notify()` — пробуждает (разблокирует для дальнейшего выполнения) один из потоков, ожидающих монитор объекта, т. е. заблокированных вызовом метода `wait()`. В случае нескольких ожидающих потоков выбор потока для пробуждения осуществляется произвольно. Пробужденный поток не сможет продолжить выполнение, пока не захватит монитор объекта, а значит, по крайней мере, пока текущий поток не освободит монитор путем выхода из синхронизированного метода или блока. Как и в случае с методом `wait()`, текущий поток должен являться владельцем монитора, соответствующего объекту данного потока. В противном случае будет выброшено исключение `java.lang.IllegalMonitorStateException`;

- `void notifyAll()` — пробуждает сразу все потоки, ожидающие монитор объекта. В остальном поведение данного метода и пробужденных потоков совпадает с таковыми для метода `notify()`.

С рассмотренными методами связан один подводный камень, удостоенный того, чтобы быть описанным в Javadoc документации JDK. Поскольку механизм `wait()`/`notify()` полагается на низкоуровневые средства, предоставляемые операционной системой, особенности реализации в той или иной ОС влияют на работу этого механизма в Java. Подводный камень заключается в том, что вошедший в ожидание через вызов `wait()`

поток может быть пробужден самой операционной системой, а не вызовом `notify()`, `notifyAll()` или `interrupt()`. Такое поведение операционной системы называют "ложным пробуждением" ("spurious wakeup"), и встречается оно, как правило, на ОС семейства Linux. Такое пробуждение может нарушить логику программы, поэтому документация рекомендует дополнять вызовы метода `wait()` проверкой на какое-то дополнительное условие, как правило, меняющееся совместно с вызовом метода `notify()`. Таким образом, рекомендуемый вызов метода `wait()` выглядит примерно так:

```
synchronized (obj) {
    while (<условие-пока-не-наступило>)
        obj.wait();

    // обработка наступления условия
}
```

Отметим, что на практике методы `wait()`, `notify()` и `notifyAll()` используются нечасто. Пример их использования будет приведен в конце этой главы, когда будет реализован класс Семафор.

10.4. Ключевое слово volatile

Любое поле в Java, примитивного или объектного типа, обычное или статическое, можно пометить модификатором `volatile`, непосредственно влияющим на синхронизацию потоков при чтении и записи в это поле. Этот модификатор добавляет два свойства для соответствующего поля.

Во-первых, чтение поля и запись в `volatile` поле происходят атомарно. Под атомарной операцией подразумевается, что при совершении такой операции каким-либо потоком ни один другой поток не увидит промежуточного состояния памяти, например, частично считанного или записанного значения примитивного типа. Другими словами, операция будет совершена как единое целое, и в случае ее успешного завершения результат будет доступен для наблюдения другими потоками полностью.

Это свойство может быть важно для полей типов `long` и `double`, поскольку спецификация[22] допускает выполнение записи в поля этих 64-битных типов как двух последовательных операций записи 32-битных значений. Следовательно, некоторые реализации JVM могут осуществлять

[22]Речь идет об уже известной нам спецификации "Java Language Specification" (JLS).

запись в поля типов `long` и `double` не атомарно. Модификатор `volatile` применительно к таким полям обеспечивает атомарность записи и чтения.

Между операцией записи в `volatile` поле и последующими операциями чтения из этого поля, происходящими в других потоках, возникает отношение "happens-before". Это в первую очередь означает, что обновленное значение поля становится доступно другим потокам сразу после его записи.

Отметим важную особенность модификатора `volatile`. Он обеспечивает видимость значений `volatile` полей, что существенно отличается от работы синхронизированных методов и блоков. При необходимости совершения нескольких действий модификатора `volatile` для одного или нескольких полей будет недостаточно для обеспечения корректной работы многопоточной программы. Например, если бы мы пометили поле `cnt` модификатором `volatile` в нашем начальном, не синхронизированном примере со счетчиком, то эта программа все равно осталась бы некорректной с точки зрения работы с памятью и конечные значения счетчика могли бы отличаться от 200.

В целом, модификатор `volatile` не является заменой синхронизации, в частности, синхронизированным методам и блокам; и использовать его нужно только при необходимости и к месту. Для корректного использования `volatile` должны соблюдаться следующие условия:

- запись в поле не должна зависеть от его текущего значения;
- проверки текущего значения поля не должны включать проверки значений других полей. Примером такой проверки может быть оператор `if` или условие цикла.

Продемонстрируем один из вариантов практического применения модификатора `volatile`. Рассмотрим гипотетическое приложение, которое включает обработку некоего сигнала остановки каких-либо вычислений или обработки данных. Условным практическим применением рассматриваемого приема может быть HTTP-сервер, который при получении определенного запроса от администратора может перестать обрабатывать все или часть запросов от пользователей. Итак, рассмотрим код приложения:

```
public class CalcThread extends Thread {

    // флаг остановки вычислений
    private volatile boolean shutdownRequested;
```

```
    public void shutdown() {
        shutdownRequested = true;
    }

    @Override
    public void run() {
        while (!shutdownRequested) {
            // выполняем некие вычисления
            fib(40);
        }
        System.out.println("Вычисления завершены");
    }

    // вычисление чисел Фибоначчи через рекурсию
    // (имеет экспоненциальное время выполнения)
    private int fib(int num) {
        if (num == 0) return 0;
        if (num == 1) return 1;
        return fib(num - 2) + fib(num - 1);
    }

}

public class Main {

    public static void main(String[] args)
            throws InterruptedException {
        System.out.println("Начало вычислений");
        CalcThread thread = new CalcThread();
        thread.start();

        // условное получение сигнала
        // о завершении вычислений
        Thread.sleep(10_000);
        System.out.println("Сигнал остановки");
        thread.shutdown();
```

```
        // условное продолжение работы программы
        Thread.sleep(10_000);

        System.out.println("Программа завершена");
    }

}
```

Это программа выведет в консоль следующее:

```
Начало вычислений
Сигнал остановки
Вычисления завершены
Программа завершена
```

Конечно, конкретно в данном примере мы могли бы использовать механизм кооперативного прерывания работы потоков с помощью знакомого нам метода `interrupt()`. Однако в случае, когда в приложении используется множество потоков и нет прямого доступа к ним, прерывание работы потоков может быть неприемлемо. Кроме того, рассмотренный вариант использования модификатора `volatile` дает больше контроля над точкой остановки вычислений, в то время как сигнал прерывания может привести к выбросу исключения `InterruptedException` в потоках, выполняющих вычисления или обработку данных, что требует корректной обработки прерывания во всех таких потоках.

11. Основные возможности библиотеки java.util.concurrent

Ранее мы рассмотрели базовые средства синхронизации в Java. Однако стандартная библиотека Java содержит большое число классов, предоставляющих более высокоуровневые абстракции, нежели рассмотренные средства. Эти классы содержатся в пакете `java.util.concurrent`. Данный пакет включает в себя различные утилиты для конкурентного многопоточного программирования. Подробное рассмотрение всех возможностей, предоставляемых этим пакетом, пожалуй, потребовало бы написания отдельной книги, во многом дублирующей стандартную докумен-

тацию. Поэтому ограничимся перечислением наиболее значимых из этих классов.

Мы отметим следующие подпакеты, классы и интерфейсы из пакета `java.util.concurrent`:

- `ThreadPoolExecutor` — уже знакомый нам класс для управления пулом потоков;
- `ScheduledThreadPoolExecutor` — класс, унаследованный от класса `ThreadPoolExecutor` и дополняющий возможности родителя. Позволяет откладывать выполнение задачи на определенное время или выполнять ее периодически;
- `ConcurrentLinkedQueue` — класс, реализующий потокобезопасную неблокирующую коллекцию-очередь;
- `PriorityBlockingQueue` — класс, реализующий потокобезопасную блокирующую коллекцию-очередь. Эта коллекция отслеживает приоритеты элементов очереди, задаваемые посредством реализации интерфейса `Comparable` или через объект `Comparator`, и имеет методы для изменения очереди, основанные на блокирующей синхронизации;
- `ConcurrentHashMap` — класс, реализующий потокобезопасную коллекцию для ассоциативного массива, основанного на хеш-таблице. Операции чтения элементов в данном классе используют неблокирующую синхронизацию, в то время как операции изменения коллекции подразумевают блокирующую синхронизацию с заданным значением конкурентности. Последнее означает, что хеш-таблица разбивается на некоторое число частей, каждая из которых обновляется независимо и параллельно с другими частями, однако одновременные изменения в одной и той же части таблицы используют блокирующую синхронизацию и выполняются последовательно;
- `Semaphore` — класс, реализующий семафор. Этот объект ограничивает количество потоков, которым разрешено одновременно войти в определенный участок кода. О семафорах мы поговорим подробно в одном из следующих параграфов;
- `CountDownLatch` — класс, позволяющий заблокировать выполнение кода до получения определенного количества сигналов или событий;

- подпакет `java.util.concurrent.locks` — содержит различные утилиты для блокировок и ожидания наступления определенного условия. Они не повторяют базовые возможности языка, а предоставляют дополнительные возможности. Например, класс `ReentrantLock`, помимо возможностей, схожих с возможностями синхронизированных блоков, позволяет осуществлять оптимистичную блокировку и может отслеживать очередность ожидающих получения блокировки потоков;

- подпакет `java.util.concurrent.atomic` — содержит набор классов, осуществляющих неблокирующую синхронизацию для какого-либо значения примитивного типа или ссылки. Например, в этом пакете имеется класс `AtomicBoolean`, позволяющий проверить текущее значение булевского типа и, если оно равно переданному ожидаемому значению, изменить его на другое значение в рамках одной атомарной операции[23].

Еще раз отметим, что мы перечислили далеко не все классы рассматриваемого пакета. В частности, он содержит потокобезопасные версии коллекций, списков, ассоциативных массивов и множеств. Кроме этих возможностей, стандартный утилитный класс `Collections` включает в себя набор методов, начинающихся с префикса `synchronized*`, например, `synchronizedSet()` и `synchronizedList()`, позволяющих превратить любую коллекцию в потокобезопасную посредством блокирующей синхронизации, основанной на использовании одного общего монитора для большинства методов коллекции. Однако в большинстве случаев потокобезопасные коллекции из пакета `java.util.concurrent` показывают бóльшую производительность в многопоточных программах.

12. Дополнительные возможности многопоточного программирования

Рассматриваемые в этом разделе возможности не являются инструментом многопоточного программирования в буквальном смысле. Скорее, это дополнительная возможность, про которую следует знать каждому программисту, пишущему многопоточные приложения.

[23]Этот класс, как и многие другие из пакета `java.util.concurrent.atomic`, основан на использовании упоминавшихся ранее CAS-инструкций.

Ранее уже говорилось о том, что стек потока используется JVM для хранения локальных переменных и других данных, относящихся к выполнению кода потока. Но помимо этих данных, в стеке потока допускается хранить так называемые локальные переменные потока ("thread-local variables"). Это значения, инициализируемые, а затем доступные исключительно в рамках выполнения конкретного потока. Другими словами, если поток А сохраняет какое-то значение в локальную переменную потока, то другой поток Б не может прочитать ее значение — оно будет доступно только из потока А.

В Java для работы с локальными переменными потока предназначен класс `java.lang.ThreadLocal`. Этот класс параметризован типом локальной переменной потока и содержит следующие методы:

- `T get()` — возвращает значение переменной для текущего потока;
- `void remove()` — очищает значение переменной для текущего потока;
- `void set(T value)` — задает значение переменной для текущего потока.

Локальные переменные потока позволяют ассоциировать какие-то объекты с конкретным потоком без хранения и передачи этих объектов в объект потока в явном виде. На практике это используют, например, для хранения диагностических данных об HTTP-запросах в веб-приложениях (как то идентификаторах запроса в микросервисных приложениях), что позволяет дополнять этими диагностическими данными записи в логах (журнале) приложения без необходимости передачи их между модулями.

В качестве демонстрации использования локальных переменных потока рассмотрим пример с упрощенной демонстрацией подхода логирования диагностических данных в веб-приложениях. Код нашей программы выглядит следующим образом:

```
public class Main {

    // локальная переменная потока для идентификатора задачи
    private static final ThreadLocal<String> TASK_ID_VAR =
            new ThreadLocal<>();

    public static void main(String[] args)
```

```
            throws InterruptedException {
        // создаем пул размера 2
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
                2, 2, 60, TimeUnit.SECONDS,
                new LinkedBlockingQueue<Runnable>(5));

        // добавляем в пул задачи
        for (int i = 0; i < 5; i++) {
            executor.execute(new ThreadLocalRunnable());
        }

        // дожидаемся завершения выполнения всех задач
        executor.shutdown();
        while (!executor.awaitTermination(10,
                TimeUnit.SECONDS)) {
            System.out.println("Ожидаем завершения");
        }
        System.out.println("Все задачи завершены");
    }

    // инициализирует идентификатор и сохраняет его в
    // локальную переменную потока
    public static void initTaskId() {
        // генерируем случайный идентификатор
        UUID id = UUID.randomUUID();
        Main.TASK_ID_VAR.set(id.toString());
    }

    // логирует сообщение + идентификатор
    public static void logWithTaskId(String msg) {
        String taskId = Main.TASK_ID_VAR.get();
        System.out.println(
                String.format("[%s] " + msg, taskId));
    }

}
```

```
public class ThreadLocalRunnable implements Runnable {

    @Override
    public void run() {
        initTaskId();

        logWithTaskId("Стартуем задачу");
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        logWithTaskId("Задача завершилась");
    }

}
```

В результате выполнения этой программы в консоль будет выведено примерно следующее:

```
[ad594df8-1d4e-4642-bad6-c6956a362c22] Стартуем задачу
[093f2dc2-b42a-4652-96cc-5bfb666da943] Стартуем задачу
[093f2dc2-b42a-4652-96cc-5bfb666da943] Задача завершилась
[ad594df8-1d4e-4642-bad6-c6956a362c22] Задача завершилась
[56179766-9b41-4ad3-b489-df2fa544d2e6] Стартуем задачу
[02795f1c-c4b9-438a-a05c-6924fe18a552] Стартуем задачу
[02795f1c-c4b9-438a-a05c-6924fe18a552] Задача завершилась
[16e36cab-3c83-4331-b61f-dcb2a9b7175e] Стартуем задачу
[56179766-9b41-4ad3-b489-df2fa544d2e6] Задача завершилась
[16e36cab-3c83-4331-b61f-dcb2a9b7175e] Задача завершилась
Все задачи завершены
```

Для каждой из задач, выполняющихся в пуле потоков, генерируется случайный идентификатор, который сохраняется в локальную переменную потока. Пример наглядно показывает, что в дальнейшем для добавления этого идентификатора в лог программы не требуется сохранять этот идентификатор в объекте задачи и передавать его при вызове других модулей. Требуется лишь обратиться к объекту `ThreadLocal` в момент формирования записи лога.

13. Пример класса Семафор

Как мы уже знаем, в стандартной библиотеке уже реализован класс `java.util.concurrent.Semaphore`. Однако в учебных целях в этом параграфе мы реализуем более простую версию[24] этого класса.

Семафор представляет собой потокобезопасный объект, отслеживающий определенное количество разрешений для конкурирующих друг с другом потоков. Сразу после создания семафора весь его набор разрешений доступен для всех потоков. Во время дальнейшей работы программы каждый из потоков, использующих семафор, вызывает у объекта-семафора метод получения разрешения и блокируется до получения разрешения. После получения разрешения и завершения связанной с разрешением работы поток обязан освободить разрешение, вызвав соответствующий метод объекта-семафора. Семафоры применяют, когда требуется ограничить одновременное количество потоков, выполняющих некую операцию и/или работающих с каким-то ресурсом.

С точки зрения нашей реализации отслеживание количества разрешений достаточно тривиально — для этого достаточно использовать счетчик и синхронизировать его чтение и изменение, например, с помощью синхронизированного блока. Однако оповещение ожидающих потоков, заблокированных до момента освобождения одного из разрешений, является более сложной задачей. Для этой цели будем использовать методы `wait()` и `notify()` базового класса `Object`. Отметим также, что наша реализация будет достаточно простой и не будет отслеживать очередь ожидающих потоков, т. е. порядок выдачи разрешений ожидающим потокам никак не гарантирован.

Ниже приведен код нашей программы:

```
/**
 * Класс Семафор
 */
public class Semaphore {

    private final int maxSize;
    private int curSize = 0;
```

[24] Отметим, что библиотечный класс `Semaphore` устроен по другим принципам и предоставляет гораздо больше возможностей, нежели наша реализация. Рекомендуем интересующимся читателям самостоятельно ознакомиться с этим классом.

```
private final Object lock = new Object();
private final Object waitLock = new Object();

public Semaphore(int maxSize) {
    if (maxSize <= 0) {
        throw new IllegalArgumentException("Размерность "
                + "семафора должна быть положительной");
    }
    this.maxSize = maxSize;
}

public void acquire() throws InterruptedException {
    boolean needToWait = false;
    synchronized (lock) {
        // пытаемся получить разрешение
        if (curSize < maxSize) {
            curSize++;
            System.out.println("Семафор++: " + curSize);
        } else {
            needToWait = true;
        }
    }

    // если нет свободного места, ждем
    if (needToWait) {
        synchronized (waitLock) {
            waitLock.wait();
        }
        // рекурсия - снова пытаемся получить разрешение
        acquire();
    }
}

public void release() {
    synchronized (lock) {
        if (curSize > 0) {
            curSize--;
```

```
                System.out.println("Семафор--: " + curSize);
                // уведомляем один из ожидающих потоков
                // (если они есть)
                synchronized (waitLock) {
                    waitLock.notify();
                }
            }
        }
    }

}

/**
 * Класс "рабочий поток"
 */
public class WorkerThread extends Thread {

    private final int id;
    private final long sleepBeforeReleaseMs;
    private final Semaphore semaphore;

    public WorkerThread(
            int id,
            long sleepBeforeReleaseMs,
            Semaphore semaphore
    ) {
        this.id = id;
        this.sleepBeforeReleaseMs = sleepBeforeReleaseMs;
        this.semaphore = semaphore;
    }

    @Override
    public void run() {
        try {
            log("пытаюсь захватить семафор");
            semaphore.acquire();
            log("успешно захватил семафор");
```

```
            log("собираюсь спать в течение "
                    + sleepBeforeReleaseMs + "мс");
            Thread.sleep(sleepBeforeReleaseMs);

            log("собираюсь отпустить семафор");
            semaphore.release();
            log("отпустил семафор. Конец работы");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private void log(String msg) {
        System.out.println("Поток " + id + ": " + msg);
    }

}

public class Main {

    public static void main(String[] args) {
        final Semaphore semaphore = new Semaphore(2);
        for (int i = 0; i < 4; i++) {
            // берем одинаковое время сна для каждой
            // следующей пары потоков
            int sleepMs = (i / 2 + 1) * 1000;
            WorkerThread thread =
                    new WorkerThread(i, sleepMs, semaphore);
            thread.start();
        }
    }

}
```

В результате работы эта программа выведет в консоль примерно следующее:

```
Поток 0: пытаюсь захватить семафор
```

```
Семафор++: 1
Поток 0: успешно захватил семафор
Поток 2: пытаюсь захватить семафор
Поток 3: пытаюсь захватить семафор
Поток 1: пытаюсь захватить семафор
Семафор++: 2
Поток 2: успешно захватил семафор
Поток 0: собираюсь спать в течение 1000мс
Поток 2: собираюсь спать в течение 2000мс
Поток 0: собираюсь отпустить семафор
Семафор--: 1
Семафор++: 2
Поток 1: успешно захватил семафор
Поток 1: собираюсь спать в течение 1000мс
Поток 0: отпустил семафор. Конец работы
Поток 1: собираюсь отпустить семафор
Поток 2: собираюсь отпустить семафор
Семафор--: 1
Семафор++: 2
Поток 1: отпустил семафор. Конец работы
Семафор--: 1
Поток 2: отпустил семафор. Конец работы
Поток 3: успешно захватил семафор
Поток 3: собираюсь спать в течение 2000мс
Поток 3: собираюсь отпустить семафор
Семафор--: 0
Поток 3: отпустил семафор. Конец работы
```

В целях лучшего усвоения материала рекомендуем читателям реализовать эту или аналогичную программу и провести ряд экспериментов.

Предметный указатель

Библиографический список

1. *Блох Дж.* Java. Эффективное программирование / Дж. Блох. — 3-е изд. — М. : Диалектика, 2019. — 464 с.
2. *Вирт Н.* Алгоритмы и структуры данных / Н. Вирт. — М. : ДМК Пресс, 2016. — 272 с. — (Классика программирования).
3. *Гамма Э.* Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес. — Санкт-Петербург : Питер, 2014. — 366 с.
4. *Курбатова И. В.* Решение комбинаторных задач на языке программирования Java : учеб.-метод. пособие / И. В. Курбатова, М. А. Артемов, Е. С. Барановский. — Воронеж : Издательский дом ВГУ, 2018. — 42 с.
5. *Курбатова И. В.* Язык программирования Java : учебное пособие / И. В. Курбатова, А. В. Печкуров. — Часть 1. Основы синтаксиса и его применение в объекто-ориенторованном программировании. — Воронеж : Издательский дом ВГУ, 2019. — 78 с.
6. *Курбатова И. В.* Язык программирования Java : учебное пособие / И. В. Курбатова, А. В. Печкуров. — Часть 2. Специальные возможности синтаксиса. — Воронеж : Издательский дом ВГУ, 2019. — 92 с.
7. *Шильдт Г.* Java 8. Руководство для начинающих / Г. Шильдт. — М. : Вильямс, 2018. — 720 с.
8. *Эккель Б.* Философия Java / Б. Эккель. — 4-е изд. — СПб. : Питер, 2009. — 640 с. — (Библиотека программиста).
9. *Arnold K.* The Java programming language / K. Arnold, J. Gosling, D. Holmes. — Forth edition. — Boston, MA : Addison Wesley Professional, 2005. — 928 p.
10. *Hill E. F.* Jess in action : Java rule-based systems / E. F. Hill. — Greenwich, CT : Manning Publications Co., 2003. — xxxii+443 p.
11. Java Concurrency in Practice / D. Lea, D. Holmes, J. Bowbeer [et al.]. — First edition. — Boston, MA : Addison-Wesley Professional, 2006. — xx+404 p.

12. The Java language specification / J. Gosling, B. Joy, G. Steele [et al.]. — Eighth edition. — Boston, MA : Pearson Education and Addison-Wesley Professional, 2015. — xx+768 p.

13. The Java virtual machine specification / T. Lindholm, F. Yellin, G. Bracha, A. Buckley. — Eighth edition. — Boston, MA : Addison-Wesley Professional, 2014. — xvi+581 p.

14. *Oaks S.* Java Performance: The Definitive Guide / S. Oaks. — First edition. — Beijing–Cambridge : O'Reilly Media, Inc., 2014. — xiv+409 p.

У ч е б н о е и з д а н и е

Курбатова Ирина Витальевна,
Печкуров Андрей Викторович

ЯЗЫК ПРОГРАММИРОВАНИЯ JAVA

Учебное пособие

Ч а с т ь 4

Особенности языка и платформы

Редактор *В. Г. Холина*
Компьютерная верстка *И. В. Курбатовой*

Подписано в печать 15.09.2020. Формат 60 × 84/16.
Уч.-изд. л. 5,9. Усл. печ. л. 5,8. Тираж 50 экз. Заказ 128

Издательский дом ВГУ
394018 Воронеж, пл. Ленина, 10

Отпечатано с готового оригинал-макета
в типографии Издательского дома ВГУ
394018 Воронеж, ул. Пушкинская, 3