Глава 2

Защитное программирование

Защитным называется такое *программирование*, которое тщательно защищено и помогает разрабатывать надежное программное обеспечение таким образом, чтобы максимально обезопасить каждый его компонент, например, проверкой достоверности недокументированных допущений [Goodliffe 2007]. Рекомендации, приведенные в этой главе, относятся к тем областям программирования на языке Java, которые помогают ограничить последствия ошибки или благополучно выйти из состояния ошибки.

Механизмы, имеющиеся в языке Java, должны применяться для ограничения области и срока действия, а также доступности программных ресурсов. Кроме того, поддерживаемые в Java аннотации могут быть использованы для документирования программы, повышения удобочитаемости ее исходного кода и упрощения ее сопровождения. Программирующие на Java должны принимать во внимание неявные виды поведения и избегать произвольных допущений по поводу поведения системы.

Общим полезным принципом защитного программирования является простота. Сложные системы, прежде всего, трудно понять, сопровождать и добиться правильного их функционирования. Если программная конструкция становится сложной для реализации, следует рассмотреть возможность ее переделки и реорганизации с целью ее упрощения.

И наконец, программа должна быть разработана как можно более надежной. Везде, где только возможно, программа должна помогать исполняющей системе Java, ограничивая применяемые в ней ресурсы и освобождая приобретаемые, когда они больше не нужны. Опять же, этого можно зачастую добиться ограничением срока действия и доступности объектов и других программных конструкций. Но предусмотреть все возможные случаи нельзя, и поэтому следует разработать стратегию для предоставления изящного выхода из положения в качестве последнего средства.

■ 22. Минимизируйте область действия переменных

Минимизация области действия помогает разработчикам избежать типичных ошибок программирования, повышает удобочитаемость кода, связывая объявление переменных с их конкретным применением, а также упрощает сопровождение исходного кода, поскольку неиспользованные переменные легче обнаруживать и удалять. Такая мера позволяет также быстрее восстанавливать объекты средствами системы "сборки мусора" и препятствует нарушению рекомендации 37, "Не затеняйте и не заслоняйте идентификаторы в подобластях действия".

Пример кода, не соответствующего принятым нормам

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, демонстрируется объявление переменной за пределами цикла `for`.

```
public class Scope {
  public static void main(String[] args) {
    int i = 0;
    for (i = 0; i < 10; i++) {
      // выполнить операции
    }
  }
}
```

Приведенный выше исходный код не соответствует принятым нормам защитного программирования на Java, поскольку переменная `i` объявляется в области действия метода, несмотря на то, что она не используется намеренно за пределами цикла `for`. Один из нескольких сценариев, где переменную `i` требуется объявить в области действия метода, возникает в том случае, когда цикл содержит оператор `break`, а значение переменной `i` должно быть проверено по завершении цикла.

Решение, соответствующее принятым нормам

Старайтесь минимизировать область действия переменных везде, где это только возможно. Например, объявляйте параметры цикла в операторе `for` следующим образом:

```
public class Scope {
  public static void main(String[] args) {
    for (int i = 0; i < 10; i++) { // содержит объявление
      // выполнить операции
    }
  }
}
```

Пример кода, не соответствующего принятым нормам

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, демонстрируется объявление переменной `count` за пределами метода `counter()`, хотя эта переменная и не используется вне этого метода.

```
public class Foo {
  private int count;
```

```
  private static final int MAX_COUNT = 10;

  public void counter() {
    count = 0;
    while (condition()) {
      /* ... */
      if (count++ > MAX_COUNT) {
        return;
      }
    }
  }

  private boolean condition() {/* ... */}
  // ни в одном другом методе нет ссылки на переменную count,
  // но в ряде других методов имеется ссылка на переменную MAX_COUNT
}
```

Возможность повторного использования метода снижается потому, что если бы метод был скопирован в другой класс, то переменную `count` пришлось бы переопределять в новом контексте. Более того, метод `counter()` было бы труднее анализировать, поскольку анализу пришлось бы подвергнуть весь поток данных в программе, чтобы определить возможные значения переменной `count`.

Решение, соответствующее принятым нормам

В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, поле `count` объявляется локально в методе `counter()`.

```
public class Foo {
  private static final int MAX_COUNT = 10;

  public void counter() {
    int count = 0;
    while (condition()) {
      /* ... */
      if (count++ > MAX_COUNT) {
        return;
      }
    }
  }
  private boolean condition() {/* ... */}
    // ни в одном другом методе нет ссылки на переменную count,
    // но в ряде других методов имеется ссылка на переменную MAX_COUNT
}
```

Применимость

Обнаружение локальных переменных, объявляемых в более крупной области действия, чем требуется в прикладном коде, является простым и действенным способом, позволяющим исключить возможность появления ложно положительных результатов. Таким же простым способом является и обнаружение нескольких операторов `for`, в которых используется та же самая индексная переменная. Ложно положительные результаты появляются лишь в необычном случае, когда значение индексной переменной предназначается для сохранения в промежутках между циклами.

Библиография

[Bloch 2001] Item 29, "Minimize the Scope of Local Variables"
[JLS 2013] §14.4, "Local Variable Declaration Statements"

23. Минимизируйте область действия аннотации `@SuppressWarnings`

Когда компилятор обнаруживает потенциальные опасности типизации данных, возникающие в результате смешения базовых типов с обобщенным кодом, он выдает *непроверяемые предупреждения*, включая *непроверяемые предупреждения о приведении типов*, *непроверяемые предупреждения о вызове методов*, *непроверяемые предупреждения о создании обобщенных массивов*, а также *непроверяемые предупреждения о преобразовании типов* [Bloch 2008]. Для подавления непроверяемых предупреждений аннотацию `@SuppressWarnings("unchecked")` допускается использовать тогда и только тогда, когда код, выдающий предупреждение, гарантированно типизирован, т.е. обеспечивает типовую безопасность. Типичным примером тому служит смешение устаревшего кода с новым клиентским кодом. Опасности, связанные с игнорированием непроверяемых предупреждений, широко обсуждаются в рекомендации "OBJ03-J. Не смешивайте обобщенные типы с необобщенными базовыми типами в новом коде" из стандарта *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012].

В документации на прикладной интерфейс API аннотаций типа `SuppressWarnings` [API 2013] по этому поводу говорится следующее:

> "В качестве стиля программирования эту аннотацию следует всегда использовать в наиболее глубоко вложенном элементе, где она оказывается наиболее эффективной. Так, если требуется подавить предупреждение в отдельном методе, то аннотацией следует снабдить именно этот метод, а не его класс".

Аннотация может быть использована в объявлении переменных и методов, а также в целом классе. Но при этом очень важно сузить ее область действия таким образом, чтобы подавлялись предупреждения, возникающие в суженой области действия.

Пример кода, не соответствующего принятым нормам

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, область действия аннотации `@SuppressWarnings` охватывает весь класс. Приведенный ниже код опасен тем, что подавляются все непроверяемые предупреждения, возникающие в классе. Пренебрежение таким характером кода может привести к возникновению исключения типа `ClassCastException` во время выполнения.

```
@SuppressWarnings("unchecked")
class Legacy {
  Set s = new HashSet();
  public final void doLogic(int a, char c) {
    s.add(a);
    s.add(c); // операция, не обеспечивающая типовую
              // безопасность, игнорируется
  }
}
```

Решение, соответствующее принятым нормам

Область действия аннотации `@SuppressWarnings` следует ограничить ближайшим фрагментом кода, где формируется предупреждение. В данном случае эта аннотация может быть использована в объявлении переменной экземпляра типа `Set`, как показано ниже.

```
class Legacy {
  @SuppressWarnings("unchecked")
  Set s = new HashSet();
  public final void doLogic(int a, char c) {
    s.add(a); // производит непроверяемое предупреждение
    s.add(c); // производит непроверяемое предупреждение
  }
}
```

Пример кода, не соответствующего принятым нормам (класс `ArrayList`)

Данный пример кода, не соответствующего принятым нормам защитного программирования на Java, взят из прежней реализации класса `java.util.ArrayList`.

```
@SuppressWarnings("unchecked")
public <T> T[] toArray(T[] a) {
  if (a.length < size) {
    // производит непроверяемое предупреждение
    return (T[]) Arrays.copyOf(elements, size, a.getClass());
  }
  // ...
}
```

Во время компиляции класса выдается следующее непроверяемое предупреждение о приведении типов:

```
// непроверяемое предупреждение о приведении типов
ArrayList.java:305: warning: [unchecked] unchecked cast found :
   Object[], required: T[]
return (T[]) Arrays.copyOf(elements, size, a.getClass());
```

Данное предупреждение не может быть подавлено только для оператора `return`, поскольку это не объявление [JLS 2013]. В итоге предупреждения подавляются для всего метода в целом. И это может вызвать осложнения, когда функции, выполняющие операции, не обеспечивающие типовую безопасность, вводятся в метод впоследствии [Bloch 2008].

Решение, соответствующее принятым нормам (класс `ArrayList`)

Если аннотации `@SuppressWarnings` нельзя использовать в подходящей области действия, как в предыдущем примере кода, не соответствующего принятым нормам, то следует объявить новую переменную, чтобы хранить в ней значение и снабдить ее аннотацией `@SuppressWarnings`.

```
// ...
@SuppressWarnings("unchecked")
T[] result = (T[]) Arrays.copyOf(elements, size, a.getClass());
return result;
// ...
```

Применимость

Если не сократить область действия аннотации `@SuppressWarnings`, это может привести к исключениям во время выполнения и нарушению гарантий типовой безопасности. Данное правило не может соблюдаться статически в полной мере. Но в некоторых особых случаях может быть использован статический анализ.

Библиография

[API 2013] Annotation Type SuppressWarnings
[Bloch 2008] Item 24, "Eliminate Unchecked Warnings"
[Long 2012] OBJ03-J. Do not mix generic with nongeneric raw types in new code

■ 24. Минимизируйте доступность классов и их членов

Классы и их члены (классы, интерфейсы, поля и методы) подлежат в Java управлению доступом. Порядок доступа к ним обозначается как наличием соответствующего модификатора доступа (`public`, `protected` или `private`), так и его отсутствием, когда доступ по умолчанию называется *закрытым пакетным доступом*.

В табл. 2.1 приведено упрощенное схематическое представление правил управления доступом. Знаком **x** обозначен конкретный доступ, разрешенный в данной предметной области. Например, знак **x** в столбце "Класс" обозначает, что член класса доступен для кода, присутствующего в том классе, где этот член объявлен. Аналогично знак **x** в столбце "Пакет" обозначает, что соответствующий член доступен из любого класса (или подкласса), определенного в том же самом пакете, при условии, что класс (или подкласс) загружается тем загрузчиком классов, который загружает класс, содержащий данный член. Одно и то же условие для загрузчика классов распространяется только на закрытый пакетный доступ к членам классов.

Таблица 2.1. Правила управления доступом

Спецификатор доступа	Класс	Пакет	Подкласс	Внешний мир
`private`	**x**			
Отсутствует	**x**	**x**	**x***	
`protected`	**x**	**x**	**x****	
`public`	**x**	**x**	**x**	**x**

**Подклассы из одного и того же пакета могут также иметь доступ к членам, у которых отсутствуют модификаторы доступа (по умолчанию или на уровне закрытого пакетного доступа). Дополнительные требования к доступу состоят в том, что подклассы должны загружаться тем загрузчиком классов, который загружает класс, содержащий члены с закрытым пакетным доступом. Подклассы из другого пакета не могут получать доступ к таким членам.*

***Для ссылки на защищенный член код, совершающий доступ к нему, должен содержаться в классе, определяющем данный защищенный член, или же в подклассе этого определяющего класса. Доступ к подклассу должен быть разрешен безотносительно к расположению подкласса в пакете.*

Классам и их членам должен быть предоставлен минимально допустимый доступ, чтобы у злонамеренного кода было как можно меньше возможностей нарушить безопасность.

Классы, насколько это возможно, должны избегать раскрытия своих методов, содержащих (или вызывающих) уязвимый код через интерфейсы, которые допускают только открыто доступные методы, а такие методы являются частью открытого прикладного программного интерфейса (API) отдельного класса. (Следует иметь в виду, что данная рекомендация противоположна рекомендации Джошуа Блоха (Joshua Bloch) отдавать предпочтение интерфейсам над прикладными интерфейсами API [Bloch 2008, Item 16].) Исключением из этого правила служит *реализация* не видоизмененного интерфейса, раскрывающего открытое неизменяемое представление изменяемого класса. (См. рекомендацию "OBJ04-J. Предоставляйте изменяемые классы с функциями копирования для надежной передачи экземпляров ненадежному коду" из стандарта *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012].) Следует также иметь в виду, что даже если неконечный класс доступен по умолчанию, то им все же можно злоупотребить, если он содержит открытые методы. Те методы, в которых выполняются все необходимые проверки и санобработка всех вводимых данных, могут оказаться открытыми через интерфейсы.

Защищенная доступность недействительна для невложенных классов, но вложенные классы могут быть объявлены защищенными. Поля неконечных классов должны объявляться защищенными лишь в редких случаях. Ненадежный код из другого пакета может выполнить подклассификацию и получить доступ к члену класса. Более того, защищенные члены являются частью прикладного интерфейса API отдельного класса, а следовательно, требуют непрерывной поддержки. Если данное правило соблюдается, то объявлять поля защищенными не нужно. В рекомендации "OBJ01-J. Объявляйте члены данных закрытыми и предоставляйте доступные методы-оболочки" из стандарта *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012] предлагается объявлять поля закрытыми.

Если класс, интерфейс, метод или поле являются частью опубликованного прикладного интерфейса API, например, конечного пункта веб-службы, то они могут быть объявлены закрытыми или пакетно-закрытыми. Например, некритичные с точки зрения безопасности классы стимулируются к предоставлению открытых статических фабричных методов для реализации управления экземплярами с помощью закрытого конструктора.

Пример кода, не соответствующего принятым нормам (открытый класс)

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, определяется класс, являющийся внутренним для системы и не входящим ни в один из общедоступных прикладных интерфейсов API. Тем не менее этот класс объявляется как открытый.

```
public final class Point {
  private final int x;
  private final int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public void getPoint() {
    System.out.println("(" + x + "," + y + ")");
  }
}
```

Несмотря на то что данный пример соответствует рекомендации "OBJ06-J. Защитное копирование изменяемых входных параметров и внутренних компонентов" из стандарта *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012], в ненадежном коде может быть получен экземпляр класса `Point` и вызван открытый метод `getPoint()` для получения координат точки.

Решение, соответствующее принятым нормам (конечные классы с открытыми методами)

В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, класс `Point` объявляется как пакетно-закрытый в соответствии с его состоянием как не входящего ни в один из общедоступных прикладных интерфейсов API.

```
final class Point {
  private final int x;
  private final int y;

  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public void getPoint() {
    System.out.println("(" + x + "," + y + ")");
  }
}
```

Класс верхнего уровня (например, `Point`) нельзя объявить закрытым. Пакетно-закрытая доступность оказывается приемлемой, при условии, что исключаются атаки со вставкой пакетов. (См. рекомендацию "ENV01-J. Размещайте весь уязвимый для безопасности код в одном архивном JAR-файле, подписывая и герметизируя его" из стандарта *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012].) Атака со вставкой пакетов происходит во время выполнения, когда защищенные или пакетно-закрытые члены класса могут быть вызваны непосредственно классом, злонамеренно вставленным в тот же самый пакет. Но совершить такую атаку трудно на практике, поскольку цель атаки и небезопасный класс должны загружаться тем же самым загрузчиком классов, помимо того, что от пакета требуется проникновение в подвергаемый атаке код. Небезопасный код, как правило, лишен подобных уровней доступа.

В связи с тем что класс является конечным, метод `getPoint()` может быть объявлен открытым. Открытый подкласс, нарушающий данное правило, не может переопределить метод и раскрыть его для небезопасного кода, а следовательно, он не является доступным для такого кода. Для неконечных классов ограничение доступности методов до закрытого или пакетно-закрытого уровня исключает такую угрозу.

Решение, соответствующее принятым нормам (неконечные классы с неоткрытыми методами)

В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, класс `Point` и его метод `getPoint()` объявляются как пакетно-закрытые. Благодаря этому класс `Point` становится неконечным, а его метод `getPoint()` может вызываться классами, присутствующими в том же самом пакете и загружаемыми общим загрузчиком классов, как показано ниже.

```
class Point {
  private final int x;
  private final int y;

  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  void getPoint() {
    System.out.println("(" + x + "," + y + ")");
  }
}
```

Пример кода, не соответствующего принятым нормам (открытый класс с открытым статическим методом)

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, снова определяется класс, являющийся внутренним для системы и не входящий ни в один из общедоступных прикладных интерфейсов API. Тем не менее этот класс объявляется открытым.

```
public final class Point {
  private static final int x = 1;
  private static final int y = 2;

  private Point(int x, int y) {}

  public static void getPoint() {
    System.out.println("(" + x + "," + y + ")");
  }
}
```

Код из данного примера также соответствует рекомендации "OBJ01-J. Объявляйте члены данных закрытыми и предоставляйте доступные методы-оболочки" из стандарта *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012]. Тем не менее в ненадежном коде может быть получен доступ к классу `Point` и вызван открытый статический метод `getPoint()`, чтобы получить координаты точки по умолчанию. А попытка реализовать управление экземпляром с помощью закрытого конструктора оказывается бесполезной, поскольку открытый статический метод раскрывает внутреннее содержимое класса.

Решение, соответствующее принятым нормам (пакетно-закрытый класс)

В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, уровень доступности класса сокращается до пакетно-закрытого. Доступ к методу `getPoint()` разрешается только классам, расположенным в том же самом пакете. Это препятствует вызову метода `getPoint()` из небезопасного кода и получению координат точки.

```
final class Point {
  private static final int x = 1;
  private static final int y = 2;
```

```
  private Point(int x, int y) {}

  public static void getPoint() {
    System.out.println("(" + x + "," + y + ")");
  }
}
```

Применимость

Предоставление чрезмерного доступа нарушает инкапсуляцию и ослабляет защиту приложений на Java. Система с прикладным интерфейсом API, предназначенным для применения (а возможно, и расширения) в стороннем коде, должна раскрывать прикладной интерфейс API через открытый интерфейс. Потребности в таком прикладном интерфейсе API превышают требования данной рекомендации.

Минимальный уровень доступности к классу и его члену в любом заданном фрагменте кода может быть вычислен таким образом, чтобы исключить внедрение ошибок компиляции. Накладываемое ограничение приводит к тому, что в подобном вычислении могут быть недостаточно учтены намерения программиста при написании кода. Например, неиспользуемые члены могут быть, очевидно, объявлены закрытыми. Но такие члены могут оказаться неиспользуемыми только потому, что в конкретном фрагменте кода, проверяемом по случайному совпадению, может недоставать ссылок на эти члены. Тем не менее такое вычисление может послужить удобной отправной точкой для программиста, стремящегося минимизировать уровень доступа к классам и их членам.

Библиография

[Bloch 2008]	Item 13, "Minimize the Accessibility of Classes and Members"
	Item 16, "Prefer Interfaces to Abstract Classes"
[Campione 1996]	Access Control
[JLS 2013]	§6.6, "Access Control"
[Long 2012]	ENV01-J. Place all security-sensitive code in a single JAR and sign and seal it
	OBJ01-J. Declare data members as private and provide accessible wrapper methods
	OBJ04-J. Provide mutable classes with copy functionality to safely allow passing instances to untrusted code
[McGraw 1999]	Chapter 3, "Java Language Security Constructs"

■ 25. Документируйте потоковую безопасность и пользуйтесь аннотациями везде, где только можно

Языковые средства аннотирования в Java удобны для документирования целей разработки. Аннотация к исходному коду является механизмом для связывания метаданных с элементом программы, чтобы сделать их доступными для анализа компилятором, анализаторами, отладчиками или виртуальной машиной Java (JVM). Для документирования потоковой безопасности или ее отсутствия имеется несколько аннотаций.

Получение аннотаций параллелизма

Два ряда аннотаций параллелизма свободно доступны и лицензированы для применения в любом коде. Первый ряд состоит из четырех аннотаций, описанных в книге *Java Concurrency in Practice* (JCIP) [Goetz 2006], которую можно загрузить по адресу `http://jcip.net`. Аннотации JCIP выпущены под лицензией "С указанием авторства" (Creative Commons Attribution License).

Второй более обширный ряд аннотаций параллелизма предоставляется и поддерживается компанией SureLogic. Эти аннотации выпускаются под лицензией на программное обеспечение веб-сервера Apache (Apache Software License) версии 2.0 и могут быть загружены по адресу `www.surelogic.com`. Аннотации можно проверить инструментальным средством JSure от компании SureLogic, и они остаются полезными для документирования кода, даже если данное инструментальное средство недоступно. К их числу относятся аннотации JCIP, поскольку они поддерживаются инструментальным средством JSure наряду с их архивным JAR-файлом.

Для того чтобы воспользоваться упомянутыми выше аннотациями, следует загрузить один или оба их архивных JAR-файла или добавить эти файлы в путь для построения кода. В последующих разделах описывается применение этих аннотаций для документирования потоковой безопасности.

Документирование намеченной потоковой безопасности

В JCIP предоставляются три аннотации на уровне класса для описания целей разработки в отношении потоковой безопасности. В частности, аннотация `@ThreadSafe` применяется к классу для его обозначения как *потокобезопасного*. Это означает, что ни одна из последовательностей доступа (для чтения и записи в открытые поля или вызова открытых методов) не в состоянии оставить объект в неопределенном состоянии независимо от того, чередуются ли эти виды доступа с динамической или любой внешней синхронизацией или же координацией со стороны вызывающего кода.

В качестве примера ниже приведен исходный код класса `Aircraft`, описываемого как потокобезопасный в документации на принятые в нем правила блокировки. Этот класс защищает свои поля `x` и `y` с помощью реентерабельной блокировки.

```
@ThreadSafe
@Region("private AircraftState")
@RegionLock("StateLock is stateLock protects AircraftState")
public final class Aircraft {
  private final Lock stateLock = new ReentrantLock();
  // ...
  @InRegion("AircraftState")
  private long x, y;
  // ...
  public void setPosition(long x, long y) {
    stateLock.lock();
    try {
      this.x = x;
      this.y = y;
    } finally {
      stateLock.unlock();
    }
  }
  // ...
}
```

Аннотации `@Region` и `@RegionLock` документируют правила блокировки, по которым утверждается обязательство сделать код потокобезопасным. Даже если использовать одну или больше аннотаций `@RegionLock` или `@GuardedBy` для документирования правил блокировки класса, аннотация `@ThreadSafe` все равно позволяет тем, кто просматривает исходный код этого класса, интуитивно выяснить, что он является потокобезопасным.

Аннотация `@Immutable` применяется к *неизменяемым* классам. Неизменяемые объекты, по существу, являются потокобезопасными. Как только эти объекты будут полностью построены, их можно опубликовать по ссылке и надежно сделать их общедоступными для использования в нескольких потоках. В приведенном ниже фрагменте кода демонстрируется неизменяемый класс `Point`.

```
@Immutable
public final class Point {
  private final int f_x;
  private final int f_y;

  public Point(int x, int y) {
    f_x = x;
    f_y = y;
  }

  public int getX() {
    return f_x;
  }

  public int getY() {
    return f_y;
  }
}
```

В рекомендации Джошуа Блоха [Bloch 2008] говорится следующее:

> "Документировать неизменяемость типов `enum` совсем не обязательно. Если только это не очевидно из возвращаемого типа, то статические фабричные методы должны документировать потоковую безопасность возвращаемого объекта, как демонстрируется в классе `Collections.synchronizedMap`".

Аннотация `@NotThreadSafe` применяется к классам, которые не являются потокобезопасными. Многие классы не в состоянии задокументировать свою безопасность для многопоточной обработки. Следовательно, программисту нелегко определить, является ли класс потокобезопасным. А данная аннотация ясно указывает на то, что классу явно недостает потоковой безопасности.

Например, большинство классов, реализующих коллекции в пакете `java.util`, не являются потокобезопасными. Так, в классе `java.util.ArrayList` потоковую безопасность можно было бы задокументировать следующим образом:

```
package java.util.ArrayList;

@NotThreadSafe
public class ArrayList<E> extends ... {
  // ...
}
```

Документирование правил блокировки

Очень важно задокументировать все блокировки, применяемые для защиты разделяемого общего состояния. В этой связи Брайан Гоетц с коллегами [Goetz 2006, стр. 28] рекомендуют следующее:

> "Все виды доступа к изменяемой переменной состояния в более чем одном потоке должны выполняться с помощью одной и той же устанавливаемой блокировки. В этом случае можно утверждать, что переменная защищена данной блокировкой".

Для этой цели предоставляется аннотация `@GuardedBy` от JCIP, а также аннотация `@RegionLock` от компании SureLogic. Поле или метод, к которому применяется аннотация `@GuardedBy`, могут быть доступны только при установке конкретной блокировки. Это может быть внутренняя или динамическая блокировка, как, например, в пакете `java.util.concurrent`. В приведенном ниже примере демонстрируется класс `MovablePoint`, реализующий перемещаемую точку, способную запоминать свое прежнее местоположение, используя списочный массив `memo`.

```
@ThreadSafe
public final class MovablePoint {

  @GuardedBy("this")
  double xPos = 1.0;
  @GuardedBy("this")
  double yPos = 1.0;
  @GuardedBy("itself")
  static final List<MovablePoint> memo
    = new ArrayList<MovablePoint>();

  public void move(double slope, double distance) {
    synchronized (this) {
      rememberPoint(this);
      xPos += (1 / slope) * distance;
      yPos += slope * distance;
    }
  }

  public static void rememberPoint(MovablePoint value) {
    synchronized (memo) {
      memo.add(value);
    }
  }
}
```

В аннотациях `@GuardedBy` к полям `xPos` и `yPos` указывается, что доступ к этим полям защищен установкой блокировки текущего объекта `this`. Метод `move()`, вносящий изменения в эти поля, также синхронизирован с текущим объектом `this`. В то же время аннотация `@GuardedBy` к списочному массиву `memo` указывает на то, что блокировка объекта типа `ArrayList` защищает его содержимое. Метод `rememberPoint()` также синхронизирован со списочным массивом `memo`.

Но недостаток аннотации `@GuardedBy` заключается в том, что она неспособна указывать на отсутствие отношений между полями в классе. Впрочем, этот недостаток можно преодолеть с помощью аннотации `@RegionLock` от компании SureLogic, где объявляется

новая региональная блокировка класса, к которому применяется данная аннотация. В результате такого объявления устанавливается новая именованная блокировка, связывающая конкретный объект блокировки с отдельной областью класса, которая может быть доступна только при установке блокировки. В приведенном ниже примере кода правила блокировки в классе `SimpleLock` обозначают, что синхронизация с экземпляром этого класса защищает все его состояние. В отличие от аннотации `@GuardedBy`, аннотация `@RegionLock` позволяет программисту присвоить явное (и желательно содержательное) имя правилу блокировки.

```
@RegionLock("SimpleLock is this protects Instance")
class Simple { ... }
```

Помимо именования правила блокировки, аннотация `@Region` позволяет присвоить имя области защищаемого состояния. Такое имя ясно дает понять, что состояние и правило блокировки подходят друг другу, как показано в следующем примере:

```
@Region("private AircraftPosition")
@RegionLock("StateLock is stateLock protects AircraftPosition")
public final class Aircraft {
  private final Lock stateLock = new ReentrantLock();

  @InRegion("AircraftPosition")
  private long x, y;

  @InRegion("AircraftPosition")
  private long altitude;
  // ...
  public void setPosition(long x, long y) {
    stateLock.lock();
    try {
      this.x = x;
      this.y = y;
    } finally {
    stateLock.unlock();
    }
  }
  // ...
}
```

В данном примере правило блокировки под названием `StateLock` служит для указания на то, что блокировка объекта типа `stateLock` защищает область под названием `AircraftPosition`, в которую входит изменяемое состояние, используемое для обозначения местоположения самолета.

Построение изменяемых объектов

Обычно построение объекта считается исключением из правила блокировки, поскольку при первоначальном создании объекты привязываются к потоку. Объект привязывается к тому потоку, в котором создается его экземпляр с помощью оператора `new`. После создания объекта его можно благополучно опубликовать в других потоках. Но объект не станет общим до тех пор, пока это не будет разрешено в том потоке, где создан экземпляр данного объекта. Способы надежной публикации объектов, обсуждаемые в рекомендации “TSM01-J.

Не допускайте исчезновения ссылки `this` при построении объекта" из стандарта *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012], могут быть кратко выражены с помощью аннотации `@Unique("return")`.

Например, в следующем фрагменте кода аннотация `@Unique("return")` документирует, что объект возвращается из конструктора в виде однозначной ссылки:

```
@RegionLock("Lock is this protects Instance")
public final class Example {
  private int x = 1;
  private int y;

  @Unique("return")
  public Example(int y) {
    this.y = y;
  }
  // ...
}
```

Документирование правил привязки к потоку

Дин Сазерленд и Уильям Шерлис (William Scherlis) предлагают аннотации, позволяющие документировать правила привязки к потоку. Их подход состоит в том, чтобы разрешить проверку аннотаций относительно написанного кода [Sutherland 2010]. В приведенном ниже примере кода аннотации выражают следующую цель разработки: в программе должен быть хотя бы один поток диспетчеризации событий из библиотеки AWT (Abstract Window Toolkit — абстрактно-оконный инструментарий) и несколько вычислительных потоков, в которых запрещено обрабатывать структуры данных или события AWT.

```
@ThreadRole AWT, Compute
@IncompatibleThreadRoles AWT, Compute
@MaxRoleCount AWT 1
```

Документирование протоколов ожидания и уведомления

Брайан Гоетц с коллегами [Goetz 2006, стр. 395] рекомендуют следующее:

> "Класс, зависящий от состояния, должен полностью раскрыть (и задокументировать) протоколы ожидания и уведомления в своих подклассах или вообще запретить им участие в этих протоколах. (Это расширение принципа "проектирования и документирования наследования, а иначе — его запрещения" [EJ Item 15].) По крайней мере, разработка класса, зависящего от состояния, для целей наследования требует раскрытия условных очередей и блокировок и документирования условных предикатов и правил синхронизации. Она может также потребовать раскрытия базовых переменных состояния. (В худшем случае класс, зависящий от состояния, может раскрыть свое состояние подклассам, но не документировать свои протоколы для ожидания и уведомления. Это все равно, как если бы класс раскрыл свои переменные состояния, но не задокументировал свои инварианты.)"

Протоколы ожидания и уведомления должны быть задокументированы надлежащим образом. В настоящее время еще неизвестны аннотации, предназначенные для этой цели.

Применимость

Аннотирование параллельно выполняемого кода помогает правильно задокументировать цель разработки и может быть использовано для автоматизации процесса обнаружения и предотвращения условий возникновения гонок, в том числе и гонок данных.

Библиография

[Bloch 2008]	Item 70, "Document Thread Safety"
[Goetz 2006]	*Java Concurrency in Practice*
[Long 2012]	TSM01-J. Do not let the **`this`** reference escape during object construction
[Sutherland 2010]	"Composable Thread Coloring"

■ 26. Всегда предоставляйте отклик на результирующее значение метода

Методы следует разрабатывать таким образом, чтобы возвращать значение, позволяющее разработчику выяснить текущее состояние объекта и/или результат операции. Данная рекомендация согласуется с рекомендацией "EXP00-J. Не пренебрегайте значениями, возвращаемыми методами" из стандарта *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012]. Возвращаемое значение должно представлять последнее известное состояние и выбираться с учетом восприятия и ментальной модели разработчика.

Отклик можно также предоставить, сгенерировав стандартное или специальное исключение в виде объекта класса, производного от класса `Exception`. При таком подходе разработчик может по-прежнему получать точную информацию о результате выполнения метода и предпринимать далее необходимые действия. Для этого исключение должно предоставить подробный отчет о ненормальном условии на соответствующем уровне абстракции.

Такие подходы должны применяться в прикладных интерфейсах API в определенном сочетании, чтобы помочь клиентам отличить правильные результаты от неправильных и стимулировать тщательную обработку любых неправильных результатов. В тех случаях, когда имеется общепринятый код ошибки, который просто невозможно интерпретировать как достоверное значение, возвращаемое из метода, должен быть возвращен именно этот код ошибки. А в остальных случаях должно быть сгенерировано исключение. Метод не должен возвращать значение, состоящее как из самого возвращаемого значения, так из кода ошибки (подробнее об этом см. в рекомендации 52 "Избегайте внутренних индикаторов ошибок").

С другой стороны, объект может предоставить метод проверки состояния [Bloch 2008], в котором проверяется, находится ли объект в согласованном состоянии. Такой подход применим только в тех случаях, когда состояние объекта не может быть видоизменено во внешних потоках. Благодаря этому исключается возникновение условия гонок типа "время проверки — время использования" (TOCTOU) между вызовом метода, проверяющего состояние объекта, а также вызовом метода, зависящего от состояния данного объекта. В промежутке между этими вызовами состояние объекта может измениться неожиданно или даже злонамеренно.

Значения или коды ошибок, возвращаемые из методов, должны точно обозначать состояние объекта на определенном уровне абстракции. Клиенты должны быть в состоянии выполнять решающие действия, опираясь на возвращаемое значение.

Пример кода, не соответствующего принятым нормам

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, метод `updateNode()` видоизменяет узел древовидной структуры, если ему удастся обнаружить данный узел в связном списке, а иначе — он ничего с ним не делает.

```
public void updateNode(int id, int newValue) {
  Node current = root;
  while (current != null) {
    if (current.getId() == id) {
      current.setValue(newValue);
      break;
    }
    current = current.next;
   }
}
```

Этому методу не удается указать, что он видоизменил любой узел. Следовательно, в вызывающем коде нельзя определить, был ли метод выполнен успешно или неудачно, но незаметно.

Решение, соответствующее принятым нормам (возврат логического значения)

В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, в качестве результата операции возвращается логическое значение `true`, если узел древовидной структуры был видоизменен, а иначе — логическое значение `false`.

```
public boolean updateNode(int id, int newValue) {
  Node current = root;
  while (current != null) {
    if (current.getId() == id) {
      current.setValue(newValue);
      return true; // узел успешно обновлен
    }
    current = current.next;
  }
  return false;
}
```

Решение, соответствующее принятым нормам (генерирование исключения)

В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, видоизмененный объект типа `Node` возвращается в том случае, если он обнаружен. А если искомый узел недоступен в списке, то генерируется исключение типа `NodeNotFoundException`.

```
public Node updateNode(int id, int newValue)
   throws NodeNotFoundException {
  Node current = root;
  while (current != null) {
```

```
    if (current.getId() == id) {
      current.setValue(newValue);
      return current;
    }
    current = current.next;
  }
  throw new NodeNotFoundException();
}
```

Применение исключений для указания сбоя в программе может стать удачным проектным решением, но генерирование исключений не всегда оказывается уместным. В целом метод должен генерировать исключение только в том случае, если предполагается его успешное завершение, но при этом возникают неисправимые ошибочные ситуации, или же в том случае, если предполагается, что ошибка должна быть исправлена в методе, вызываемом из класса, расположенного вверх по иерархии.

Решение, соответствующее принятым нормам (возврат пустого значения)

В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, возвращается обновленный объект типа `Node`, чтобы разработчик мог просто проверить пустое возвращаемое значение, если операция завершится неудачно.

```
public Node updateNode(int id, int newValue) {
  Node current = root;
  while (current != null) {
    if (current.getId() == id) {
      current.setValue(newValue);
      return current;
    }
    current = current.next;
  }
  return null;
}
```

Возвращаемое значение, которое может быть пустым (`null`), представляет собой внутренний индикатор ошибки, более подробно рассматриваемый в рекомендации 52 “Избегайте внутренних индикаторов ошибок”. Такое проектное решение допустимо, но считается подчиненным другим проектным решениям, в том числе и другим решениям, соответствующим принятым нормам и представленным в данной рекомендации.

Применимость

Если не предоставить подходящий отклик в виде определенного сочетания возвращаемых значений, кодов ошибок и исключений, то объект может перейти в неопределенное состояние, а поведение программы станет непредсказуемым.

Библиография

[Bloch 2008] Item 59. Avoid unnecessary use of checked exceptions

[Long 2012] EXP00-J. Do not ignore values returned by methods

[Ware 2008] *Writing Secure Java Code*

■ 27. Распознавайте файлы, используя несколько файловых атрибутов

Многие уязвимости в защите, имеющие отношение к файлам, возникают в результате того, что программа получает доступ не к тому файловому объекту. Это нередко происходит потому, что имена файлов лишь слабо связаны с базовыми файловыми объектами. Ведь имена файлов не предоставляют сведения, касающиеся характера самого файлового объекта. Более того, привязка имени файла к файловому объекту пересматривается всякий раз, когда имя файла используется в операции. Такой пересмотр может привести к возникновению в приложении условия гонки типа "время проверки — время использования" (TOCTOU). Объекты типа `java.io.File` и `java.nio.file.Path` привязываются к базовым файловым объектам на уровне операционной системы только тогда, когда осуществляется доступ к файлу.

Конструкторы и методы `renameTo()` и `delete()` из класса `java.io.File` опираются только на имена файлов для их распознавания. Это же относится и к методам `get()` из класса `java.nio.file.Path`, `move()` и `delete()` из класса `java.nio.file.Files`. Поэтому всеми этими методами следует пользоваться крайне осторожно.

Правда, файлы можно зачастую распознавать по другим атрибутам, помимо имени файла. Их, например, можно сравнивать по времени создания или модификации. Сведения о созданном или закрытом файле могут быть сохранены, а затем использованы для проверки достоверности подлинности файла, если его требуется открыть снова. А сравнение нескольких атрибутов увеличивает вероятность того, что повторно открываемый файл окажется тем же самым, что и файл, открывавшийся прежде.

Распознавание файлов имеет меньшее значение для приложений, где файлы хранятся в безопасных каталогах, откуда они могут быть доступны только для их владельца, а возможно, и системного администратора (см. рекомендацию "FIO00-J. Не оперируйте файлами в общих каталогах" из стандарта *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012]).

Пример кода, не соответствующего принятым нормам

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, файл распознается по символьной строке с его именем, чтобы выяснить, был ли он открыт, обработан, закрыт и затем еще раз открыт для чтения.

```
public void processFile(String filename){
  // распознать файл из пути к нему
  Path file1 = Paths.get(filename);
  // открыть файл для записи
  try (BufferedWriter bw = new BufferedWriter(new
    OutputStreamWriter(Files.newOutputStream(file1)))) {
      // записать в файл...
  } catch (IOException e) {
      // обработать ошибку
  }

  // закрыть файл

  /*
   * Возникающее здесь условие гонок позволяет совершающему
   * атаку злоумышленнику сменить один файл на другой
  */
```

```
  // еще раз открыть файл для чтения
  Path file2 = Paths.get(filename);

  try (BufferedReader br = new BufferedReader(new
     InputStreamReader(Files.newInputStream(file2)))) {
    String line;
    while ((line = br.readLine()) != null) {
      System.out.println(line);
    }
  } catch (IOException e) {
    // обработать ошибку
  }
}
```

Привязка имени файла к базовому файловому объекту пересматривается при создании объекта типа `BufferedReader`, и поэтому в данном примере кода нельзя гарантировать, что файл, открытый для чтения, окажется тем же самым, что и файл, отрывавшийся ранее для записи. В этом случае совершающий атаку злоумышленник может заменить исходный файл (например, символической ссылкой) в промежутке между первым вызовом метода `close()` и последующим созданием объекта типа `BufferedReader`.

Пример кода, не соответствующего принятым нормам (вызов метода `Files.isSameFile()`)

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, предпринимается попытка обеспечить открытие для чтения того же самого файла, который открывался ранее для записи. И с этой целью вызывается метод `Files.isSameFile()`.

```
public void processFile(String filename){
  // распознать файл из пути к нему
  Path file1 = Paths.get(filename);

  // открыть файл для записи
  try (BufferedWriter bw = new BufferedWriter(new
     OutputStreamWriter(Files.newOutputStream(file1)))) {
    // записать в файл...
  } catch (IOException e) {
    // обработать ошибку
  }

  // ...
  // еще раз открыть файл для чтения
  Path file2 = Paths.get(filename);
  if (!Files.isSameFile(file1, file2)) {
    // файл был заменен, обработать ошибку
  }

  try (BufferedReader br = new BufferedReader(new
     InputStreamReader(Files.newInputStream(file2)))) {
    String line;
    while ((line = br.readLine()) != null) {
      System.out.println(line);
    }
```

```
  } catch (IOException e) {
    // обработать ошибку
  }
}
```

К сожалению, в прикладном интерфейсе API языка Java не гарантируется, что в методе `isSameFile()` на самом деле производится проверка, является ли файл тем же самым. Так, в документации на прикладной интерфейс API версии Java 7 в отношении метода `isSameFile()` говорится следующее:

> "Если оба объекта типа `Path` равны, то данный метод возвращает логическое значение `true`, даже не проверяя, существует ли файл".

Это означает, что метод `isSameFile()` может просто проверить, одинаковы ли пути к обоим сравниваемым файлам. Но при этом он не в состоянии обнаружить, был ли файл по данному пути заменен другим файлом в промежутке между двумя последовательными операциями его открытия.

Решение, соответствующее принятым нормам (проверка с помощью нескольких атрибутов)

В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, проверяются даты создания и последнего видоизменения файлов с целью увеличить вероятность того, что файл, открывавшийся для чтения, окажется тем же самым, что и файл, открывавшийся для записи.

```
public void processFile(String filename) throws IOException{
  // распознать файл из пути к нему
  Path file1 = Paths.get(filename);
  BasicFileAttributes attr1 =
    Files.readAttributes(file1, BasicFileAttributes.class);
  FileTime creation1 = attr1.creationTime();
  FileTime modified1 = attr1.lastModifiedTime();

  // открыть файл для записи
  try (BufferedWriter bw = new BufferedWriter(new
      OutputStreamWriter(Files.newOutputStream(file1)))) {
    // записать в файл...
  } catch (IOException e) {
    // обработать ошибку
  }

  // еще раз открыть файл для чтения
  Path file2 = Paths.get(filename);
  BasicFileAttributes attr2 =
    Files.readAttributes(file2, BasicFileAttributes.class);
  FileTime creation2 = attr2.creationTime();
  FileTime modified2 = attr2.lastModifiedTime();
  if ( (!creation1.equals(creation2)) ||
       (!modified1.equals(modified2)) ) {
    // файл был заменен, обработать ошибку
  }

  try (BufferedReader br = new BufferedReader(new
      InputStreamReader(Files.newInputStream(file2)))){
```

```
    String line;
    while ((line = br.readLine()) != null) {
      System.out.println(line);
    }
  } catch (IOException e) {
    // обработать ошибку
  }
}
```

Несмотря на то что данное решение является достаточно безопасным, решительный злоумышленник, совершающий атаку, может создать символическую ссылку с теми же самыми датами создания и последнего видоизменения, что и у исходного файла. Кроме того, условие гонки типа "время проверки — время использования" (TOCTOU) может возникнуть в промежутке между моментами первоначального чтения атрибутов файла и его первого открытия. Аналогично, еще одно условие гонки типа "время проверки — время использования" может возникнуть в промежутке между моментами вторичного чтения атрибутов файла и его повторного открытия.

Решение, соответствующее принятым нормам (проверка с помощью атрибута `fileKey` по стандарту POSIX)

В средах, поддерживающих атрибут `fileKey`, применяется более надежный подход к проверке одинаковости атрибутов `fileKey` двух сравниваемых файлов. Атрибут `fileKey` представляет собой объект, который "однозначно определяет файл" [API 2013], как показано в представленном здесь решении, соответствующем принятым нормам защитного программирования на Java.

```
public void processFile(String filename) throws IOException{
  // распознать файл из пути к нему
  Path file1 = Paths.get(filename);
  BasicFileAttributes attr1 =
    Files.readAttributes(file1, BasicFileAttributes.class);
  Object key1 = attr1.fileKey();
  // открыть файл для записи
  try (BufferedWriter bw = new BufferedWriter(
    new OutputStreamWriter(Files.newOutputStream(file1)))) {
    // записать в файл...
  } catch (IOException e) {
    // обработать ошибку
  }

  // еще раз открыть файл для чтения
  Path file2 = Paths.get(filename);
  BasicFileAttributes attr2 =
    Files.readAttributes(file2, BasicFileAttributes.class);
  Object key2 = attr2.fileKey();

  if ( !key1.equals(key2) ) {
    System.out.println("File tampered with");
    // файл был заменен, обработать ошибку
  }

  try (BufferedReader br = new BufferedReader(
      new InputStreamReader(Files.newInputStream(file2)))) {
```

```
    String line;
    while ((line = br.readLine()) != null) {
      System.out.println(line);
    }
  } catch (IOException e) {
    // обработать ошибку
  }
}
```

Такой поход годится не для всех платформ. Например, в версии Windows 7 Enterprise Edition все атрибуты `fileKey` являются пустыми. Однозначность файлового ключа, возвращаемого методом `fileKey()`, гарантируется только в том случае, если файловая система и файлы остаются статическими. Например, идентификатор файла может повторно использоваться в файловой системе после его удаления. Как и в предыдущем совместимом решении, в промежутке между моментами первоначального чтения атрибутов файла и его первого открытия может возникнуть условие гонки типа "время проверки — время использования" (TOCTOU). А еще одно такое же условие гонки может возникнуть в промежутке между моментами вторичного чтения атрибутов файла и его повторного открытия.

Решение, соответствующее принятым нормам (применение класса `RandomAccessFile`)

Более совершенный подход заключается в том, чтобы избежать повторного открытия файла. В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, демонстрируется применение класса `RandomAccessFile`, который позволяет открывать файл как для чтения, так и для записи. Благодаря тому, что файл закрывается только автоматически в операторе `try` с ресурсами, исключается возникновение условия гонок. Следует, однако, иметь в виду, что в этом и других решениях, соответствующих принятым нормам защитного программирования на Java, метод `readLine()` применяется только ради наглядности примера. А о возможных слабых сторонах этого метода см. в рекомендации "MSC05-J. Не исчерпывайте область динамической памяти" из стандарта *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012].

```
public void processFile(String filename) throws IOException{
  // распознать файл из пути к нему
  try ( RandomAccessFile file = new RandomAccessFile(filename, "rw")) {
    // записать в файл...
    // вернуться в начало файла и прочитать его содержимое
    file.seek(0);
    String line;
    while ((line = file.readLine()) != null) {
      System.out.println(line);
    }
  }
}
```

Пример кода, не соответствующего принятым нормам (проверка размера файла)

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, предпринимается попытка убедиться, что открываемый файл содержит точно 1024 байта.

```
static long goodSize = 1024;

public void doSomethingWithFile(String filename) {
  long size = new File(filename).length();
  if (size != goodSize) {
    System.out.println("File has wrong size!");
    return;
  }

  try (BufferedReader br = new BufferedReader(
      newInputStreamReader(new FileInputStream(filename)))) {
    // ... обработать файл
  } catch (IOException e) {
    // обработать ошибку
  }
}
```

В приведенном выше коде может возникнуть условие гонки типа “время проверки — время использования” (TOCTOU) в промежутке между моментами проверки размера файла и его открытия. Если совершающий атаку злоумышленник заменит файл размером 1024 байта другим файлом в промежутке, когда может возникнуть данное условие гонки, то программа способна открыть фактически любой файл в обход проверки его размера.

Решение, соответствующее принятым нормам (проверка размера файла)

В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, для получения размера файла применяется метод `FileChannel.size()`. А поскольку этот метод применяется к объекту типа `FileInputStream` только после открытия файла, то данное решение исключает появление окна для гонок.

```
static long goodSize = 1024;

public void doSomethingWithFile(String filename) {
  try (FileInputStream in = new FileInputStream(filename);
       BufferedReader br = new BufferedReader(
                       new InputStreamReader(in))) {
    long size = in.getChannel().size();
    if (size != goodSize) {
      System.out.println("File has wrong size!");
      return;
    }

    String line;
    while ((line = br.readLine()) != null) {
      System.out.println(line);
    }
  } catch (IOException e) {
    // обработать ошибку
  }
}
```

Применимость

Совершающие атаки нередко используют уязвимости в защите, имеющие отношение к файлам, чтобы заставить программу получить доступ не к тому файлу. Для того чтобы предотвратить использование подобных уязвимостей в защите, требуется организовать надлежащее распознавание файлов.

Библиография

[API 2013] Class **java.io.File**
Interface **java.nio.file.Path**
Class **java.nio.file.Files**
Interface **java.nio.file.attribute.BasicFileAttributes**

[Long 2012] FIO00-J. Do not operate on files in shared directories

■ 28. Не присоединяйте значимость к порядковому значению, связанному с перечислением

Для перечислимых типов в языке Java имеется метод `ordinal()`, возвращающий числовую позицию каждой константы перечислимого типа в объявлении ее класса. В документации на прикладной интерфейс API языка Java [API 2013] по этому поводу говорится следующее:

> "Метод `public final int ordinal()` из обобщенного класса `Enum<E extends Enum<E>>` возвращает константу перечислимого типа (ее позицию в объявлении перечисления, где исходной константе присваивается нулевое порядковое значение). Большинство программистов вряд ли будут пользоваться этим методом. Ведь он предназначен для применения в сложных структурах данных на основе перечислений, включая `EnumSet` и `EnumMap`".

В §8.9 "Перечисления" спецификации языка Java (JLS) [JLS 2013] не предписывается применение метода `ordinal()` в программах. Но присоединение значимости к порядковому значению константы перечислимого типа, возвращаемому методом `ordinal()`, чревато ошибками, а следовательно, нежелательно в защитном программировании.

Пример кода, не соответствующего принятым нормам

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, объявляется перечисление `Hydrocarbon`, а его метод `ordinal()` служит для того, чтобы предоставить результат вызова метода `getNumberOfCarbons()`.

```
enum Hydrocarbon {
  METHANE, ETHANE, PROPANE, BUTANE, PENTANE,
  HEXANE, HEPTANE, OCTANE, NONANE, DECANE;
  public int getNumberOfCarbons() {
   return ordinal() + 1;
  }
}
```

И хотя код из данного примера, не соответствующего принятым нормам защитного программирования на Java, ведет себя так, как и предполагалось, сопровождать его совсем не просто. Так, если переставить константы в перечислении `Hydrocarbon`, метод `getNumberOfCarbons()` возвратит неверные значения. Более того, ввод дополнительной константы `BENZENE` в перечисление может нарушить инвариант, предполагаемый методом `getNumberOfCarbons()`, поскольку у бензола имеется шесть атомов углерода, а порядковое значение шесть уже присвоено гексану.

Решение, соответствующее принятым нормам

В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, константы перечислимого типа явно связываются с соответствующими целочисленными значениями, обозначающими количество атомов углерода.

```
enum Hydrocarbon {
  METHANE(1), ETHANE(2), PROPANE(3), BUTANE(4), PENTANE(5),
  HEXANE(6), BENZENE(6), HEPTANE(7), OCTANE(8), NONANE(9),
  DECANE(10);

  private final int numberOfCarbons;

  Hydrocarbon(int carbons) { this.numberOfCarbons = carbons; }

  public int getNumberOfCarbons() {
    return numberOfCarbons;
  }
}
```

Метод `ordinal()` больше не применяется в методе `getNumberOfCarbons()` для обнаружения количества атомов углерода, соответствующих каждому порядковому значению. С одним и тем же порядковым значением могут быть связаны разные константы, как показано на примерах констант `HEXANE` и `BENZENE`. Более того, в таком решении отсутствует всякая зависимость от порядка перечисления. Метод `getNumberOfCarbons()` будет и дальше действовать правильно, даже если переупорядочить перечисление.

Применимость

Когда порядок констант в перечислении стандартный, а дополнительные константы в него не вводятся, то вполне допустимо пользоваться порядковыми значениями, связанными с перечислимым типом. Например, применение порядковых значений допускается в приведенном ниже перечислимом типе. Как правило, применение порядковых значений для выведения целых значений затрудняет сопровождение программы и может привести к ошибкам в программе.

```
public enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
                  THURSDAY, FRIDAY, SATURDAY }
```

Библиография

[API 2013] Class **`Enum<E extends Enum<E>>`**

[Bloch 2008] Item 31, "Use Instance Fields Instead of Ordinals"

[JLS 2013] §8.9, "Enums"

29. Принимайте во внимание числовое продвижение типов

Числовое продвижение типов служит для преобразования операндов числового оператора к общему типу, позволяющему выполнить операцию. Когда в арифметических операторах употребляются операнды разного размера, то более узкие операнды продвигаются к типу более широких операндов.

Правила продвижения типов

В §5.6 "Числовые продвижения типов" спецификации JLS [JLS 2013] описываются следующие правила числового продвижения типов.

1. Если любые операнды относятся к ссылочному типу, то выполняется преобразование с распаковкой.
2. Если один из операндов относится к типу `double`, то другой операнд преобразуется в тип `double`.
3. Иначе если один из операндов относится к типу `float`, то другой операнд преобразуется в тип `float`.
4. Иначе если один из операндов относится к типу `long`, то другой операнд преобразуется в тип `long`.
5. Иначе оба операнда преобразуются в тип `int`.

Расширение преобразований, получающееся в результате числового продвижения типов, позволяет сохранить общую величину числа. Но продвижение типов, в котором операнды преобразуются из типа `int` в тип `float` или из типа `long` в тип `double`, могут привести к потере точности. (Подробнее об этом см. в рекомендации "NUM13-J. Избегайте потери точности при преобразовании примитивных целых значений в значения с плавающей точкой" из стандарта *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012].)

Подобные преобразования могут происходить при использовании операторов умножения и деления (**%**, *****, **/**), сложения и вычитания (**+**, **–**), сравнения (**<**, **>**, **<=**, **>=**), равенства (**==**, **!=**) и целочисленных поразрядных операторов (**&**, **|**, **^**).

Примеры

В следующем примере происходит продвижение к типу `double` перед применением арифметического оператора **+**:

```
int a = одно_значение;
double b = другое_значение;
double c = a + b;
```

В приведенном ниже примере кода значение переменной `b` сначала преобразуется в тип `int`, чтобы применить оператор **+** к операндам одного и того же типа. Затем результат операции `(a + b)` преобразуется в тип `float`, и наконец, применяется оператор сравнения.

```
int a = некоторое_значение;
char b = некоторый_символ;

if ((a + b) > 1.1f) {
  // сделать что-нибудь
}
```

Составные операторы

Если в составных выражениях употребляются операторы смешанных типов, то может произойти приведение типов. К числу составных операторов присваивания относятся операторы **+=**, **-=**, ***=**, **/=**, **&=**, **^=**, **%=**, **<<=**, **>>=**, **>>>=** и **|=**. В §15.26.2 "Составные операторы присваивания" спецификации JLS [JLS 2013] утверждается следующее:

> "Составное выражение присваивания в форме `E1 op= E2` равнозначно выражению `E1 = (T)((E1) op (E2))`, где `T` обозначает тип выражения `E1`, за исключением того, что тип выражения `E1` вычисляется только один раз".

Это означает, что в составном выражении присваивания производится неявное приведение типа результата вычисления к типу левого операнда. Если же операнды относятся к разным типам, то может произойти несколько преобразований. Так, если выражение `E1` относится к типу `int`, а выражение `E2` — к типу `long`, `float` или `double`, то сначала происходит расширение типа выражения `E1` до типа выражения `E2` (до выполнения оператора `op`), а затем следует сужение типа выражения `E2` обратно до типа `int` (после выполнения оператора `op`, но перед присваиванием).

Пример кода, не соответствующего принятым нормам (умножение)

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, значение переменной `big` типа `int` умножается на значение переменной `one` типа `float`.

```
int big = 1999999999;
float one = 1.0f;
// двоичная операция, в ходе которой происходит потеря
// точности вследствие неявного приведения типов
System.out.println(big * one);
```

В данном случае числовое продвижение типов требуется для того, чтобы продвинуть тип переменной `big` к типу `float`, прежде чем произойдет умножение. И в конечном итоге это приводит к потере точности, т.е. в данном примере кода в качестве результата умножения переменных выводится значение **`2.0E9`**, а не **`1.999999999E9`**. (См. рекомендацию "NUM13-J. Избегайте потери точности при преобразовании примитивных целых значений в значения с плавающей точкой" из стандарта *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012].)

Решение, соответствующее принятым нормам (умножение)

В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, вместо типа `float` употребляется тип `double` как более безопасное средство расширенного преобразования примитивных типов, происходящего в результате целочисленного продвижения типов.

```
int big = 1999999999;
double one = 1.0d; // объявить тип double вместо типа float
System.out.println(big * one);
```

Такое решение дает предполагаемый результат **1.999999999E9**, т.е. значение, получаемое при присваивании значения типа int значению типа double с предварительно выполняемым неявным приведением типов. Подробнее о смешении арифметических операций над целыми значениями и значениями с плавающей точкой см. в рекомендации 60 "Преобразуйте целые значения в значения с плавающей точкой для выполнения операций с плавающей точкой".

Пример кода, не соответствующего принятым нормам (сдвиг влево)

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, демонстрируется целочисленное продвижение типов, возникающее в результате выполнения поразрядного логического оператора ИЛИ.

```
byte[] b = new byte[4];
int result = 0;
for (int i = 0; i < 4; i++) {
  result = (result << 8) | b[i];
}
```

Каждый элемент байтового массива расширяется до 32 бит и дополняется знаком перед использованием в качестве операнда. Так, если бы этот элемент массива первоначально содержал значение **0xff**, то в конечном итоге он содержал бы значение **0xffffffff** [FindBugs 2008]. Таким образом, результирующее значение отличается от сцепления четырех элементов массива.

Решение, соответствующее принятым нормам (сдвиг влево)

В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, для достижения предполагаемого результата маскируются старшие 24 бита в значении элемента байтового массива.

```
byte[] b = new byte[4];
int result = 0;
for (int i = 0; i < 4; i++) {
  result = (result << 8) | (b[i] & 0xff);
}
```

Пример кода, не соответствующего принятым нормам (составная операция сложения и присваивания)

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, выполняется составная операция сложения и присваивания.

```
int x = 2147483642; // 0x7ffffffa
x += 1.0f; // переменная x содержит значение 2147483647
         // (0x7fffffff) после вычисления
```

Составная операция включает в себя значение типа int, содержащее слишком много битов, чтобы их можно было вместить в 23-разрядной мантиссе числового значения типа float, поддерживаемого в Java. Это приводит к расширяющему преобразованию типа int в тип float с потерей точности. Зачастую результирующее значение оказывается не тем, что ожидается.

Решение, соответствующее принятым нормам (составная операция сложения и присваивания)

Для целей защитного программирования следует избегать употребления любых составных операторов присваивания значений переменным типа `byte`, `short` или `char`. Следует также воздерживаться от применения более широкого операнда в правой части выражения. В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, все операнды относятся к поддерживаемому в Java типу `double`.

```
double x = 2147483642; // 0x7ffffffa
x += 1.0; // переменная x содержит значение 2147483643.0
          // (0x7ffffffb.0), как и предполагалось
```

Пример кода, не соответствующего принятым нормам (составная операция поразрядного сдвига и присваивания)

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, применяется составной оператор сдвига вправо с целью сдвинуть значение переменной `i` на один разряд.

```
short i = -1;
i >>>= 1;
```

К сожалению, значение переменной `i` остается прежним. Сначала значение переменной `i` продвигается к типу `int`. Это приводит к расширяющему преобразованию примитивных типов, а следовательно, к потере точности. Если коротко, то значение **`-1`** получает двоичное представление **`0xffff`**. В результате преобразования в тип `int` получается двоичное значение **`0xffffffff`**, которое затем сдвигается вправо на 1 бит, чтобы получилось двоичное значение **`0x7fffffff`**. Для того чтобы сохранить это значение обратно в переменной `i` типа `short`, в Java выполняется неявное сужающее преобразование типов, в ходе которого отбрасываются 16 старших битов. В конечном итоге получается то же самое двоичное значение **`0xffff`** или десятичное значение **`-1`**.

Решение, соответствующее принятым нормам (составная операция поразрядного сдвига и присваивания)

В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, составной оператор присваивания применяется к типу `int`, и для этого не требуется расширение и последующее сужение преобразования типов. Следовательно, переменная `i` получает значение **`0x7fffffff`**.

```
int i = -1;
i >>>= 1;
```

Применимость

Если не принять во внимание правила продвижения типов в целочисленных операндах, а также в операндах с плавающей точкой, то в конечном итоге это может привести к потере точности.

Библиография

[Bloch 2005]	Puzzle 9, "Tweedledum"
	Puzzle 31, "Ghost of Looper"
[Findbugs 2008]	"BIT: Bitwise OR of Signed Byte Value"
[JLS 2013]	§4.2.2, "Integer Operations"
	§5.6, "Numeric Promotions"
	§15.26.2, "Compound Assignment Operators"
[Long 2012]	NUM13-J. Avoid loss of precision when converting primitive integers to floating-point

■ 30. Активизируйте проверку типов в методах с переменным количеством аргументов во время компиляции

Методом с переменным количеством аргументов (или *varargs*) называется такой метод, который может принимать нефиксированное количество аргументов. Тем не менее у него должен быть хотя бы один фиксированный аргумент. При обработке вызова метода с переменным количеством аргументов компилятор Java проверяет типы всех его аргументов. При этом все эти аргументы должны соответствовать формальному типу переменного количества аргументов. Но проверка типов во время компиляции оказывается неэффективной, если в методе применяются параметры обобщенных типов или типа `Object` [Bloch 2008]. Для того чтобы активизировать строгую проверку типов в методах с переменным количеством аргументов во время компиляции, следует указать как можно более конкретный тип данных для параметров таких методов.

Пример кода, не соответствующего принятым нормам (тип `Object`)

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, ряд числовых значений суммируется в методе с переменным количеством аргументов типа `Object`. Следовательно, этот метод принимает произвольное сочетание параметров в виде разнотипных объектов. Такое объявление параметров метода редко употребляется как допустимое, хотя из этого правила имеются исключения, упоминаемые в разделе "Применимость" данной рекомендации.

```
double sum(Object... args) {
  double result = 0.0;
    for (Object arg : args) {
      if (arg instanceof Byte) {
        result += ((Byte) arg).byteValue();
      } else if (arg instanceof Short) {
        result += ((Short) arg).shortValue();
      } else if (arg instanceof Integer) {
        result += ((Integer) arg).intValue();
      } else if (arg instanceof Long) {
        result += ((Long) arg).longValue();
      } else if (arg instanceof Float) {
        result += ((Float) arg).floatValue();
      } else if (arg instanceof Double) {
```

```
        result += ((Double) arg).doubleValue();
      } else {
        throw new ClassCastException();
      }
    }
    return result;
}
```

Решение, соответствующее принятым нормам (тип `Number`)

В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, определяется тот же самый метод, но для объявления его аргументов используется тип `Number`. Это тип абстрактного класса, который является в достаточной степени обобщенным, чтобы охватить все числовые типы данных, но в то же время исключить нечисловые типы данных.

```
double sum(Number... args) {
  // ...
}
```

Пример кода, не соответствующего принятым нормам (обобщенный тип)

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, объявляется тот же самый метод с переменным количеством аргументов, но на этот раз — обобщенного типа. Этот метод принимает переменное количество параметров, причем все они относятся к *одному и тому же* типу объектов, хотя и могут представлять разнотипные объекты. Опять же, такое объявление параметров метода редко употребляется как допустимое.

```
<T> double sum(T... args) {
  // ...
}
```

Решение, соответствующее принятым нормам (обобщенный тип)

В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, определяется тот же самый метод, а для объявления его аргументов используется тип `Number`.

```
<T extends Number> double sum(T... args) {
  // ...
}
```

При объявлении параметров метода с переменным количеством аргументов следует как можно конкретнее определять их типы, избегая типа `Object` и неконкретности обобщенных типов. Переделка старых методов с параметрами в виде конечного массива на методы с переменным количеством параметров обобщенного типа не всегда оказывается целесообразной. Если, например, метод не принимает аргумент конкретного типа, то проверку его типа во время компиляции можно было бы заменить на переменное количество параметров обобщенного типа, чтобы метод был скомпилирован скорее чисто, чем правильно. Но это привело бы к ошибке во время выполнения [Bloch 2008].

Следует также иметь в виду, что автоупаковка препятствует строгой проверке примитивных типов и соответствующих им классов-оболочек во время компиляции. Например, в представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, выдается приведенное ниже предупреждение, хотя код действует так, как и предполагалось. Этим конкретным предупреждением компилятора можно благополучно пренебречь.

```
Java.java:10: warning: [unchecked] Possible heap pollution from
parameterized vararg type T
<T extends Number> double sum(T... args) {
```

Применимость

Опрометчивое употребление типов в методах с переменным количеством аргументов препятствует организации строгой проверки типов во время компиляции. Оно приводит к неоднозначности и делает менее удобочитаемым исходный код.

Сигнатуры методов с переменным количеством аргументов, в которых используется тип `Object` и неточные обобщенные типы, оказываются вполне пригодными в тех случаях, когда в теле метода отсутствуют операции приведения типов и автоупаковки, а его исходный код компилируется без ошибок. Рассмотрим следующий пример кода, корректно действующего для всех типов объектов и успешно выполняющего проверки типов:

```
<T> Collection<T> assembleCollection(T... args) {
  return new HashSet<T>(Arrays.asList(args));
}
```

В некоторых случаях для переменного числа аргументов приходится употреблять тип `Object`. Характерным тому примером служит метод `java.util.Formatter.format(String format, Object... args)`, способный отформатировать объекты любого типа. В этом случае автоматическое обнаружение объектов осуществляется напрямую.

Библиография

[Bloch 2008]	Item 42, "Use Varargs Judiciously"
[Steinberg 2008]	Using the Varargs Language Feature
[Oracle 2011b]	Varargs

■ 31. Не объявляйте открытыми и конечными константы, значения которых могут измениться в последующих выпусках программы

С помощью ключевого слова `final` можно указать значения констант, т.е. значения, которые не изменяются во время выполнения программы. Но константы, которые могут быть изменены в течение срока эксплуатации программы, не следует объявлять открытыми и конечными (`public final`). В спецификации JLS [JLS 2013] допускается встраивание значения любого открытого конечного поля в любую единицу компиляции, где осуществляется чтение этого поля. Следовательно, если при редактировании объявления класса в новой версии открытому конечному полю присваивается новое значение, то единицам компиляции, где это поле читается, может быть по-прежнему доступно старое значение до тех пор, пока они не

будут перекомпилированы. Подобное затруднение может, например, возникнуть в том случае, когда произойдет переход к последней версии обновленной сторонней библиотеки, а обращающийся к ней код не перекомпилирован.

Аналогичная ошибка может возникнут и в том случае, если объявляется статическая конечная (`static final`) ссылка на изменяемый объект. Подробнее об этом см. в рекомендации 73 "Не путайте неизменяемость ссылки и доступного по ссылке объекта".

Пример кода, не соответствующего принятым нормам

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, в классе `Foo` из пакета `Foo.java` объявляется поле, значение которого обозначает версию программы.

```
class Foo {
  public static final int VERSION = 1;
  // ...
}
```

А в дальнейшем доступ к этому осуществляется из класса `Bar` в отдельной единице компиляции (пакете `Bar.java`), как показано ниже.

```
class Bar {
  public static void main(String[] args) {
    System.out.println("You are using version " + Foo.VERSION);
  }
}
```

После компиляции и запуска программы на выполнение будет правильно выведено следующее сообщение:

```
You are using version 1
(Вы пользуетесь версией 1)
```

Но если разработчик заменит значение в поле `VERSION` на **2**, видоизменив и перекомпилировав пакет `Foo.java`, но забыв перекомпилировать заодно и пакет `Bar.java`, то программа неправильно выведет прежнее сообщение:

```
You are using version 1
```

Этот недостаток нетрудно исправить, перекомпилировав пакет `Bar.java`. Но имеется и лучшее решение.

Решение, соответствующее принятым нормам

В §13.4.9 "Конечные поля и константы" спецификации JLS [JLS 2013] говорится следующее:

> "Кроме истинных математических констант, переменные классов рекомендуется объявлять статическими и конечными (`static final`) в исходном коде лишь в самом крайнем случае. Если же требуется конечная переменная только для чтения, то ее лучше объявить закрытой и статической (`private static`), чтобы получить ее значение подходящим методом доступа".

В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, поле `version` объявляется в пакете `Foo.java` как закрытое и статическое (`private static`), а доступ к нему осуществляется методом `getVersion()`.

```
class Foo {
  private static int version = 1;
  public static final int getVersion() {
    return version;
  }

  // ...
}
```

В класс `Bar` из пакета `Bar.java` вносятся изменения, чтобы вызывать метод `getVersion()` и получать значение поля `version` из пакета `Foo.java`, как показано ниже.

```
class Bar {
  public static void main(String[] args) {
    System.out.println("You are using version " + Foo.getVersion());
  }
}
```

В данном решении значение закрытого поля `version` нельзя скопировать в класс `Bar` во время компиляции, и благодаря этому устраняется упомянутая выше программная ошибка. Следует, однако, иметь в виду, что подобное преобразование едва ли скажется на производительности, поскольку большинство динамических (JIT) генераторов кода в состоянии встроить метод `getVersion()` во время выполнения.

Применимость

Объявление значения типа `final`, изменяющегося в течение срока эксплуатации программы, может привести к неожиданным результатам. В §9.3, "Объявления полей и констант" спецификации JLS [JLS 2013] по этому поводу говорится следующее:

> "Каждое поле неявно объявляется в теле интерфейса как открытое, статическое и конечное (`public static final`). В объявлении такого поля дополнительно разрешается указывать любые или все модификаторы".

Следовательно, данная рекомендация не распространяется на поля, определяемые в интерфейсах. Очевидно, что если значение поля в интерфейсе изменится, то каждый класс, в котором реализуется и применяется этот интерфейс, должен быть перекомпилирован. Подробнее об этом см. в рекомендации 35 "Тщательно разрабатывайте интерфейсы, прежде чем их выпускать".

Данная рекомендация не распространяется на константы, объявляемые как перечислимые типа `enum`. А константы, значения которых вообще не изменяются в течение срока действия программы, могут быть объявлены как конечные (`final`). Например, математические константы в спецификации JLS рекомендуется объявлять открытыми.

Библиография

[JLS 2013] §4.12.4, "final Variables"
§8.3.1.1, "static Fields"
§9.3, "Field (Constant) Declarations"
§13.4.9, "final Fields and Constants"

■ 32. Избегайте циклических зависимостей пакетов

В книге *The Elements of Java™ Style* [Allen 2000] и стандарте *JPL Java Coding Standard* [Havelund 2009] указывается следующее требование: структура зависимостей пакета не должна содержать циклы. Это означает, что она должна быть такой же представительной, как и направленный ациклический граф (DAG). Исключение циклических зависимостей пакетов дает ряд преимуществ.

- **Тестирование и сопровождение исходного кода.** Циклические зависимости усиливают последствия от изменения и вставки "заплат" в исходный код. А уменьшение последствий от этих изменений упрощает тестирование и облегчает сопровождение исходного кода. Неспособность выполнять надлежащее тестирование из-за циклических зависимостей нередко служит источником уязвимостей в защите.
- **Повторное использование.** Циклические зависимости пакетов требуют синхронного выпуска и обновления пакетов. Но данное требование уменьшает возможность повторного использования пакетов.
- **Выпуски и сборки.** Исключение циклических зависимостей помогает также переносить разработку в среду, где поощряется разбиение на модули.
- **Развертывание.** Исключение циклических зависимостей пакетов сокращает их связывание. Сокращение связывания пакетов приводит к снижению ошибок во время выполнения, например, ошибки типа `ClassNotFoundError`. А это, в свою очередь, упрощает развертывание программ.

Пример кода, не соответствующего принятым нормам

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, содержатся пакеты `account` и `user`, состоящие из классов `AccountHolder`, `User` и `UserDetails` соответственно. В частности, класс `UserDetails` расширяет класс `AccountHolder`, поскольку пользователь выступает в роли владельца счета. Класс `AccountHolder` зависит от нестатического служебного метода, определенного в классе `User`. Аналогично класс `UserDetails` зависит от класса `AccountHolder`, расширяя его.

```
package account;
import user.User;
public class AccountHolder {
  private User user;
  public void setUser(User newUser) {user = newUser;}

  synchronized void depositFunds(String username, double amount) {
    // использовать служебный метод из класса User для проверки
    // существования пользователя с заданным именем
    if (user.exists(username)) {
      // внести сумму на счет
    }
  }

  protected double getBalance(String accountNumber) {
    // возвратить остаток на счете
    return 1.0;
  }
}
```

```
package user;
import account.AccountHolder;
public class UserDetails extends AccountHolder {
  public synchronized
  double getUserBalance(String accountNumber) {
    // использовать метод из класса AccountHolder для
    // получения остатка на счете
    return getBalance(accountNumber);
  }
}

public class User {
  public boolean exists(String username) {
    // проверить существование пользователя с заданным именем
    return true; // такой пользователь существует
  }
}
```

Решение, соответствующее принятым нормам

Тесное связывание классов в двух пакетах можно ослабить, внедрив интерфейс `BankApplication` в третий пакет `bank`. Циклическая зависимость пакетов исключается гарантией того, что класс `AccountHolder` не зависит от класса `User`, а вместо этого опирается на интерфейс, импортируя пакет `bank`, но не реализуя интерфейс `BankApplication`.

В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, подобные функциональные возможности достигаются введением параметра интерфейса типа `BankApplication` в метод `depositFunds()`. Такое решение дает классу `AccountHolder` надежный контракт с пакетом `bank`. Кроме того, в классе `UserDetails` предоставляются конкретные реализации методов из интерфейса `BankApplication` и в то же время наследуются другие методы из класса `AccountHolder`.

```
package bank;
public interface BankApplication {
  void depositFunds(BankApplication ba, String username, double amount);
  double getBalance(String accountNumber);
  double getUserBalance(String accountNumber);
  boolean exists(String username);
}

package account;
import bank.BankApplication; // импортировать из третьего пакета
class AccountHolder {
  private BankApplication ba;
  public void setBankApplication(BankApplication newBA) {
    ba = newBA;
  }

  public synchronized void depositFunds(BankApplication ba,
       String username, double amount) {
    // использовать служебный метод из класса UserDetails для
    // проверки существования пользователя по заданному имени
    if (ba.exists(username)) {
      // внести сумму на счет
    }
  }
```

```
  public double getBalance(String accountNumber) {
    // возвратить остаток на счете
    return 1.0;
  }
}

package user;
import account.AccountHolder; // односторонняя зависимость
import bank.BankApplication;  // импортировать из третьего пакета
public class UserDetails extends AccountHolder
      implements BankApplication {
  public synchronized double getUserBalance(String accountNumber) {
    // использовать метод из класса AccountHolder для получения
    // остатка на счете
    return getBalance(accountNumber);
  }

  public boolean exists(String username) {
    // проверить существование пользователя по заданному имени
    return true;
  }
}
```

Оказывается, что интерфейс `BankApplication` содержит лишние методы, в том числе `depositFunds()` и `getBalance()`. А поскольку эти методы присутствуют в данном интерфейсе, то если они переопределяются в подклассе, то в суперклассе остается возможность для внутреннего вызова методов из подкласса по принципу полиморфизма. Например, метод `ba.getBalance()` можно вызвать с его переопределенной реализацией из класса `UserDetails`. Одно из последствий такого решения состоит в том, что методы, объявленные в интерфейсе, должны быть непременно открытыми в тех классах, где они определяются.

Применимость

Циклические зависимости пакетов могут стать причиной прочности формируемых сборок. Уязвимость защиты в одном пакете может легко проникнуть в другие пакеты.

Библиография

[Allen 2000] *The Elements of Java™ Style*

[Havelund 2009] JPL Coding Standard, Version 1.1

[Knoernschild 2002] Chapter 1, "OO Principles and Patterns"

■ 33. Отдавайте предпочтение определяемым пользователем исключениям над более общими типами исключений

Исключение перехватываются по его типу, и поэтому их лучше всего определять для конкретных целей, вместо того чтобы использовать общие типы исключений для многих целей. Генерирование исключений общих типов затрудняет понимание и сопровождение исходного кода, а также нивелирует бо́льшую часть преимуществ механизма обработки исключений в Java.

Пример кода, не соответствующего принятым нормам

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, предпринимается попытка провести различие между разными видами поведения исключений, исходя из анализа сообщения об исключении. Так, если в методе `doSomething()` генерируется исключение или ошибка типа, относящегося к подклассу, производному от класса `Throwable`, то в операторе `switch` можно выбрать конкретный вариант для дальнейшего выполнения кода обработки исключений и ошибок. Например, если получается сообщение об исключении "Файл не найден", то в коде обработки исключений выполняется соответствующее действие.

```
try {
  doSomething();
} catch (Throwable e) {
  String msg = e.getMessage();
  switch (msg) {
    case "file not found":
      // обработать ошибку
      break;
    case "connection timeout":
      // обработать ошибку
      break;
    case "security violation":
      // обработать ошибку
      break;
    default: throw e;
  }
}
```

Тем не менее любое изменение в литералах, используемых в сообщении об исключении, приведет к нарушению нормальной работы кода. Допустим, что выполняется следующая строка кода:

```
throw new Exception("cannot find file");
```

Генерируемое в ней исключение должно быть обработано в первой ветви `case` оператора `switch`. Но вместо этого исключение будет сгенерировано повторно, поскольку символьная строка сообщения не совпадает ни с одним из условий в ветвях `case` оператора `switch`.

Более того, исключения могут быть сгенерированы и без уведомляющих о них сообщений. Данный пример кода, не соответствующего принятым нормам защитного программирования на Java, подпадает под действие правила обработки исключений типа `ERR08-EX0` из рекомендации "ERR08-J. Не перехватывайте исключение типа `NullPointerException` и любые его предшественники" из стандарта *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012], поскольку общие исключения не только перехватываются в нем, но и генерируются повторно.

Решение, соответствующее принятым нормам

В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, применяются конкретные типы исключений, а там, где требуется, определяются новые типы исключений специального назначения.

```
public class TimeoutException extends Exception {
  TimeoutException () {
```

```
    super();
  }
  TimeoutException (String msg) {
    super(msg);
  }
}

// ...

try {
  doSomething();
} catch (FileNotFoundException e) {
  // обработать ошибку
} catch (TimeoutException te) {
  // обработать ошибку
} catch (SecurityException se) {
  // обработать ошибку
}
```

Применимость

Исключения служат для обработки исключительных ситуаций. Если исключение не перехватывается, то выполнение программы прекращается. Исключение, перехваченное неправильно или не на том уровне восстановления работоспособности кода, зачастую приводит к неверному поведению.

Библиография

[JLS 2013] Chapter 11, "Exceptions"

[Long 2012] ERR08-J. Do not catch NullPointerException or any of its ancestors

■ 34. Старайтесь изящно исправлять системные ошибки

По поводу исключений в §11.1.1 "Виды исключений" спецификации JLS [JLS 2013] говорится следующее:

> "Непроверяемые исключения относятся к классу `RuntimeException` и его подклассам, а также к классу `Error` и его подклассам. Все остальные исключения относятся к классам проверяемых исключений".

Классы непроверяемых исключений не подвергаются проверке во время компиляции, поскольку на данной стадии очень трудно учесть все исключительные ситуации, тогда как восстановить работоспособность кода нередко оказывается затруднительно, а то и вообще невозможно. Но даже в том случае, если восстановить работоспособность кода невозможно, виртуальная машина Java (JVM) предоставляет изящный выход из этого затруднительного положения и возможность хотя бы зарегистрировать ошибку. Такая возможность появляется при использовании блока `try-catch`, в котором перехватывается исключение типа `Throwable`. Перехват исключения типа `Throwable` разрешается и в том случае, если в коде требуется исключить утечку потенциально уязвимой информации. Во всех остальных случаях

перехват исключения типа `Throwable` не рекомендуется, поскольку он затрудняет обработку конкретных исключений. А в тех случаях, когда могут быть выполнены операции очистки вроде освобождения системных ресурсов, в коде должен быть использован блок `finally` или оператор `try` с ресурсами для освобождения ресурсов.

Перехват исключения типа `Throwable` вообще запрещается в рекомендации "ERR08-J. Не перехватывайте исключение типа `NullPointerException` и любые его предшественники" из стандарта *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012]. Но он все же разрешается, если организуется фильтрация исключений по правилу обработки исключений типа `ERR08-EX0` из данной рекомендации.

Пример кода, не соответствующего принятым нормам

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, ошибка типа `StackOverflowError` генерируется в результате бесконечной рекурсии. Ведь в противном случае наступает исчерпание оперативной памяти, выделяемой под стек, а следовательно, и отказ в обслуживании.

```
public class StackOverflow {
  public static void main(String[] args) {
    infiniteRun();
    // ...
  }

  private static void infiniteRun() {
    infiniteRun();
  }
}
```

Решение, соответствующее принятым нормам

В представленном здесь решении, соответствующем принятым нормам защитного программирования на Java, демонстрируется применение блока `try-catch` для перехвата ошибок типа `java.lang.Error` или исключений типа `java.lang.Throwable`. В блоке `try-catch` может быть сделана длинная запись в журнале регистрации, а затем предприняты попытки освободить главные системные ресурсы в блоке `finally`.

```
public class StackOverflow {
  public static void main(String[] args) {
    try {
      infiniteRun();
    } catch (Throwable t) {
      // передать управление обработчику исключений
    } finally {
      // освободить кеш и ресурсы
    }
    // ...
  }

  private static void infiniteRun() {
    infiniteRun();
  }
}
```

Следует, однако, иметь в виду, что код передачи управления обработчику исключений должен работать корректно в условиях ограниченного объема оперативной памяти, поскольку стек или динамическая область памяти (так называемая "куча") оказывается близкой к полному исчерпанию. Во избежание подобных ситуаций имеет смысл заблаговременно зарезервировать память, специально предназначенную для обработчика исключений, связанных с исчерпанием оперативной памяти.

Следует также иметь в виду, что в данном решении исключение типа `Throwable` перехватывается при попытке обработать ошибку. И это согласуется с правилом обработки исключений типа `ERR08-EX2` из рекомендации "ERR08-J. Не перехватывайте исключение типа `NullPointerException` и любые его предшественники" из стандарта *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012].

Применимость

Если предоставить системе возможность внезапно прекратить выполнение программы на Java, то может возникнуть уязвимость к нарушению безопасности типа отказа в обслуживании. Если же оперативная память исчерпывается, то, скорее всего, некоторые данные программы оказываются в неопределенном состоянии. Поэтому данный процесс лучше всего перезапустить. Если же предпринять попытку продолжить выполнение кода, то в качестве эффективного обходного приема можно порекомендовать сокращение количества потоков. Такая мера может оказаться действенной в подобных случаях, поскольку в потоках нередко возникают утечки памяти, а для дальнейшего их существования может потребоваться дополнительный объем оперативной памяти. Для обработки ошибок типа `OutOfMemoryError` в потоках могут быть использованы методы `Thread.setUncaughtExceptionHandler()` и `ThreadGroup.uncaughtException()`.

Библиография

[JLS 2013]	§11.2, "Compile-Time Checking of Exceptions"
[Kalinovsky 2004]	Chapter 16, "Intercepting Control Flow: Intercepting System Errors"
[Long 2012]	ERR08-J. Do not catch NullPointerException or any of its ancestors

■ 35. Тщательно разрабатывайте интерфейсы, прежде чем их выпускать

Интерфейсы служат для группирования всех методов, которые класс обязуется сделать открытыми. В частности, классы, реализующие интерфейс, обязаны предоставить конкретные реализации всех его методов. Интерфейсы являются неотъемлемой составляющей большинства общедоступных прикладных интерфейсов API. После их выпуска очень трудно устранить обнаруженные в них недостатки, не нарушив исходный код, реализующий прежние их версии. Ниже перечислены некоторые последствия некачественной разработки интерфейсов.

- Изменения, вносимые в интерфейс с целью устранить обнаруженные в нем ошибки, могут серьезно нарушить контракты с классами, реализующими этот интерфейс. Например, устранение ошибки в последующей версии интерфейса может повлечь за собой видоизменения в несвязанном с ним интерфейсе, который теперь придется реализовать клиенту. Но клиенту может быть запрещено устранять ошибку, поскольку новый интерфейс может наложить на него дополнительное бремя своей реализации.

- Средства реализации могут предоставить своим клиентам исходные или скелетные реализации методов интерфейса для их последующего расширения. Но такой код может отрицательно сказаться на поведении подклассов. Следовательно, в отсутствие подобных исходных реализаций подклассы должны предоставить фиктивные реализации, стимулирующие формирование среды разработки, где нередки комментарии вроде "игнорировать этот код и ничего не делать". Такой код может быть вообще не протестирован.

Если обнаруживается уязвимость в защите общедоступного прикладного интерфейса API (см., например, обсуждение методов из класса `ThreadGroup` в рекомендации "THI01-J. Не вызывайте методы из класса `ThreadGroup`" из стандарта *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012]), то она будет сохраняться в течение всего срока действия данного прикладного интерфейса API, оказывая отрицательное воздействие на безопасность любого приложения или библиотеки, в котором этот прикладной интерфейс применяется. И даже после полного или частичного устранения данной уязвимости в защите общедоступного прикладного интерфейса API его небезопасная версия будет по-прежнему применяться в приложениях и библиотеках до тех пор, пока они не будут сами обновлены.

Пример кода, не соответствующего принятым нормам

В данном примере кода, не соответствующего принятым нормам защитного программирования на Java, интерфейс `User` закреплен со следующими двумя методами: `authenticate()` и `subscribe()`. А в дальнейшем поставщики предоставляют бесплатно выпускаемую услугу, не опирающуюся на аутентификацию.

```
public interface User {
  boolean authenticate(String username, char[] password);
  void subscribe(int noOfDays);
  // внедрено после открытого выпуска класса
  void freeService();
}
```

К сожалению, ввод метода `freeService()` нарушает весь клиентский код, реализующий интерфейс. Более того, средствам реализации, стремящимся воспользоваться только методом `freeService()`, придется предоставить и два других метода, а это "засоряет" прикладной интерфейс API по описанным выше причинам.

Пример кода, не соответствующего принятым нормам

Другой замысел состоит в том, чтобы отдать предпочтение абстрактным классам для решения задачи постоянного развития констант. Но для этого придется пожертвовать гибкостью интерфейсов, благодаря которой в классе можно реализовать несколько интерфейсов, но расширить только один класс. К числу примечательных шаблонов для решения подобной задачи относится распространение поставщиком абстрактного скелетного класса, реализующего развивающийся интерфейс. Такой скелетный класс может выборочно реализовывать одни методы и принудительно расширять классы для предоставления конкретных реализаций других методов. Так, если в интерфейс вводится новый метод, скелетный класс может предоставить неабстрактную по умолчанию реализацию метода, который может быть дополнительно переопределен в расширяющем классе. И такой скелетный класс демонстрируется в приведенном ниже примере кода, не соответствующего принятым нормам защитного программирования на Java.

```
public interface User {
  boolean authenticate(String username, char[] password);
  void subscribe(int noOfDays);
  void freeService(); // внедрено после открытого выпуска
                      // прикладного интерфейса API
}

abstract class SkeletalUser implements User {
  public abstract boolean authenticate(String username,
      char[] password);
  public abstract void subscribe(int noOfDays);
  public void freeService() {
    // добавить позднее, предоставить реализацию
    // и еще раз выпустить класс
  }
}

class Client extends SkeletalUser {
  // реализовать методы authenticate() и subscribe(),
  // но не метод freeService()
}
```

Несмотря на всю пользу от такого шаблона, он может оказаться небезопасным, поскольку поставщик, не имеющий представления об исходном коде расширяющего класса, может выбрать реализацию, вносящую уязвимости в защиту клиентского прикладного интерфейса API.

Решение, соответствующее принятым нормам (разбиение на модули)

Более совершенное проектное решение состоит в том, чтобы заранее предусмотреть перспективное развитие предоставляемых услуг. Базовые функциональные возможности должны быть реализованы в интерфейсе `User`. В качестве их расширения в данном случае может потребоваться только срочная услуга. И для того чтобы воспользоваться новой бесплатной услугой, в существующем классе может быть затем выбрано одно из двух: реализация нового интерфейса `FreeUser` или полное игнорирование этой услуги.

```
public interface User {
  boolean authenticate(String username, char[] password);
}

public interface PremiumUser extends User {
  void subscribe(int noOfDays);
}

public interface FreeUser {
  void freeService();
}
```

Решение, соответствующее принятым нормам (превращение нового метода в неиспользуемый)

Другое решение, соответствующее принятым нормам защитного программирования на Java, состоит в том, чтобы сгенерировать исключение в новом методе `freeService()`, определенном в реализующем его подклассе.

```
class Client implements User {
  public void freeService() {
    throw new AbstractMethodError();
  }
}
```

Решение, соответствующее принятым нормам (делегирование реализации новых методов подклассам)

Еще одно возможное, хотя и менее гибкое решение, соответствующее принятым нормам защитного программирования на Java, состоит в том, чтобы делегировать реализацию метода подклассам, производным от класса, в котором реализуется базовый интерфейс.

```
abstract class Client implements User {
  public abstract void freeService();
  // делегировать реализацию нового метода подклассам
  // Другие конкретные реализации
}
```

Применимость

Если не опубликовать устойчивые, безукоризненные интерфейсы, то могут быть нарушены контракты с реализующими их классами, "засорен" прикладной интерфейс API, а возможно, и внесены уязвимости в защиту реализующих их классов.

Библиография

[Bloch 2008] Item 18, "Prefer Interfaces to Abstract Classes"

[Long 2012] THI01-J. Do not invoke ThreadGroup methods

■ 36. Пишите код, удобный для "сборки мусора"

Система "сборки мусора" в Java предоставляет значительные преимущества по сравнению с другими языками программирования, где такая система отсутствует. Система "сборки мусора" предназначена для автоматического освобождения недоступной области памяти и предотвращения утечек памяти. Несмотря на то что такая система вполне способна справиться с подобной задачей, злоумышленник может начать атаку типа отказа в обслуживании на систему "сборки мусора", внедрив, в частности, код для аномального выделения динамической области памяти или чрезмерно продолжительного удерживания объектов в оперативной памяти. Например, в некоторых версиях системы "сборки мусора" может потребоваться остановка всех исполняющихся процессов для своевременной обработки входящих запросов на выделение оперативной памяти, приводящее к повышению интенсивности операций по управлению динамической областью памяти. И в этом случае резко падает производительность системы.

В частности, системы реального времени уязвимы к более изощренной атаке типа отказа в обслуживании, проникающей в систему путем кражи циклов центрального процессора (ЦП) и медленно приводящей в постепенному исчерпанию оперативной памяти. Совершающий атаку злоумышленник может распределить оперативную память таким образом, чтобы резко увеличить потребление системных ресурсов (например, ЦП, энергии заряда аккумуляторной

батареи и оперативной памяти), не вызывая ошибку типа `OutOfMemoryError`. Написание кода, удобного для "сборки мусора", ограничивает многие пути для проникновения подобных атак в систему.

Применяйте неизменяемые объекты с коротким сроком действия

Начиная с версии JDK 1.2 затраты на выделение памяти в системе "сборки мусора разных поколений" сокращены, как правило, до уровня, ниже чем в языке C или C++. Сокращение подобных затрат в системе "сборки мусора разных поколений" достигается благодаря группированию объектов по отдельным поколениям, причем более *молодое поколение* состоит из объектов с коротким сроком действия. Система "сборки мусора" освобождает оперативную память от молодого поколения уже недействующих объектов [Oracle 2010a]. Усовершенствованные алгоритмы "сборки мусора" позволяют сократить затраты на "сборку мусора" пропорционально количеству действующих объектов в молодом поколении, а не количеству объектов, для которых была выделена оперативная память, начиная с момента последней "сборки мусора".

Следует, однако, иметь в виду, что объекты из молодого поколения, долго сохраняющиеся в оперативной памяти, считаются *долгоживущими* и переводятся в разряд *долгоживущего поколения*. Лишь немногие объекты из молодого поколения продолжают существовать до следующего цикла "сборки мусора", а остальные подготавливаются к "сборке мусора" в предстоящем цикле [Oracle 2010a].

Благодаря системе "сборки мусора разных поколений" применение неизменяемых объектов с коротким сроком действия, как правило, оказывается более эффективным, чем изменяемых объектов с длительным сроком действия, в том числе и пула объектов. Исключение пула объектов повышает эффективность системы "сборки мусора". Пулы объектов влекут за собой дополнительные затраты и риски, поскольку они способны затруднить синхронизацию и могут потребовать явного управления процессом освобождения оперативной памяти от объектов, а также вызвать осложнения, связанные с появлением висячих указателей.

Кроме того, определение оптимального объема оперативной памяти для резервирования пула объектов может быть затруднено, особенно это касается ответственного кода. Поэтому применение изменяемых объектов с длительным сроком действия остается приемлемым в тех случаях, когда выделение оперативной памяти для объектов оказывается особенно затратным (например, при соединении нескольких таблиц из разных баз данных). Аналогично пулы объектов являются подходящим проектным решением, когда объекты представляют ограниченные ресурсы, например, пулы потоков и соединений с базой данных.

Избегайте крупных объектов

Выделение оперативной памяти для крупных объектов является затратной операцией отчасти потому, что затраты на инициализацию их полей оказываются пропорциональными их размерам. Кроме того, частое выделение оперативной памяти для крупных объектов разных размеров может вызвать их фрагментацию или затруднить выполнение уплотняющих операций "сборки мусора".

Не вызывайте систему "сборки мусора" явным образом

Систему "сборки мусора" можно вызвать явным образом с помощью метода `System.gc()`. И хотя в документации говорится, что при вызове данного метода выполняется "сборка

мусора", в то же время никак не гарантируется, что "сборка мусора" вообще будет выполнена и когда именно это произойдет. На самом деле вызов метода `System.gc()` позволяет лишь *предположить*, что "сборка мусора" будет выполнена впоследствии. А виртуальная машина JVM может вполне проигнорировать такое предположение.

Таким образом, *безответственное использование* такой возможности способно *значительно понизить* производительность системы, если "сборка мусора" будет начинаться в неподходящие моменты, а не ожидать удобных моментов, когда можно будет благополучно произвести "сборку мусора" без существенного вмешательства в ход выполнения программы.

В виртуальной машине Java Hotspot VM, применяемой по умолчанию, начиная с версии JDK 1.2, вызов метода `System.gc()` принуждает к явной "сборке мусора". Подобные вызовы могут быть глубоко скрыты в недрах библиотек, что затрудняет их отслеживание. Для того чтобы проигнорировать вызов системы "сборки мусора" в подобных случаях, следует использовать параметр командной строки **`-XX:+DisableExplicitGC`**. А во избежание длительных приостановок на время выполнения полной "сборки мусора" может быть запущен менее ответственный параллельный цикл с помощью параметра командной строки **`-XX:ExplicitGC InvokedConcurrent`**.

Применимость

Злоупотребление системой "сборки мусора" может привести к значительному снижению производительности. И это обстоятельство может быть использовано для атаки типа отказа в обслуживании. Уязвимость GERONIMO-4574 в веб-серверах Apache Geronimo и Tomcat, о которой сообщалось в марте 2009 года, возникла в результате того, что объекты данных, предназначавшиеся для обработки средствами класса `PolicyContext`, устанавливались в потоке и вообще не освобождались, и в конечном итоге они оставались в оперативной памяти дольше, чем требовалось.

Когда приложение проходит несколько стадий, включая инициализацию и подготовку к работе, в промежутках между этими стадиями может потребоваться уплотнение динамической области памяти. В подобных случаях может быть вызван метод `System.gc()`, при условии, что в промежутках между стадиями наступает подходящий момент, не отмеченный никакими примечательными событиями.

Библиография

[API 2013]	Class **`System`**
[Bloch 2008]	Item 6, "Eliminate Obsolete Object References"
[Coomes 2007]	"Garbage Collection Concepts and Programming Tips"
[Goetz 2004]	Java Theory and Practice: Garbage Collection and Performance
[Lo 2005]	"Security Issues in Garbage Collection"
[Long 2012]	OBJ05-J. Defensively copy private mutable class members before returning their references
	OBJ06-J. Defensively copy mutable inputs and mutable internal components
[Oracle 2010a]	Java SE 6 HotSpot™ Virtual Machine Garbage Collection Tuning