



HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

Лекция 10

Логирование и дженерики

Программирование на языке Java

Роман Гуров

ВШЭ БИ 2021





Логирование

Логирование (рус. журналирование) – автоматическая запись в хронологическом порядке информации о событиях, происходящих в рамках какого-либо процесса с некоторым объектом

Печать исключения на экран уже предоставляет нам некоторую информацию о произошедшей ошибке

Стек вызовов исключения позволяет локализовать конкретное место возникновения ошибки, но не дает никакой информации о том, что происходило до этой самой ошибки

Хочется иметь некоторый протокол событий, которые происходили в программе, чтобы в будущем всегда была возможность проанализировать процесс исполнения

```
Task with id 12551 started
task_id 12551: Validating config...
task_id 12551: Calculating initial solution...
task_id 12551: Optimizing iteration 1...
task_id 12551: Optimizing iteration 2...
task_id 12551: Optimizing iteration 3...
Task with id 12551 failed by exceeding the time limit
```



Логирование

Самый простой способ это сделать – использовать в коде в каждом желаемом месте `System.out.println` с соответствующим сообщением

Но это не гибко – даже банальное добавление времени к каждой записи лога потребует от разработчика дописывать его к каждой из них руками

Так как все пожелания для логирования примерно стандартны, Java предоставляет своё готовое решение

Все связанные с этим классы лежат в пакете `java.util.logging`

```
package ru.hse.lecture10;  
  
import java.util.logging.*;
```

Logger

Основное класс – Logger

Получить объект этого класса можно вызовом статического метода `Logger.getLogger(<имя логгера>)`

```
package ru.hse.lecture10;
import java.util.logging.*;

public class MyHotNewClass {
    private static final Logger LOGGER = Logger.getLogger(MyHotNewClass.class.getName());
}
```

`getName()` вернёт
"ru.hse.lecture10.MyHotNewClass"

Стандартная практика – каждый класс заводит свой собственный логгер и хранит его в приватном финальном статическом поле

Но как им пользоваться?

Логирование с помощью Logger

Основной метод объекта Logger – log

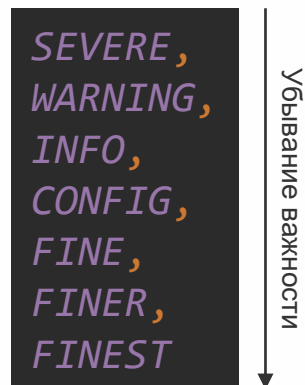
```
public class Main {  
    private static final Logger LOGGER = Logger.getLogger(Main.class.getName());  
  
    public static void main(String[] args) {  
        LOGGER.log(Level.INFO, "Main function logs itself!");  
    }  
}
```

```
Feb 14, 2022 3:23:09 AM ru.hse.lecture10.Main main  
INFO: Main function logs itself!  
  
Process finished with exit code 0
```

log принимает уровень важности сообщения и его текст

Уровни “важности”

Логгер позволяет разделить все сообщения по уровню необходимости их отображения



SEVERE предназначен для серьезных ошибок, очевидно важнее всех остальных

INFO — просто информация о чём-то, явно менее важна, чем предупреждение или ошибка

FINE и ниже — для очень подробных деталей, которые обычно вообще не хотелось бы видеть

Логгер можно настроить *игнорировать сообщения*, уровень которых *ниже* некоторого заданного

Уровни “важности”

Логгер можно настроить *игнорировать сообщения*, уровень которых *ниже* некоторого заданного

```
public static void main(String[] args) {  
    LOGGER.log(Level.SEVERE, "death");  
    LOGGER.log(Level.WARNING, "i warn you");  
    LOGGER.log(Level.INFO, "jfyi");  
    LOGGER.log(Level.FINEST, "All the finest wines improve with age");  
}
```

```
2022-02-14 03:52:54 SEVERE ru.hse.lecture10.Main main: death  
2022-02-14 03:52:54 WARNING ru.hse.lecture10.Main main: i warn you  
2022-02-14 03:52:54 INFO ru.hse.lecture10.Main main: jfyi
```

Изменить уровень логгера можно методом `setLevel`

Также, для всех уровней есть свои методы

```
LOGGER.setLevel(Level.SEVERE);  
LOGGER.severe("death");  
LOGGER.warning("i warn you");  
LOGGER.info("jfyi");  
LOGGER.finest("All the finest wines improve with age");
```

```
2022-02-14 04:05:04 SEVERE ru.hse.lecture10.Main main: death  
  
Process finished with exit code 0
```



Логирование динамических данных

Как залогировать динамические данные, такие как значение переменной?

Самое очевидное решение, которое нам уже знакомо – конкатенация строк:

```
LOGGER.log(Level.FINEST, "Current value of x is " + x);
```

Но получается так, что для игнорируемого лога всё равно будет вычислена и создана итоговая строка
Это может быть *очень медленно*

Поэтому, логгер умеет форматировать строки сам:

```
LOGGER.log(Level.FINEST, "Current value of x is {0}", x);
```


Вместо {0} в строку подставится переданное значение параметра

Для нескольких параметров придется передать их в массиве:

```
LOGGER.log(Level.FINEST,  
    "Current point coordinates are ({0}, {1})",  
    new Object[] {x, y});
```

Также, есть специальная перегрузка, печатающая исключение в привычном нам формате:

```
LOGGER.log(Level.SEVERE, "Task failed with unexpected exception", ex);
```

Куда уходят логи?

В консоль или в файл? А может что-то даже более хитрое?

Логгер даёт полную свободу в этом выборе

Сам логгер не печатает сообщение никуда,
он просто передаёт его своим обработчикам – наследникам `java.util.logging.Handler`

Есть три стандартных хэндлера, предоставляемых Java:

- `ConsoleHandler` – пишет лог прямо в консоль
- `FileHandler` – пишет лог в указанный файл
- `SocketHandler` – отправляет лог по сети

У каждого хэндлера тоже есть свой уровень логирования, по которому он так же отшивает нерелевантные записи

Чтобы логгер передавал сообщения вашему хэндлеру, хэндлер надо «привязать» к логгеру методом `addHandler`

```
LOGGER.addHandler(new FileHandler("MyProg.log"));  
LOGGER.severe("i'll go to a file too");
```



Формат сообщения

К сообщению лога добавляется ещё куча информации о времени, месте, и многом другом

Кто за это отвечает?

Перед отправкой сообщения в пункт назначения, хэндлер превращает его в строку с помощью форматировщика – наследника `java.util.logging.Formatter`

Таких имеется два:

- `SimpleFormatter` – простой человекочитаемый вид (можно конфигурировать)
- `XMLFormatter` – формат XML, удобный для обработки машиной

```
Handler fileHandler = new FileHandler("MyProg.log");
fileHandler.setFormatter(new SimpleFormatter());
LOGGER.addHandler(fileHandler);
```

Настройка логирования через файл

Не обязательно прописывать все настройки логгера прямо в коде, удобно делать это через отдельный конфигурационный файл

```
package ru.hse.lecture10;

import java.util.logging.*;

public class MyHotNewClass {
    private static final Logger LOGGER = Logger.getLogger(Main.class.getName());

    public static void main(String[] args) {
        LOGGER.fine("Program started");
        try {
            randomFailingAlgorithm();
        }
        catch (IllegalStateException e) {
            LOGGER.log(Level.SEVERE, "Exception caught", e);
            System.exit(2);
        }
        LOGGER.fine("Finished successfully");
    }

    private static void randomFailingAlgorithm() {
        double randomNumber = Math.random();
        LOGGER.log(Level.FINE, "Generated random number: {0}", randomNumber);
        if (randomNumber < 0.5) {
            throw new IllegalStateException("Invalid phase of the Moon");
        }
    }
}
```

Файл logging.properties:

```
# To use this config start JVM with parameter:
# -Djava.util.logging.config.file=logging.properties

.level=ALL
.handlers=java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level=ALL
```



Финал

А какой хэндлер писал нам сообщения всё это время? Мы ведь его не создавали

На самом деле, у логов есть иерархия

После передачи сообщения своим хэндлерам, логгер начинает передавать их хэндлерам своего родителя, потом родителя родителя, и так по цепочке до самого конца

Родитель – ближайший существующий логгер, имеющий более общее имя:
`ru.hse.lecture10.MyHotNewClass`, `ru.hse.lecture10`, `ru.hse`, `ru`

Но мы не создавали никаких логов с такими именами

В каждой программе создается корневой логгер, имеющий название `""` (пустая строка)

Собственно, он подходит как родитель вообще для всех (но в последнюю очередь по цепочке!)

По-умолчанию, ему автоматически привязан `ConsoleHandler`, и именно этот хэндлер нам всё и логировал

Дженерики

Пусть, мы создаём структуру данных «Двоичное дерево поиска» и для этого храним данные в узлах:

```
class TreeNode {  
    String value;  
    TreeNode left;  
    TreeNode right;  
}
```

Или, ищем минимум в массиве:

```
public static BigInteger minElement(BigInteger[] values) {  
    if (values.length == 0) {  
        return null;  
    }  
  
    BigInteger min = values[0];  
    for (int i = 1; i < values.length; ++i) {  
        if (min.compareTo(values[i]) > 0) {  
            min = values[i];  
        }  
    }  
  
    return min;  
}
```

Как избежать копипасты кода, если нужно реализовать тот же самый алгоритм для другого типа данных?



Принимаем всех

Как избежать копипасты кода, если нужно реализовать тот же самый алгоритм для другого типа данных?

На первый взгляд, можно попробовать заменить тип на `Object` и таким образом принимать любые объекты

Но использование станет неудобным:

```
TreeNode rootNode = new TreeNode();
rootNode.value = "someValue";
// ...
String value = (String) rootNode.value;

BigInteger[] bigIntArray = { /*...*/ };
BigInteger min = (BigInteger) minElement(bigIntArray);
```

Вот бы можно было бы задать тип как параметр для метода или класса, чтобы просто выбирать используемый тип данных...

Дженерики

Именно это и делают *дженерики*, которые также носят название *параметризованные типы*

```
public static <T extends Comparable<T>> T minElement(T[] values) {  
    if (values.length == 0) {  
        return null;  
    }  
  
    T min = values[0];  
    for (int i = 1; i < values.length; ++i) {  
        if (min.compareTo(values[i]) > 0) {  
            min = values[i];  
        }  
    }  
  
    return min;  
}
```

Вместо конкретного типа можно объявить и использовать некоторую типовую переменную, вместо которой может быть подставлен любой тип

Как с ними вообще работать – узнаем позже



Ограничения дженериков

На дженерики распространяется ряд ограничений:

```
TreeNode<String> stringNode; // good
TreeNode<Integer> integerNode; // good
TreeNode<int[]> intArrayNode; // good
TreeNode<int> intNode; // NOT GOOD!
TreeNode<10> tenNode; // NOT GOOD!
```

Значение параметра может быть только ссылочным типом,
нельзя использовать примитивный тип

И передать объект или численное значение тоже не выйдет



Optional

Рассмотрим простой пример класса, использующего дженерик – `Optional`

`Optional` умеет *опционально* хранить объект некоторого типа,
то есть, может хранить и *пустое* значение

Но зачем, если у ссылочных переменных и так есть значение `null`?

Посмотрев на переменную ссылочного типа, нельзя сказать, предусматривает она значение `null` или нет

Всегда проверять переменную на `null` – лень, что рано или поздно приводит нас к `NullPointerException`

Отдельный тип `Optional` позволяет легко отличать обычную ссылку от потенциально отсутствующей:

```
String text = "some text";  
Optional<String> optionalText = Optional.of("more text");
```

Optional

Также, `Optional` позволяет писать код без `if`'ов:

```
Optional<String> optionalText = Optional.of("more text");
optionalText.ifPresent(System.out::println);
```

`println` будет вызван только если значение присутствует

Так бы это выглядело с ручной проверкой

```
String s = /* ??? */;
if (s != null) {
    System.out.println(s);
}
```

И предоставление значения по-умолчанию, на случай его отсутствия:

Аналог без `Optional`:

```
Optional<String> optionalText = Optional.empty();
String value = optionalText.orElse("default text");
```

```
String s = /* ??? */;
String s = s != null ? s : "default text";
```



Объявление дженериков

Посмотрим, как объявлять свои дженерики на примере `Optional`

```
package java.util;

public final class Optional<T> {
    private final T value;

    private Optional(T value) {
        this.value = Objects.requireNonNull(value);
    }

    public static <T> Optional<T> of(T value) {
        return new Optional<>(value);
    }

    public T get() {
        if (this.value == null) {
            throw new NoSuchElementException("No value present");
        }
        return this.value;
    }
    // ...
}
```



Объявление дженериков

```
public final class Optional<T> {
```

Для параметризации класса параметры нужно перечислить после его имени в угловых скобках через запятую

Если требуется ограничить параметр, то можно потребовать наследования другого класса или реализации интерфейса с помощью `extends`

```
public final class Optional<T extends BaseClass & Interface1 & Interface2> {
```

Заметьте, не `implements`

Объявление дженериков

```
package java.util;

public final class Optional<T> {
    private final T value;

    private Optional(T value) {
        this.value = Objects.requireNonNull(value);
    }

    public static <T> Optional<T> of(T value) {
        return new Optional<>(value);
    }

    public T get() {
        if (this.value == null) {
            throw new NoSuchElementException("No value present");
        }
        return this.value;
    }
    // ...
}
```

Тип `T` можно использовать для объявления поля, возвращаемого значения метода, параметра метода или локальной переменной в теле метода



Объявление дженериков

Дженерик-параметр класса используется для параметризации экземпляров, поэтому он не доступен статическим полям и методам

Метод можно параметризовать отдельно от класса, объявив дженерик-параметры перед возвращаемым типом

```
public static <T> Optional<T> of(T value) {  
    return new Optional<>(value);  
}
```

В этом методе параметр `T` уже свой собственный, не связанный с тем `T`, что есть у класса



Использование дженериков

Наш конкретный класс `Optional` не имеет публичных конструкторов, все экземпляры создаются из статических фабричных методов:

```
Optional<String> empty = Optional.empty();  
Optional<String> some = Optional.of("text"); // null will throw  
Optional<String> maybeNull = Optional.ofNullable("text?"); // null is empty Optional
```

Для методов компилятор сам по типу аргумента определит тип их дженерик-параметров

Но можно указать и явно, например, если преобразуем к интерфейсу:

```
Optional<CharSequence> optionalCharSeq = Optional.<CharSequence>ofNullable("text");
```

Будь конструктор публичный, вызвать его можно было бы так:

```
Optional<String> newOptional = new Optional<>("text");
```

Пустые угловые скобки называются `diamond operator`, тип дженерика будет определен по типу в начале строки

Но можно указать тип во вторых скобках и явно



Тонкости дженериков

Дженерики тупее, чем кажется

При компиляции в байт-код вместо них тупо подставляется `Object`

Вся магия происходит в том месте, где класс используется:

```
Optional<String> optional = Optional.of("test");  
String value1 = optional.orElse("default");  
String value2 = optional.get();
```

Компилятор, зная, что в типе указан `String`, запретит нам передавать в методы не `String`, и автоматически преобразует возвращаемые `Object`'ы в `String`

Но это обман и все методы на самом деле всё ещё принимают и возвращают `Object` вместо `T`

Увидеть реальную сущность дженериков можно просто убрав угловые скобки:

```
Optional optional = Optional.of("test");  
String value1 = (String) optional.orElse("default");  
String value2 = (String) optional.get();
```

Вот теперь видим правду, это тот же `Object` в удобной обёртке



Ещё ограничения дженериков

Дженерики тупее, чем кажется

При компиляции в байт-код вместо них тупо подставляется `Object`

А это означает, что в телах методов параметрический тип не особо то и поиспользуешь, ведь он есть `Object`

А именно, с параметрическим типом запрещено делать следующее:

```
T obj = new T();

T[] arr = new T[5];

if (obj instanceof T) {
    // ...
}
```

- создавать экземпляр класса-параметра
- создавать массив с классом-параметром
- использовать проверку `instanceof` с ним