



Lesson 11

13.04.2021

```
public class Ex1 {  
    public static void main(String[] args) {  
        StringBuilder sb = new StringBuilder("abc");  
        String s = "abc";  
        sb.reverse().append("d");  
        s.toUpperCase().concat("d");  
        System.out.println("." + sb + ". ." + s + ".");  
    }  
}
```



```
public class Ex2 {  
    public static void main(String[] args) {  
        List<String> list = new ArrayList<>();  
        list.add("apple");  
        list.add("carrot");  
        list.add("banana");  
        list.add(1, "plum");  
        System.out.println(list);  
    }  
}
```





```
public class Ex3 {  
    public static void main(String[] args) {  
        String s = "JAVA";  
        s = s + "rock";  
        s = s.substring(4, 8);  
        s.toUpperCase();  
        System.out.println(s);  
    }  
}
```

```
public class Ex4 {  
    public static void main(String[] args) {  
        String[] name = {"Sasha", "Ivan", "Masha"};  
        List<String> names = name.asList();  
        names.set(0, "Kate");  
        System.out.println(name[0]);  
    }  
}
```

```
public class Ex5 {  
    public static void main(String[] args) {  
        StringBuilder sb = new StringBuilder("0123456789");  
        sb.delete(2, 8);  
        sb.append("-").insert(2, "+");  
        System.out.println(sb);  
    }  
}
```

Java 14 - record

Принципиально новая языковая фича, доступна в Java 14. Основная цель — избавиться от большого количества вермишели в коде. *Records* должны заменить классы, которые используются только для хранения данных в полях без какого-либо описанного поведения. Определим *Record*, а компилятор сам сгенерирует конструктор, геттеры, методы `equals()` и `hashCode()`, `toString()`. Запись выглядит следующим образом:

```
public record Person(String name, int age) { }
```

```
public final class Person extends java.lang.Record {  
    private final java.lang.String name;  
    private final int age;  
  
    public Person(java.lang.String name, int age) { /* compiled code */ }  
  
    public java.lang.String toString() { /* compiled code */ }  
  
    public final int hashCode() { /* compiled code */ }  
  
    public final boolean equals(java.lang.Object o) { /* compiled code */ }  
  
    public java.lang.String name() { /* compiled code */ }  
  
    public int age() { /* compiled code */ }  
}
```

Java 14 – text block

Текстовые блоки доступны в Java 14. Напомню, что их основная задача — упростить работу с многострочными литералами. Очень удобная фишка для написания *SQL-запросов*, *HTML-кода*, *JSON*. Думаю, что это одна из функций, которая станет очень полезной. Напомним синтаксис. Например, нам необходимо написать *SQL-запрос*. До Java 13 - 14 , для написания запроса удобочитаемым мы бы использовали конкатенацию строк, и запрос выглядел бы примерно так

```
String sql = "SELECT name, age " +  
            "FROM PERSON" +  
            "WHERE name = \'Igor\'" +  
            "AND car=\'Niva\'";
```

```
String sql = ""  
            SELECT name, age  
            FROM PERSON  
            WHERE name = 'Igor'  
            AND car='Niva'  
            "";
```

В Java 14 добавлены два новых разделителя, которые можно использовать в текстовых блоках. Первый, это одиночный пробел: `\s`. Второй, это символ новой строки: `\`.



Java 14 - Switch Expressions

Начиная с Java 14, **switch expressions** переходит из превью-фичи в полноценную функцию. Напомним коротко особенности нового оператора:

- Лямбда-синтаксис
- Возможность использовать более одного выражения в *case*
- Вывод ошибки в случае неполного покрытия множества возможных значений *switch()*.



Java 14 - Pattern Matching

Pattern Matching новая фишка доступная в Java 14. О ней много говорили, её многие ждали и вот она здесь. Цель — объединить в операторе *instanceof* проверку типа объекта и его преобразование.

Java 14 - NullPointerExceptions

Более дружелюбные *NullPointerExceptions*. Нет, компилятор все еще не умеет исправлять *NPE* за нас, но теперь описание исключения станет более информативным

```
var street = message.getAuthor().getAddress().getStreet();
```

```
Exception in thread "main" java.lang.NullPointerException  
    at Main.main(Main.java:11)
```

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "Address.getStreet()" because the return value of "Author.getAddress()" is null  
    at Main.main(Main.java:11)
```



String

У String есть две фундаментальные особенности:


- это immutable (неизменный) класс
- это final класс

В общем, у класса String не может быть наследников (***final***) и экземпляры класса нельзя изменить после создания (***immutable***).

StringBuffer и StringBuilder

Отличие между String, StringBuilder, StringBuffer:

- Классы `StringBuffer` и `StringBuilder` в Java используются, когда возникает необходимость сделать много изменений в строке символов.
- В отличие от `String`, объекты типа `StringBuffer` и `StringBuilder` могут быть изменены снова и снова.
- Основное различие между `StringBuffer` и `StringBuilder` в Java является то, что методы `StringBuilder` не являются безопасными для потоков (несинхронизированные).
- Рекомендуется использовать `StringBuilder` всякий раз, когда это возможно, потому что он быстрее, чем `StringBuffer` в Java.
- Однако, если необходима безопасность потоков, наилучшим вариантом являются объекты `StringBuffer`.



Метод **charAt()** возвращает символ в указанной позиции. А **setCharAt()** изменяет символ в указанной позиции

Метод **append()** присоединяет подстроку к строке

Метод **insert()** вставляет подстроку в указанную позицию

Метод **reverse()** используется для инвертирования строки

Метод **delete()** удаляет подстроку, используя указанные позиции

Метод **deleteCharAt()** удаляет символ с указанной позиции

Метод **replace()** заменяет подстроку в указанной позиции другой



Сборка Java проектов

- Apache Ant <http://ant.apache.org/>
- Apache Maven <https://maven.apache.org/>
- Gradle <https://gradle.org/>




Что такое система сборки?

Система сборки это программа, которая собирает другие программы. На вход система сборки получает исходный код, а на выход выдаёт программу, которую уже можно запустить.

Чем она отличается от компилятора? Если коротко, то система сборки вызывает компилятор при своей работе, а компилятор о существовании системы сборки даже не подозревает.

Если более длинно, то сборка, помимо компиляции, включает в себя ещё целый спектр задач, для решения которых компилятор не пригоден от слова совсем.

Например, если программе для работы нужны какие-нибудь картинки, то положить их в директорию с программой — задача системы сборки. Если программе нужны сторонние библиотеки, то положить их в директорию с программой — задача системы сборки. Ну и так далее.



Автоматизация процесса сборки программного продукта связана с разработкой различных скриптов для выполнения таких действий, как :

компиляция исходного кода в бинарный;

сборка бинарного кода;

подключение внешних библиотек;

выполнение тестов

разворачивание программы на сервере (удаленном компьютере);

оформление сопроводительной документации или описание изменений.

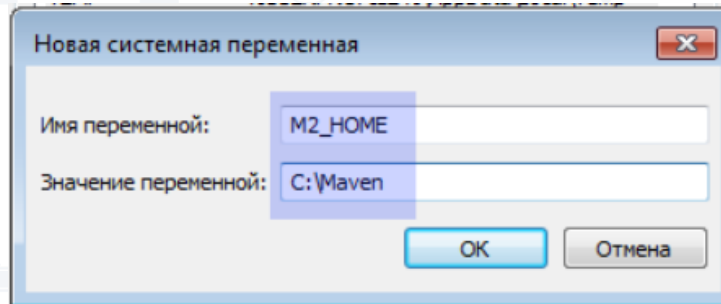
Установка Apache Maven

Установка в Linux

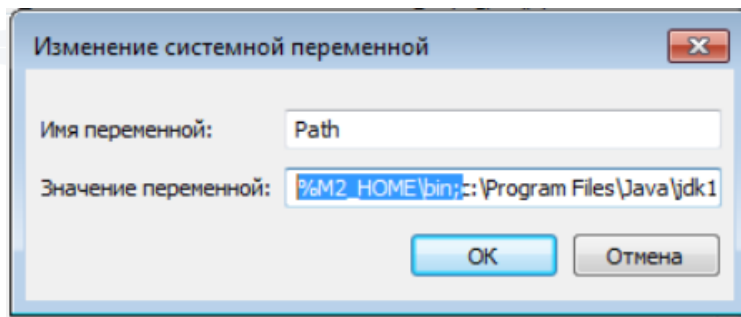
`apt-get install maven2`

Установка в Windows

1. Скачайте нужную вам версию maven с <http://maven.apache.org/download.cgi>
2. Загруженный архив распакуйте куда-нибудь, например в C:\Maven\
3. Перед тем как начать использовать maven, надо настроить переменные окружения. «M2_HOME» и придайте ей значение «C:\Maven» (или куда вы его распаковали):



Ещё желательно добавить путь к maven в переменную PATH. Для этого в том же диалоге найти системную переменную PATH, и добавить к ней «%M2_HOME%/bin»:



Для проверки установилась ли версия maven:

```
mvn -version
```

Зависимости в Maven

Apache Maven — отличная штука, для управления сторонними зависимостями в вашем проекте. Достаточно сказать ему, какой артефакт вам нужен и всё остальное Maven сделает сам.

Все зависимости перечисляются в секции `<dependencies/>`, одна за одной, в любом порядке.

```
1 <dependencies>
2   <dependency>
3     <groupId>junit</groupId>
4     <artifactId>junit</artifactId>
5     <version>4.12</version>
6     <scope>test</scope>
7   </dependency>
8 </dependencies>
```




Области видимости

compile — область видимости по умолчанию. Зависимости с этим score будут доступны и во время сборки и во время тестирования и их даже добавляют в конечный пакет, чтобы они были доступны и во время исполнения.

provided — Почти как *compile*, но в пакет зависимость добавлена не будет. Предполагается, что данные библиотеки будут предоставлены средой выполнения, например J2EE контейнером. Каноничный пример такой зависимости — J2EE API, конкретная реализация которых предоставляется контейнером J2EE.

runtime — антипод *provided*. Означает зависимость, которая требуется для исполнения/тестирования кода, но не для его сборки. Зависимости из этого score так же будут добавлены в пакет.



test — зависимости, которые нужны только и исключительно для тестов. Как [JUnit](#) из примера выше.

system — зависимость которая присутствует в среде Java всегда, тем или иным путём. Maven не будет пытаться предоставить этот артефакт или класть его в пакет итд.

import — используются для импорта зависимостей из других артефактов и управлением зависимостями в сложных пакетах, состоящих из нескольких артефактов.

Сборочный цикл в Maven (23 по умолчанию / 7 явных)

validate — в этой фазе проверяется корректность проекта и обеспечивается доступность необходимых зависимостей

compile — Компилируется исходный код проекта

test — Код [тестов](#) компилируется и запускаются unit тесты

package — Скомпилированный код собирается в пакет (jar/war/ear/etc)

verify — Запускаются интеграционные тесты


install — Собранный ранее пакет устанавливается в локальный репозиторий и становится доступен для сборки других локальных проектов

deploy — Пакет публикуется в удалённых репозиториях, устанавливается на серверы приложений и так далее

Основные плагины maven

Плагин ***compiler*** компилирует исходный java код приложения и тестов в байткод. Особенностью плагина являются его настройки по умолчанию: он ожидает, что исходный код должен быть совместим с java версии 1.5 и генерирует байткод для той же самой версии 1.5.

Плагин ***surefire*** реализует всю магию по исполнению [юнит тестов](#). По умолчанию он проверяет все классы, начинающиеся словом *Test** или наоборот, заканчивающиеся на **Test*, **Tests*, **TestCase* и пытается найти в них тесты и запустить их. Плагин исполняет все найденные тесты и пишет отчёты по каждому запущенному тесту, помещая их в каталог *target/surefire-reports*. В случае если имеется хотя бы один провалившийся тест, плагин прерывает сборку с ошибкой.



Плагин ***failsafe*** — брат близнец плагина ***surefire*** и так же занимается исполнением тестов. Но ***failsafe***, в отличие от ***surefire***, исполняет интеграционные тесты, а не модульные. Поэтому он ищет файлы, оканчивающиеся на ****IT***, ****ITCase*** или начинающиеся на ***IT****. Отчёты о выполнении тестов пишутся в каталог ***target/failsafe-reports***.

Плагин ***jar*** отвечает за упаковку вашего приложения в ***jar*** пакет.



Project lombok

Lombok это прекраснейший препроцессор аннотаций, который сильно упрощает код и делает java похожей на современный язык.

Моя самая любимая функциональность проекта lombok — генерация геттеров и сеттеров. Наконец-то java разработчики избавляются от тяжёлого наследия и сбрасывают с себя цепи.

Генерируют геттеры и сеттеры аннотации *@Getter* и *@Setter*

Конструкторы

Аннотация **@NoArgsConstructor** создаёт конструктор по умолчанию. В случае, если в классе есть **final** поля, такой конструктор сгенерирован не будет. Но, если очень сильно попросить, передав в **@NoArgsConstructor** параметр **force = true**, то конструктор будет сгенерирован, а **final** поля будут инициализированы пустыми значениями.

@RequiredArgsConstructor генерирует конструктор, принимающий значения для каждого **final** поля или поля с аннотацией **@NonNull**. Аргументы конструктора будут сгенерированы в том порядке, в котором поля перечислены в классе.

@AllArgsConstructor генерирует конструктор для всех полей класса.



equals() и hashCode()

Эти методы глубоко пересечены, поэтому генерируются вдвоём, аннотацией **@EqualsAndHashCode**

toString()

генерирует аннотация **@ToString**

@Data

Она добавляет **@Getter/@Setter** ко всем полям, добавляет **@EqualsAndHashCode** и **@ToString**



Исключения в Java

В java исключением называется любая ошибка, которая возникает в ходе выполнения программы. Это может быть несоответствие типов данных, деление на ноль, обрыв связи с сервером и многое другое. Операции по их поиску и предотвращению называются обработкой исключений.

У всех исключений есть общий класс-предок ***Throwable***.

От него происходят две большие группы — исключения (***Exception***) и ошибки (***Error***).

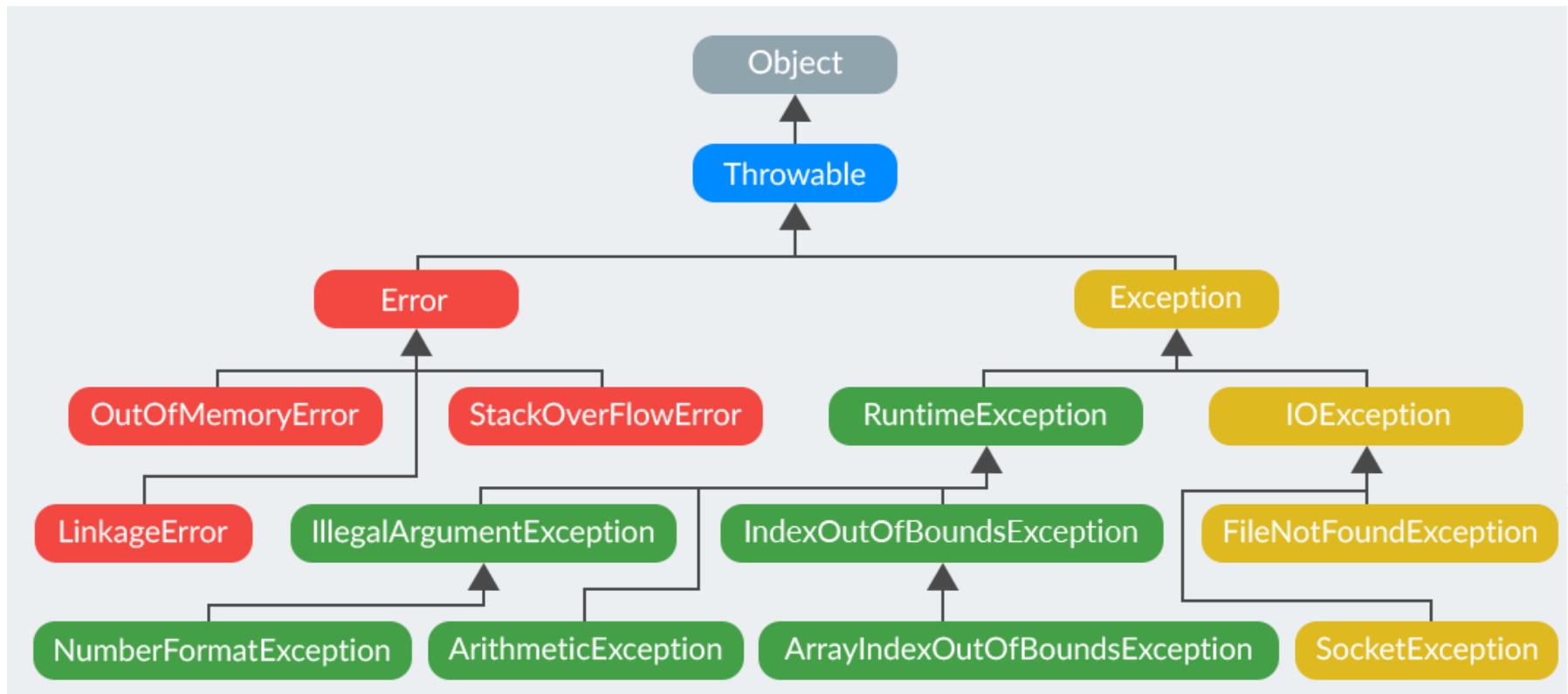
Error — это критическая ошибка во время исполнения программы, связанная с работой виртуальной машины Java. В большинстве случаев Error не нужно обрабатывать, поскольку она свидетельствует о каких-то серьезных недоработках в коде.



Exceptions — это, собственно, исключения: исключительная, незапланированная ситуация, которая произошла при работе программы.

Это не такие серьезные ошибки, как Error, но они требуют нашего внимания.

Все исключения делятся на 2 вида — проверяемые (checked) и непроверяемые (unchecked).





Ключевые слова:

try – определяет блок кода, в котором может произойти исключение;

catch – определяет блок кода, в котором происходит обработка исключения;

finally – определяет блок кода, который является необязательным, но при его наличии выполняется в любом случае независимо от результатов выполнения блока try.

throw – используется для возбуждения исключения;

throws – используется в сигнатуре методов для предупреждения, о том что метод может выбросить исключение.



Обработка исключения

