



План занятия

1. Приведение типов
2. Классы обёртки
3. Парами строки и не только
4. Передача в метод по ссылке и по значению



Приведение типов

Приведение типов

Самая распространённая ошибка при работе с типами данных — ошибка несовместимости типов **`incompatible types`**. Она возникает при попытке сохранить в переменную одного типа значение другого, например:

```
int a = 32.888; // не получится сохранить дробное число в переменную целого типа
```



Приведение типов

Если ошибка произошла с числовыми примитивами — как в этом примере, то её можно решить с помощью **приведения одного типа к другому**. Приведение типов может быть автоматическим, когда программисту ничего не нужно делать, и явным, когда разработчик самостоятельно преобразует один тип в другой.



Автоматическое приведение типов

При написании кода можно столкнуться с ситуацией, когда значение переменной одного типа требуется для метода, который принимает значения другого типа. К примеру, запустите код, где метод `checkMethod()` принимает тип `int`, но переменная `smallNumber`, переданная как аргумент, хранит значение типа `byte`


```
public class Practice {  
    public static void main(String[] args) {  
        byte smallNumber = 40; // тип переменной - byte  
        checkMethod(smallNumber); // передали smallNumber в метод в качестве аргумента  
    }  
  
    // тип параметра метода - int  
    public static void checkMethod(int importantBigNumber) {  
        System.out.println("Метод работает!");  
        System.out.println("smallNumber = " + importantBigNumber);  
    }  
}
```



Результат

```
Метод работает!  
smallNumber = 40
```

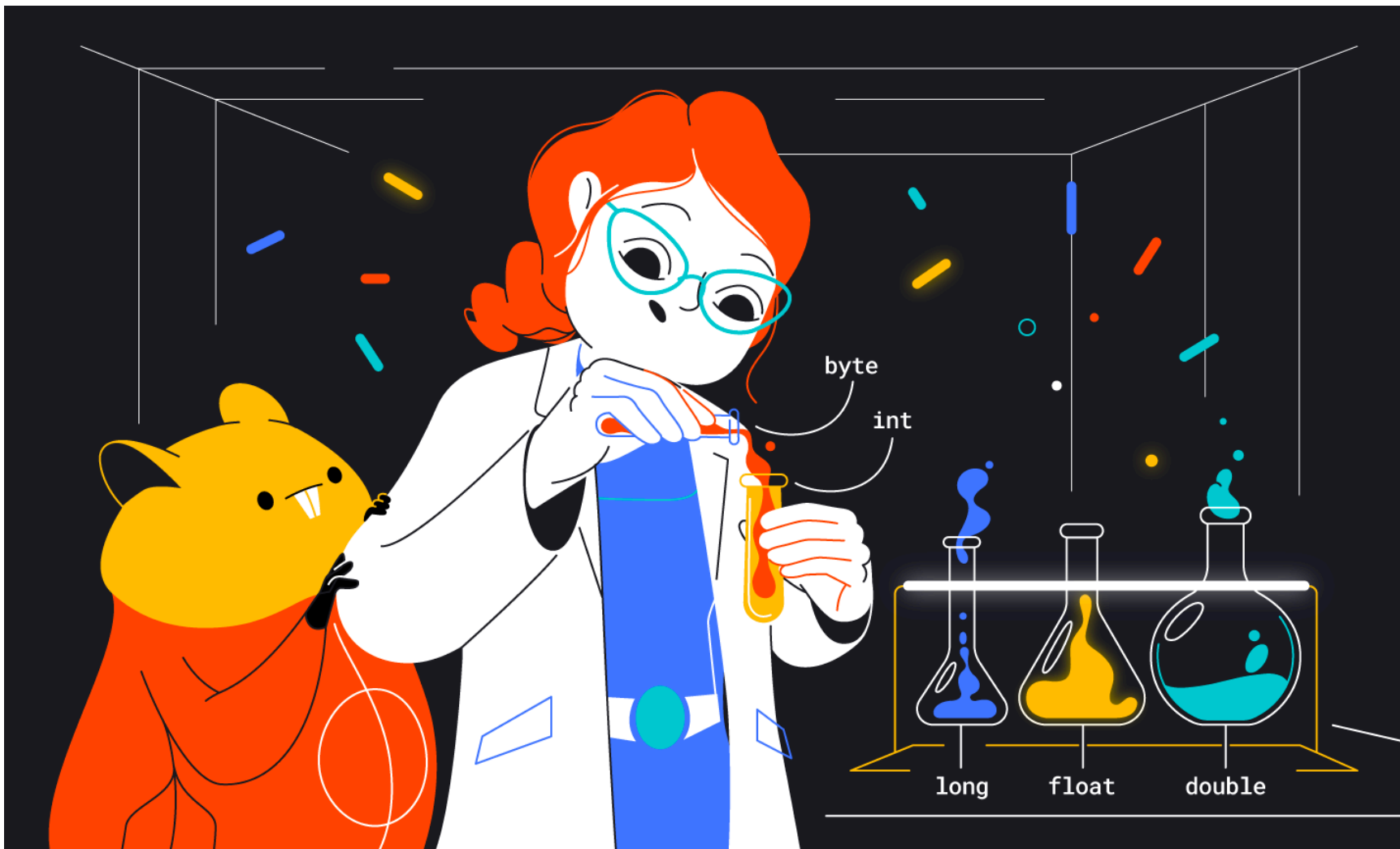
Ошибки нет. Код сработал, так как Java понимает, что работает с «родственными» целочисленными типами и диапазон **byte** входит в диапазон типа **int**. Поэтому компилятор легко конвертирует **byte** в **int**. Это называется **автоматическим (или неявным) приведением типов**.




Автоматическое приведение типов работает, когда типы данных с меньшим диапазоном нужно привести к типам с большим диапазоном. Это всё равно что перелить жидкость из маленькой ёмкости, например, стакана, в ёмкость побольше — кувшин. Вода точно поместится и не прольётся. Один тип автоматически приводится к другому не только при передаче в метод, но и при обычной инициализации переменных:

```
int smallNumber = 8;  
long bigNumber = smallNumber; // Ошибки не будет
```


Подобное преобразование называется **расширяющим приведением типа**. Оно возможно для любых числовых типов. Например, `short` можно расширить до `long`, `float` до `double` и так далее.





Кроме того, любой из целочисленных типов при необходимости будет автоматически расширен до дробного. Дело в том, что диапазон меньшего из дробных примитивов **float** включает диапазон большего из целочисленных типов **long**. Сравните: в **long** можно хранить числа примерно от $-9 * 10^{18}$ примерно до $9 * 10^{18}$, а в переменных типа **float** от $-3.4 * 10^{38}$ до $3.4 * 10^{38}$. Поэтому любой целочисленный тип автоматически будет приведён к любому дробному:

```
int integerNum = 999;  
double a = integerNum; // корректное приведение типов, ошибки не будет  
long longNum = 9_223_372_036_854_775_807L; // Максимальное значение для типа long  
float floatNum = longNum; // и здесь ошибки не будет
```

Явное приведение типов

Разберём обратную ситуацию. Передадим в метод, принимающий **byte**, переменную типа **int**:

```
public class Practice {  
    public static void main(String[] args) {  
        int integerNumber = 40; // тип переменной - int  
        checkMethod(integerNumber);  
    }  
    // тип параметра метода - byte  
    public static void checkMethod(byte importantSmallNumber) {  
        System.out.println("Метод работает!");  
        System.out.println(importantSmallNumber);  
    }  
}
```

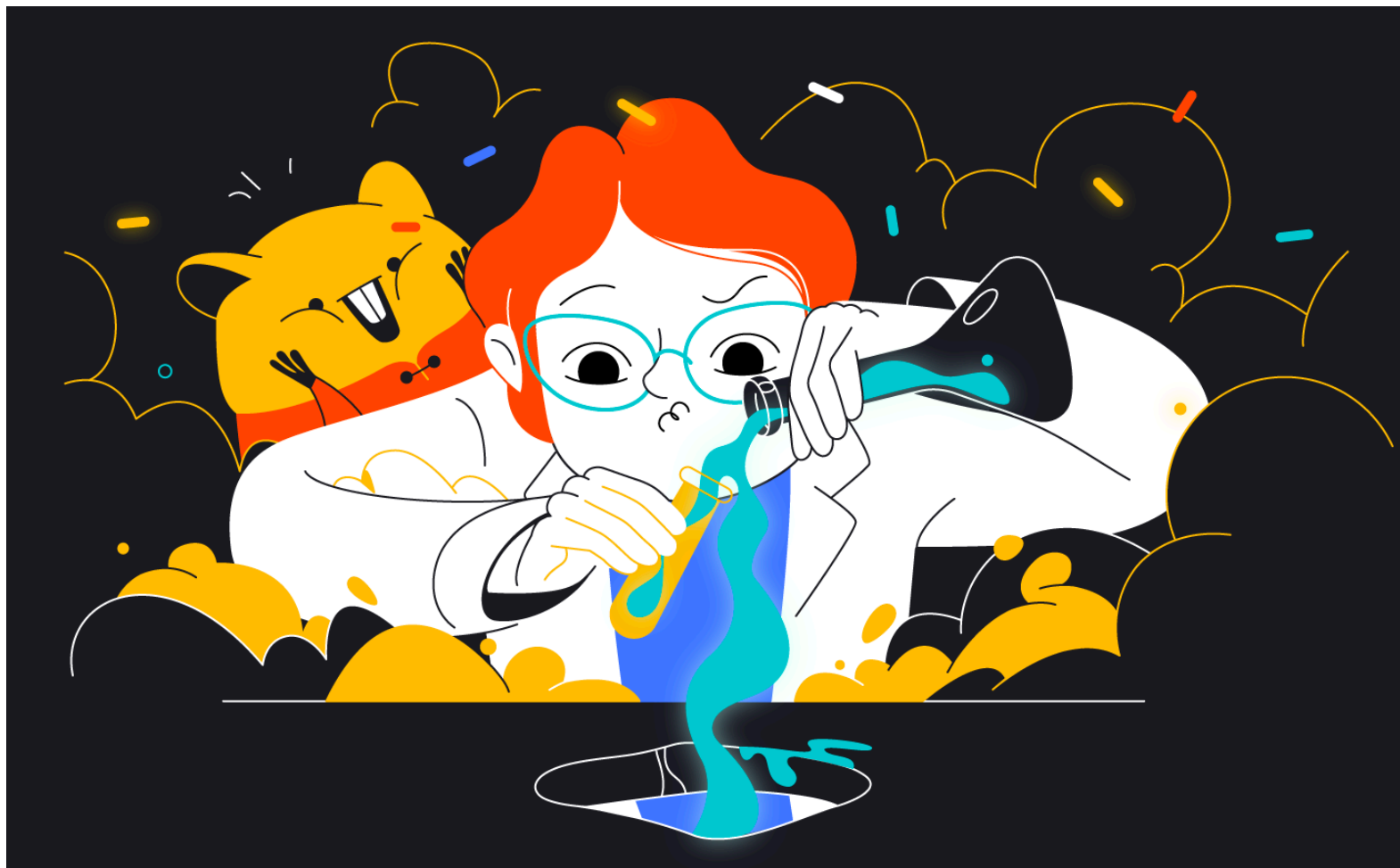


Результат

```
java: incompatible types: possible lossy conversion from int to byte
```

Несмотря на то, что значение **integerNumber** не выходит за пределы диапазона типа **byte**, произошла ошибка. Тип с бóльшим диапазоном не получается конвертировать в тип с меньшим.

Java не позволяет автоматически провести **сужающее приведение типов**. Текст в консоли предупреждает, что при попытке это сделать возникает риск потери данных — **possible lossy conversion from int to byte**. В литровый кувшин нельзя налить два литра воды — часть жидкости не войдёт и прольётся.



Однако значение переменной `integerNumber` входит в диапазоны обоих типов `byte` и `int` — и в отличие от Java мы знаем, что риска потери данных нет. В этом случае можно привести типы самостоятельно. В круглых скобках перед значением не того типа нужно указать нужный. В нашем примере это (`byte`):

```
public class Practice {
    public static void main(String[] args) {
        int integerNumber = 40;
        checkMethod((byte) integerNumber); // привели тип int к типу byte
    }

    public static void checkMethod(byte importantSmallNumber) {
        System.out.println("Метод работает!");
        System.out.println("integerNumber = " + importantSmallNumber);
    }
}
```



Результат

```
Метод работает!  
integerNumber = 40
```

Это называется **явным приведением типов** — его осуществляет сам разработчик, а не Java. Так можно привести любой числовой тип с большим диапазоном к типу с меньшим, но будьте осторожны — помните о рисках.

К примеру, присвоим переменной `integerNumber` значение за пределами диапазона `byte` (500) и явно приведём его к типу `int`:

```
public class Practice {
    public static void main(String[] args) {
        int integerNumber = 500;
        byte smallNumber = (byte) integerNumber; // привели integerNumber к типу byte

        checkMethod(smallNumber);
    }
    public static void checkMethod(byte importantSmallNumber) {
        System.out.println("Метод работает!");
        // ожидаем, что значение importantSmallNumber будет равно 500
        System.out.println("smallNumber = " + importantSmallNumber);
    }
}
```


Результат

Метод работает!
`smallNumber = -12`

Программа напечатает `smallNumber = -12` — совсем не тот результат, который требовался. Java провела приведение типа согласно своим внутренним алгоритмам и часть информации потерялась.

Зафиксируем правило: **производить явное сужающее приведение типов нужно крайне аккуратно!**

Дробные типы также можно явно привести к целочисленным. В этом случае останется только целая часть, дробная будет просто откинута:

```
public class Practice {  
    public static void main(String[] args) {  
        double doubleNumber = 41.935;  
        byte smallNumber = (byte) doubleNumber; // привели double к типу byte  
  
        checkMethod(smallNumber);  
    }  
  
    public static void checkMethod(byte importantSmallNumber) {  
        System.out.println("Метод работает!");  
        System.out.println("smallNumber = " + importantSmallNumber);  
    }  
}
```



Результат

```
Метод работает!  
smallNumber = 41
```



Задача

Часто при обновлении программ необходимо поддерживать старый код. Доработайте код новой версии игры-стратегии, чтобы он поддерживал параметры из старой версии:

https://github.com/practicetasks/java_tasks/tree/main/types/task_2



Решение

<https://gist.github.com/practicetasks/4f1e08b2de3907b1f4c63bc778327d1f>



Классы-обёртки



Классы-обёртки

Примитивы неспроста так называются. Они умеют выполнять только одну задачу — хранить помещённое в них значение. Этого функционала бывает недостаточно. К примеру, многие структуры данных на Java не работают с примитивами. Поэтому у примитивов есть «старшие братья» — классы-обёртки.



Классы-обёртки

Классы-обёртки — специальные классы из стандартной библиотеки Java, призванные расширить функционал и возможности использования примитивных типов.



Классы-обёртки

В отличие от примитивов классы-обёртки:

- Хранят не значение, а ссылку на него.
- Не имеют фиксированного размера в памяти компьютера.
- В качестве значения по умолчанию возвращают **null**.
- Обладают своими методами.

Имена классов-обёрток являются производными от названий примитивов и пишутся в коде с заглавной буквы.

Соответствие примитивов и обёрток

● Примитив

● Обёртка

● Примитив

● Обёртка

byte

Byte

short

Short

int

Integer

long

Long

float

Float

double

Double

char

Character


boolean

Boolean



Классы-обёртки


Диапазон значений класса-обёртки такой же, как и у соответствующего ему примитива. К примеру, в переменной типа **Short** можно хранить только числа от -32 768 до 32 767.



Конвертировать примитивы в обёртки Java умеет автоматически. Такой процесс называется *boxing* (от англ. *box* — «коробка»), или **упаковкой примитива**. Упаковка происходит каждый раз, когда переменной класса-обёртки передаётся значение соответствующего ему примитивного типа. Например, при инициализации переменных:

```
Integer number = 10; // значение типа int конвертируется в обёртку Integer
Boolean flag = true; // значение boolean упаковали в Boolean
Character letter = 'a'; // упаковка char в Character
Float amount = 4.55595993045F; // число типа float конвертируется в объект типа Float
```

Все четыре переменные здесь: **number**, **flag**, **letter** и **amount** — объекты классов-обёрток. А присваиваемые им значения — примитивы. Поэтому в момент инициализации происходит упаковка.



Упаковка также произойдёт, если передать в объект класса-обёртки имя переменной примитивного типа:

```
int primitive = 7;  
Integer wrapper = primitive; // здесь произошла упаковка примитива int в обёртку
```


Этот же процесс происходит и при передаче аргументов в методы:

```
public class Practice {  
    public static void main(String[] args) {  
        byte primitive = 7;  
        method(primitive); // передаём переменную примитивного типа  
    }  
    public static void method(Byte number) { // здесь произойдёт упаковка в обёртку  
        System.out.println("У нас тут обёртка - " + number);  
    }  
}
```



Результат

У нас тут обёртка - 7



Обратный процесс по приведению класса-обёртки к примитиву называется *unboxing*, или распаковкой типов. Он также происходит автоматически:

```
Boolean wrapper = true; // упаковали значение в класс-обёртку Boolean
boolean primitive = wrapper; // распаковали обратно в примитив boolean
```



Распаковка сработает при передаче переменной класса-обёртки в метод:

```
public class Practice {  
    public static void main(String[] args) {  
        Short wrapper = 7; // упаковали примитив в обёртку  
        method(wrapper); // передали в метод  
    }  
  
    public static void method(short number) { // здесь сработала распаковка  
        System.out.println("Это очень примитивно!");  
    }  
}
```



Результат

Это очень примитивно!



Есть только одно исключение — если в переменной класса-обёртки хранится значение `null`. В этом случае при распаковке Java выдаст ошибку:

```
public class Practice {  
    public static void main(String[] args) {  
        Float wrapper = null;  
        float primitive = wrapper;  
        System.out.println(primitive);  
    }  
}
```

Результат

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke  
"java.lang.Float.floatValue()" because "wrapper" is null  
    at Practice.main(Practice.java:4)
```

Причина сбоя в том, что если в переменной **wrapper** значение **null**, то в ней нет объекта, а значит, и распаковывать нечего. Такая же ошибка может возникнуть при работе со значениями по умолчанию: у классов-обёрток и примитивов они не совпадают. Поэтому с распаковкой нужно быть внимательными!




Парсим строки и не только



Парсим строки и не только

В отличие от примитивов, у классов-обёрток есть свои методы. Их достаточно много, но большая часть используется редко. Мы расскажем о тех методах, которые с высокой вероятностью могут пригодиться в работе.




У всех классов-обёрток, кроме **Character**, есть метод, позволяющий преобразовывать строки в свой тип. На профессиональном сленге это называется «парсить» строки (от англ. *parse* — «разбирать»). Это происходит с помощью метода **parse[примитив]()**.

Преобразование строки в число выглядит так:


```
String input = "1000";  
Integer number = Integer.parseInt(input);
```

Важно, чтобы в строку было записано именно число, а не его буквенное выражение или какая-то последовательность.



Метод **parse** вызывается с помощью имени класса-обёртки и точечной нотации. В качестве аргумента в него передаётся имя переменной или значение, которое нужно преобразовать. Так будет выглядеть преобразование строк для каждого из классов-обёрток:

```
Byte.parseByte("12");  
Short.parseShort("345");  
Integer.parseInt("999999");  
Long.parseLong("1000000000000");  
Float.parseFloat("12.3");  
Double.parseDouble("456.789");  
Boolean.parseBoolean("true");
```

Ваш коллега очень торопился и некорректно написал код метода, который должен возвращать сумму двух чисел, которые передаются как строки. Найдите и исправьте ошибку. Вносить изменения можно только в тело метода.

```
public class Practice {  
    public static void main(String[] args) {  
        String firstNumber = "123.45";  
        String secondNumber = "234.56";  
        System.out.println(addNumbers(firstNumber, secondNumber));  
    }  
  
    private static Float addNumbers(String firstNumber, String secondNumber) {  
        return Float.parseFloat(firstNumber + secondNumber);  
    }  
}
```

Результат

```
Exception in thread "main" java.lang.NumberFormatException: multiple points
    at java.base/
jdk.internal.math.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:1914)
    at java.base/jdk.internal.math.FloatingDecimal.parseFloat(FloatingDecimal.java:122)
    at java.base/java.lang.Float.parseFloat(Float.java:556)
    at Practice.addNumbers(Practice.java:9)
    at Practice.main(Practice.java:5)
```

Решение

```
private static Float addNumbers(String firstNumber, String secondNumber) {  
    return Float.parseFloat(firstNumber) + Float.parseFloat(secondNumber);  
}
```



Результат

358.01

С помощью методов классов-обёрток **Integer**, **Long**, **Float** и **Double** можно находить максимальное и минимальное значение из двух вариантов. Методы **max()** и **min()** вызываются также с помощью имени класса-обёртки и точечной нотации. Они возвращают примитив. Например:

```
public class Practice {  
    public static void main(String[] args) {  
        long a = 142_858_234;  
        long b = 9_123_456_678L;  
        long maximum = Long.max(a, b);  
        System.out.println("Максимальное значение - " + maximum);  
  
        double c = 0.00175764;  
        double d = 0.00138534;  
        // возвращенный примитив можно сразу же упаковать в обёртку  
        Double minimum = Double.min(c, d);  
        System.out.println("Минимальное значение - " + minimum);  
    }  
}
```



Парсим строки и не только

В качестве аргументов в методы нужно передавать значения соответствующего класса или его примитива. То есть в этом примере сигнатура методов будет такой — `Long.max(long, long)` и `Double.min(double, double)`.

У классов `Byte` и `Short` методов по поиску максимального и минимального значения нет. При работе с переменными типов `byte` и `short` минимум и максимум можно вычислить при помощи соответствующих методов класса `Integer`. Для этого нужно воспользоваться явным приведением типов.

```
public class Practice {  
    public static void main(String[] args) {  
        byte b1 = 10;  
        byte b2 = 20;  
        printMaxOfBytes(b1, b2);  
  
        short sh1 = -5;  
        short sh2 = 5;  
        printMinOfShorts(sh1, sh2);  
    }  
  
    public static void printMaxOfBytes(byte b1, byte b2) {  
        int max_of_bytes = Integer.max((int)b1, (int)b2);  
        System.out.println(max_of_bytes);  
    }  
  
    public static void printMinOfShorts(short sh1, short sh2) {  
        int min_of_shorts = Integer.min((int)sh1, (int)sh2);  
        System.out.println(min_of_shorts);  
    }  
}
```


Задача

Допишите реализацию метода, который возвращает максимум двух чисел типа **byte**. Необходимо использовать метод `Integer.max(int, int)`.

```
public class Practice {  
    public static void main(String[] args) {  
        byte a = 10;  
        byte b = 20;  
        System.out.println(findMax(a, b));  
    }  
  
    private static byte findMax(byte firstNumber, byte secondNumber) {  
        return ...  
    }  
}
```


Решение

```
private static byte findMax(byte firstNumber, byte secondNumber) {  
    return (byte) Integer.max(firstNumber, secondNumber);  
}
```



Для приведения переменных классов-обёрток к примитивам можно ещё воспользоваться встроенным методом `[имя примитива, к которому нужно привести]value()`. Например, приведём переменную типа `Long` к типу `short`:

```
Long bigNumber = 10L;  
short smallNumber = bigNumber.shortValue();
```

Для вызова метода используется имя переменной-объекта и точечная нотация.

Логика приведения типов при помощи метода `value()` такая, как и при явном приведении типов с помощью круглых скобок.

Задача

Кота Пикселя на вечер субботы оставили бабушке. Она приготовила ему на выбор говядину и курицу, а перед сном налила молока и поставила блюдо сливок. Пиксель внимательно следит за питанием и в обоих случаях выбрал наименее калорийную еду. Вычислите, сколько всего ккал съел Пиксель за день, и проверьте, уложился ли питомец в свой лимит в 100 ккал. Чтобы код сработал, вам также нужно найти и исправить ошибки, касающиеся типов.

https://github.com/practicetasks/java_tasks/tree/main/types/task_3



Решение

<https://gist.github.com/practicetasks/7c2776b97d6e7b7afaaf1e521636d400>




Передача в метод по ссылке и по значению



Передача в метод по ссылке и по значению

В качестве аргумента в метод можно передать как примитив, так и объект класса. В первом случае состоится передача **по значению**, а во втором случае — **по ссылке**. Разберём подробно, как это происходит.

Переменные примитивного типа хранят в себе непосредственно сами значения. Поэтому, когда вы используете такую переменную в качестве аргумента, её содержимое копируется в метод. Это значит, что появляются две переменные с одинаковым значением [мы уже немного рассказывали об этом в теме о методах]. Изменение значения одной не повлияет на значение другой.



```
public class Practice {
    public static void main(String[] args) {
        int number = 10; // объявили переменную примитивного типа
        changeVariable(number); // передали её значение в метод
        System.out.println(number); // значение number не изменилось
    }

    private static void changeVariable(int variable) {
        // переменная variable получила значение 10
        variable = variable * 3; // значение переменной variable стало 30
    }
}
```



Результат

10

Будет напечатано значение переменной **number**: оно не поменялось и равно **10**.

Чтобы **number** стало равно **variable**, нужно вернуть новое значение из метода с помощью оператора **return** и присвоить его **number**.

```
public class Practice {  
    public static void main(String[] args) {  
        int number = 10; // объявили переменную примитивного типа  
        number = changeVariable(number); // присвоили number новое значение  
        System.out.println(number); // теперь number равно 30  
    }  
  
    private static int changeVariable(int variable) {  
        return variable = variable * 3; // возвращаем новое значение variable  
    }  
}
```



Результат

30



Передача в метод по ссылке и по значению

Передача по ссылке работает по-другому. В метод передаётся не значение, а ссылка на него, и переменная (объект, на который указывает ссылка) при этом не дублируется. Метод переходит по ссылке и меняет значение в первоисточнике. Старое значение не сохраняется.

```
public class Practice {
    public static void main(String[] args) {
        Cat pixel = new Cat("Рыжий"); // создали рыжего кота
        changeColor(pixel); // передали объект в метод
        System.out.println(pixel.color + " очень идёт коту."); // кот теперь чёрный
    }

    private static void changeColor(Cat someCat) { // метод принимает объекты класса Cat
        someCat.color = "Чёрный"; // и меняет цвет объекта, кот становится чёрным
    }
}

class Cat {
    String color;

    public Cat(String catColor) {
        color = catColor;
    }
}
```



Результат

Чёрный очень идёт коту



Передача в метод по ссылке и по значению

Когда происходит передача в метод по ссылке, действия выглядят так: «сделай с объектом (котом), находящимся по такому-то адресу (ссылка на объект кота), такое-то действие (поменяй коту цвет)».

Когда мы объясняли разницу между примитивными типами и ссылочными, мы использовали пример с конвертами из интернет-магазина. В одном из них лежал журнал — значение, а в другом буклет с адресом, где можно забрать кофемашину — ссылка. Чтобы понять разницу между тем, как работает передача по ссылке и по значению, обратимся к этому примеру ещё раз.



Передача в метод по ссылке и по значению

Представьте, что идентичные конверты доставили и вам, и вашему соседу. После того как вы почитаете журнал, вырежете из него страницы или уроните в ванну, с журналом соседа ничего не произойдёт. А вот кофемашина одна — кто первый сходит за ней, тому она и достанется.

При передаче в метод класса-обёртки нужно быть внимательными! Из-за автоматических процессов упаковки и распаковки в примитив и обратно можно получить совсем не тот результат, который ожидается.

Например, угадайте, что будет напечатано в результате замены типа `int` на класс-обёртку `Integer`

```
public class Practice {  
    public static void main(String[] args) {  
        Integer number = 10;  
        changeVariable(number);  
        System.out.println(number);  
    }  
  
    private static void changeVariable(Integer variable) {  
        variable = variable * 3;  
    }  
}
```


Результат

10

Снова получили **10**, хотя рассчитывали на **30**! Разберём, почему значение не изменилось. Передача была по ссылке, но посмотрим внимательнее, что произошло внутри метода. После перемножения **variable * 3** получим **int**. При сохранении **int** в переменную типа **Integer** произойдёт упаковка и будет создан новый объект! В переменную **variable** сохранится ссылка на новое утроенное значение, а в переменной **number** останется ссылка на старое.

Задача

При оплате проезда на автобусе пассажир получает билет с уникальным номером. Однако текущая реализация программы по учёту поездок выдаёт одинаковые номера для всех билетов. Исправьте код так, чтобы каждый следующий билет был больше предыдущего на единицу. Сигнатуры методов **increaseTicketNumber** и **increase** должны остаться неизменными, при этом учтите, что тип возвращаемого значения в сигнатуру не входит.

```
class Bus {  
    public Bus(int initialNumber) {  
        ticketNumber = initialNumber;  
    }  
  
    int ticketNumber;  
}
```

```
public class Practice {  
    public static void main(String[] args) {  
        Bus bus = new Bus(23765);  
        String[] passengersTimestamps = new String[]{  
            "08:33", "09:42", "10:43", "17:59", "18:01", "19:15"  
        };  
  
        for (int i = 0; i < passengersTimestamps.length; i++) {  
            increaseTicketNumber(bus);  
            System.out.println("Оплата поездки в " + passengersTimestamps[i]  
                + ". Номер билета: " + bus.ticketNumber);  
        }  
    }  
  
    private static void increaseTicketNumber(Bus bus) {  
        increase(bus.ticketNumber, 1);  
    }  
  
    private static void increase(int numberToIncrease, int increaser) {  
        numberToIncrease = numberToIncrease + increaser;  
    }  
}
```



Ожидаемый результат

Оплата поездки в 08:33. Номер билета: 23766
Оплата поездки в 09:42. Номер билета: 23767
Оплата поездки в 10:43. Номер билета: 23768
Оплата поездки в 17:59. Номер билета: 23769
Оплата поездки в 18:01. Номер билета: 23770
Оплата поездки в 19:15. Номер билета: 23771

Решение

```
private static void increaseTicketNumber(Bus bus) {  
    bus.ticketNumber = increase(bus.ticketNumber, 1);  
}  
  
private static int increase(Integer numberToIncrease, Integer increaser) {  
    return numberToIncrease + increaser;  
}
```