

**The Class Object**

In this example, you will review some basic recommendations for creating the method *equals ()*.

**Step 1. Checking references for equivalence**

First, you must check to see if the references **this** and **obj** are identical. The expression below is used to optimize the check: It is a lot faster to check the equivalence of references than to compare fields of objects.

**if** (**this** == obj) **return** true;

**Step 2. Checking the null reference**

Next, you have to find out if the reference **obj** is **null**. If it is, the value **false** must be returned.

**if** (obj == null) **return** false;

**Step 3. Comparing classes**

It is necessary to compare the classes **this** and **obj**. If the verification semantics can change in the subclass, you need to use the method ***getClass()***.

**if** (**this**.getClass() != obj.getClass()) **return** false;

**Step 4. Using instanceof**

If the verification principle remains valid for all subclasses, you should use the operation **instanceof**.

**if** (!(obj **instanceof** NameClass)) **return** false;

**Step 5. Transformation into a variable**

It is necessary to transform the object **obj** into a variable of the necessary class.

NameClass other = (NameClass) obj;

**Step 6. Comparing fields**

Finally, you need to compare all the fields. For fields of primitive types, use the comparison operator (==) for equality, while for object fields, use the method *equals()*. If all the fields of the two objects match, the value **true** is returned; otherwise, **false** is returned.

**return** field1 == other.field1 && field2.equals(other.field2) && ...;

Important! If the method ***equals()*** is overridden in a subclass, you need to include the call **super.equals(other)** in it.

**The method hashCode()**

The method *hashCode()* should always be overridden when the method *equals(Object)* has been overridden.

When overriding the method *hashCode()*, three rules should always be followed:

* When executing a Java application, if the method used to calculate the hash code is called at the same object several times, it should return the same value, unless the object has changed.
* When two objects are equal (i.e., the result of calling equals(Object) is true), calling the method used to calculate the hash code for each of the two objects should return **exactly the same** result.
* When two objects are not equal (i.e., the result of calling equals(Object) is false), calling the method used to calculate the hash code for each of the two objects should return **different** results.

The method *hashCode()* should return an integer (which can be negative)—a unique identifier that, in most cases, depends only on the values of the objects' properties. To ensure that different objects have unique hash codes, it is enough to combine the hash codes of the instance fields.

Take a look at an example of overriding the method *hashCode()*.

**public** **class** Student {

**private** String name;

**private** **long** phone;

**private** **int** age;

*// …*

@Override

**public** **int** hashCode() {

**int** result = 17;

result = 31 \* result + name.hashCode();

result = 31 \* result + (int) (phone ^ (phone >>> 32));

result = 31 \* result + age;

**return** result;

}

}

**public** **class** Demo9 {

**public** **static** **void** main(String[] arg) {

Student stud1 = **new** Student(**"Peter"**, 5558956L, 20);

Student stud2 = **new** Student(**"Ivan"**, 9876543L, 18);

Student stud3 = **new** Student(**"Dasha"**, 5558956L, 20);

Student stud4 = **new** Student(**"Ivan"**, 9876543L, 18);

System.out.println(stud1.hashCode());

System.out.println(stud2.hashCode());

System.out.println(stud3.hashCode());

System.out.println(stud4.hashCode());

}

}

**Output:**

1160475683

-1786389060

-1015000986

-1786389060

**Regular Expressions**

**Metacharacters to search for a match of the strings or text boundaries**

^ — string beginning.

$ — string end.

\b — word boundary.

\B — not a word boundary.

\A — input start.

\G — end of the previous match.

\Z — input end, except for the end terminator, if applicable.

\z — input end.

**Metacharacters to search for character classes**

\d — numeric character.

\D — non-numeric character.

\s — whitespace character.

\S — non-whitespace character.

\w — alphanumeric character or an underscore.

\W — any character, except for an alphabetic, numeric character or the underscore character.

. — (full stop) any character, except for the new string character.

**Metacharacters to search for text delimiter characters**

\t — tabulation character.

\n — new line character.

\r — carriage return character.

\f — switching to a new page.

\u0085 — next line unicode character.

\u2028 — line separator unicode character.

\u2029 — paragraph separator unicode character.

**Metacharacters to group characters**

[abc] — any of the listed (a,b, or c).

[^abc] — any, except for the listed (neither a, nor b, nor c).

[a-zA-Z] — merging ranges (Roman characters from a to z without considering case).

[a-d[m-p]] — combining characters (from a to d and from m to p).

[a-z&&[def]] — overlapping characters (characters d,e,f).

[a-z&&[^bc]] — subtracting characters (characters a, d-z).

**Quantifiers**

These are metacharacters that are used to indicate the number of characters. They always come after a character or a group of characters.

? — one or absent.

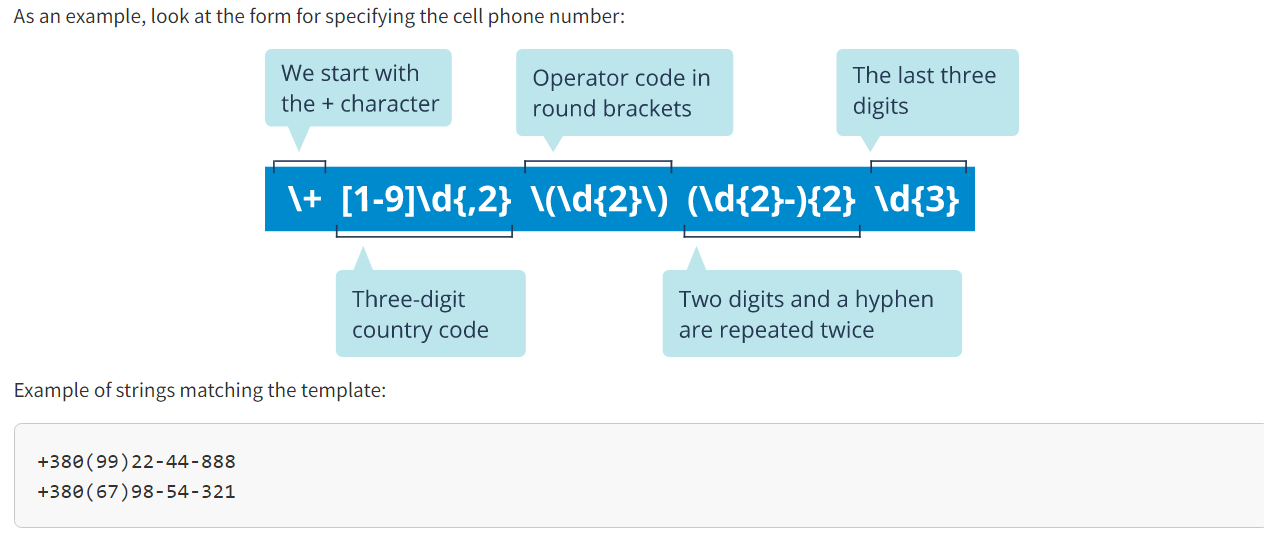
\* — zero or more times.

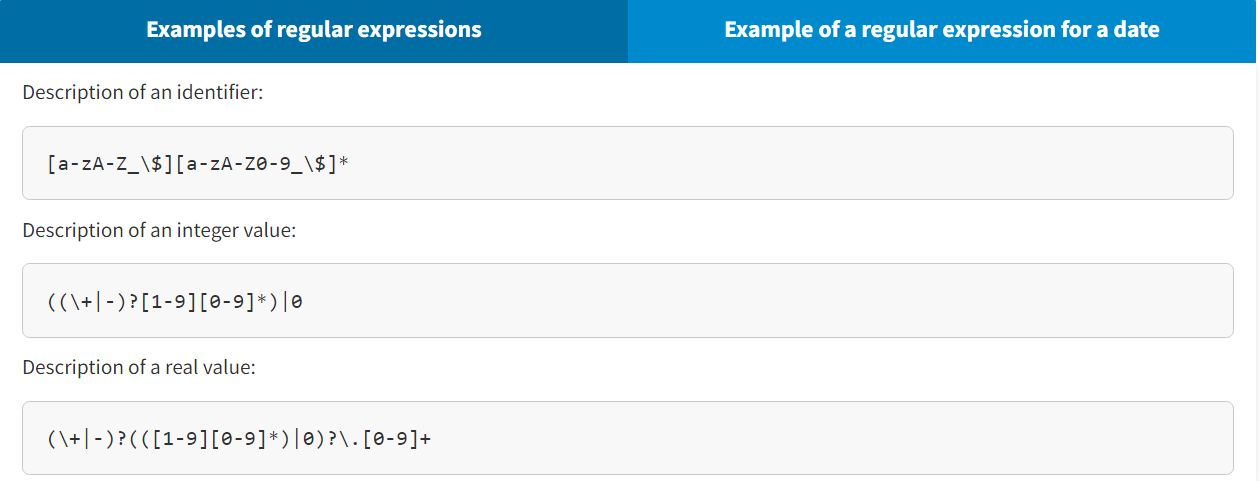
+ — one or more times.

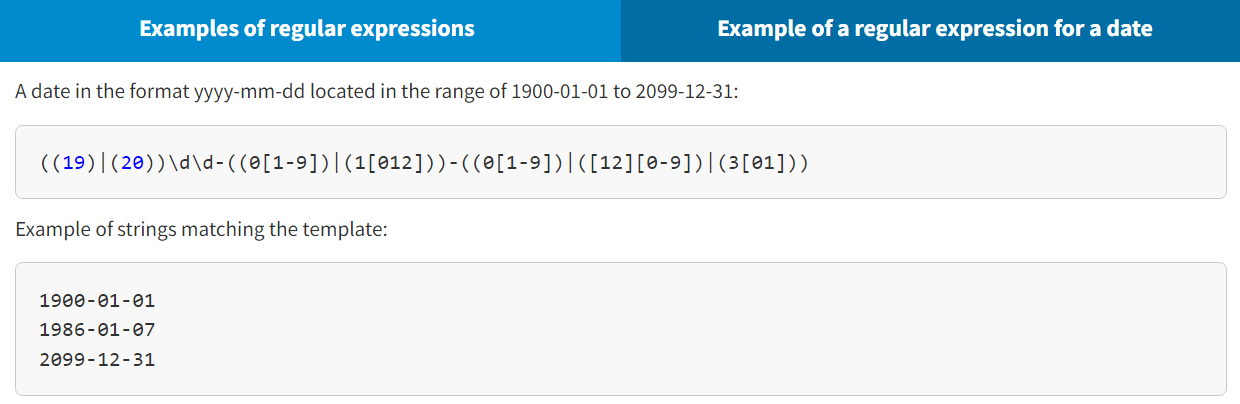
{n} — n times.

{n,} — n times and more.

{n,m} — at least n times but no more than m times.

****

****

****

**Custum Exception**

When writing custom exceptions, keep in mind the following:

* If you want to write a **checked exception**, then you need to expand the **Exception** class.
* If you want to write a **runtime exception (unchecked)**, then you need to expand the **RuntimeException** class.

Custom exceptions are used to handle logical errors of program in the same way as for system exceptions.

**Example of using a custom exception**

The example describes the **Student** class that has setters to set fields. The *mark* field has limitations for the range of values; therefore, the *setMark()* method first checks the input value. If the value is out of range, an exception is thrown. This is not a common exception but a logical one. To handle it in the same way as common exceptions, a custom exception **MarkException** is described as a subclass of **Exception**. The program can resume its work after this error and the range of values is known; therefore, this exception refers to the checked category (checked exception). Since the *setMark()* method does not have information about how to handle this exception, an exception passes to the calling method. The *setMark()* method is called in the *main()* method. According to the rules for handling checked exceptions, a call to the *setMark()* method is controlled for occurring exceptionhandled, which is handled in the *main()* method — in that case, by showing information about the occurred situation to the user.

**public** **class** MarkException **extends** Exception {

@Override

**public** String getMessage() {

**return** **"Unacceptable value!"**;

}

}

**public** **class** Student {

**private** String firstName;

**private** **int** group;

**private** **double** mark;

**public** Student(String firstName, **int** group) {

**this**.firstName = firstName;

**this**.group = group;

}

**public** **void** setMark(**double** mark) **throws** MarkException {

**if** (mark < 0 || mark > 100) {

**throw** **new** MarkException();

}

**this**.mark = mark;

}

}

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Student stud = **new** Student(**"Ivan"**, 505);

**try** {

stud.setMark(101);

} **catch** (MarkException ee) {

System.err.println( ee.getMessage() );

}

}

}

 Copy

**Console Output:**

Unacceptable value!

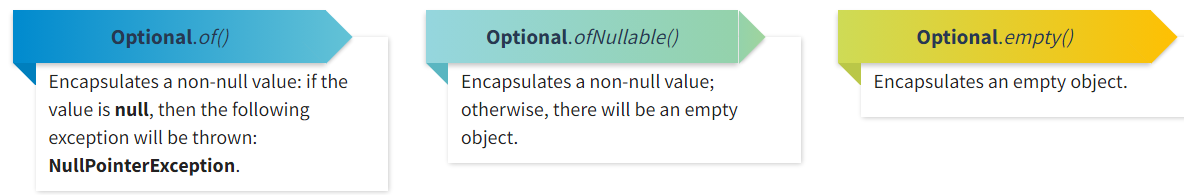
You also received practical recommendations to be used for handling exceptions:

* Specify exceptions
* Use the handling mechanism correctly
* Log events in handlers
* Do not lose the initial exception
* Do not generalize exceptions
* Avoid unnecessary wraps of exceptions

Optional

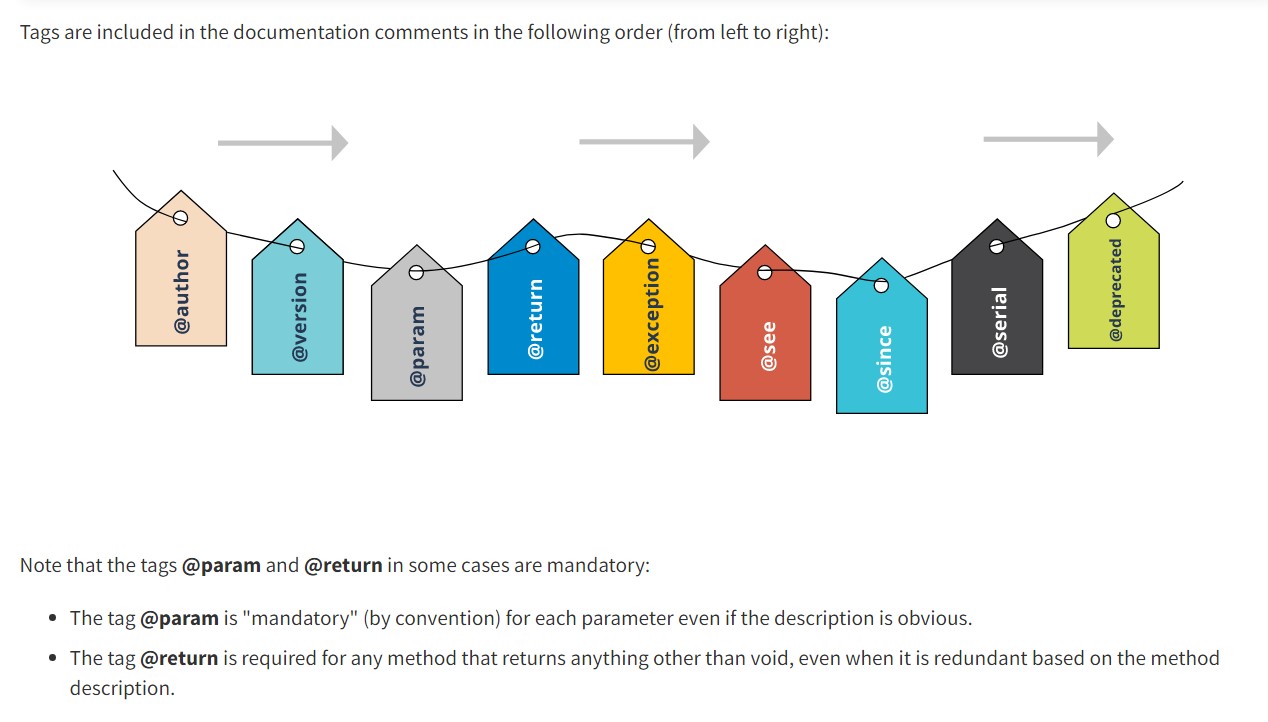
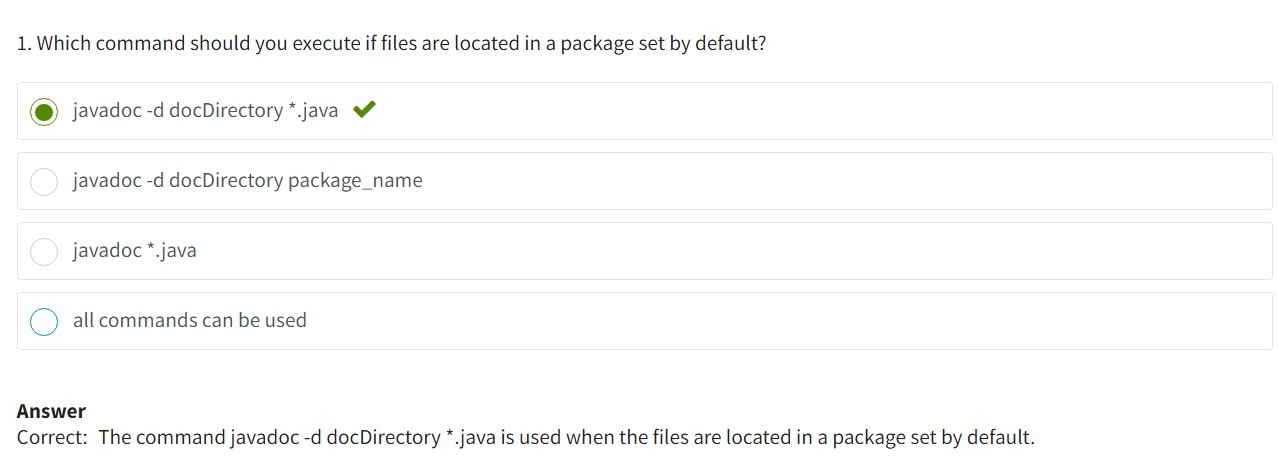
The **java.util.Optional** class is a container for any object type. This class is used to identify the problems of returning a **null** value using the method to perform search, perform processing, or generate any object in a situation when this action gave no acceptable result.

When a developer uses a method returning some reference to a value, this means that the developer is planning to use the returned object. If the method returns **null**, then using it will certainly throw a **NullPointerException**. The developer will have to add a special check for **null** to the code to protect it from the exception. The **Optional** class does not solve the problem of returning **null**, but using this class as a returned value clearly identifies this problem.

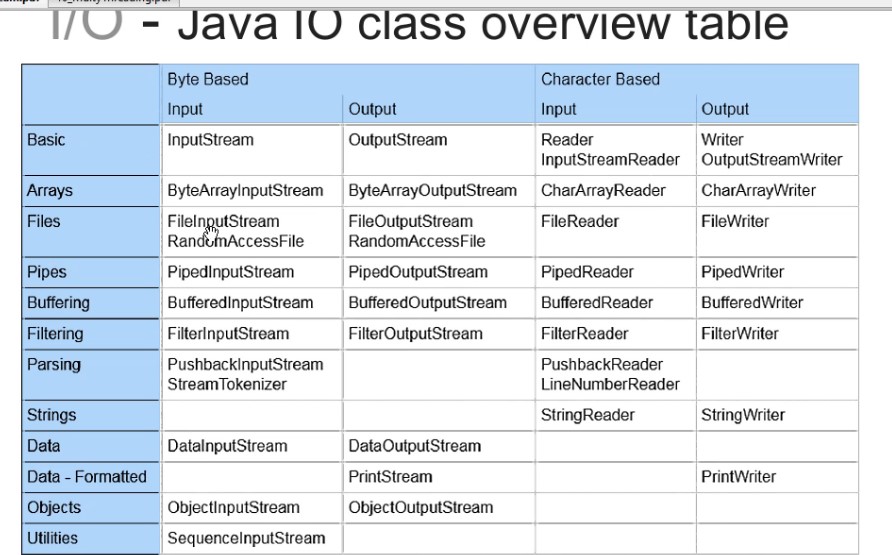


* Methods of verifying and returning a result get(), isPresent(), ifPresent()
* Methods of returning a conditional result orElse(), orElseGet(), orElseThrow()

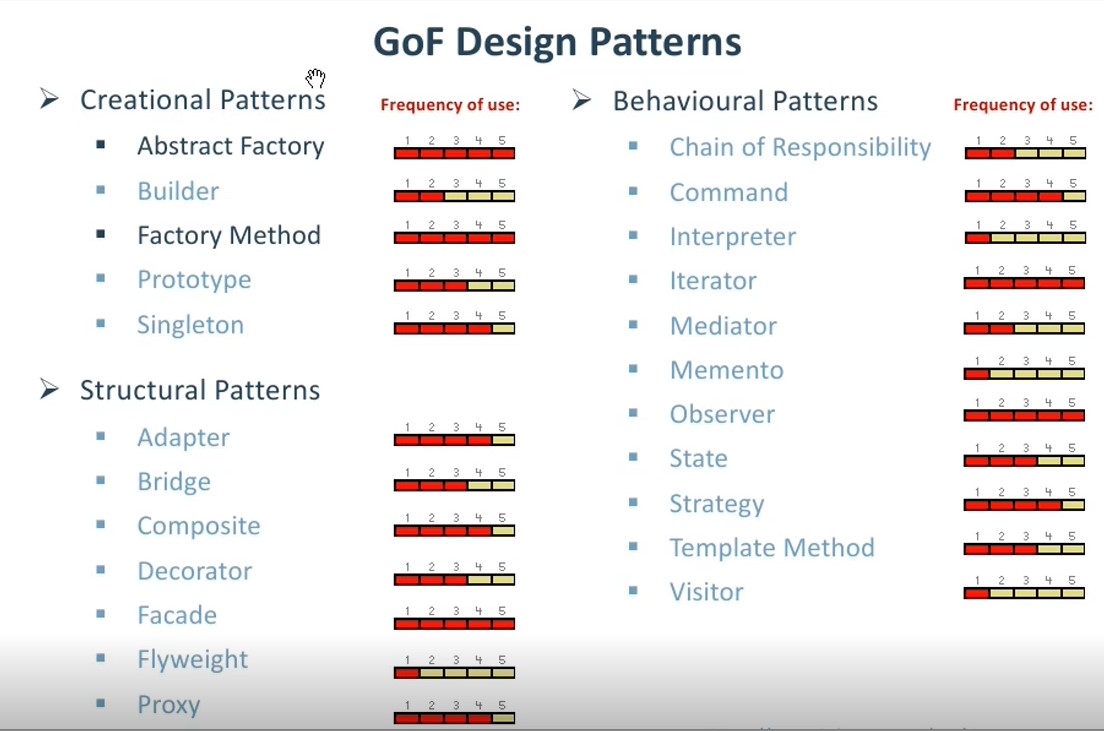
Javadoc



**IO**



**Patterns**



**Collections**

Collection -> Iterable

• **List**

• ArrayList (быстрый поиск)

• Vector <- Stack (sync)

• LinkedList (быстрая вставка)

**• Set**

• HashSet (сортирует по своему алгоритму, работает быстрее всех)

• LinkedHashSet (сохраняет порядок добавления, медленный поиск)

• TreeSet (сортирует элементы от меньшего к большему, функции поиска, вставки, удаления работают за log n)

• **Map**

• HashMap

• LinkedHashMap

• Hashtable (sync)

• TreeMap

• **Queue**

• LinkedList

• **Deque**

• LinkedList

• ArrayDeque