1)Метод toString:

Метод toString служит для получения представления данного объекта в виде строки. Полученное мной значение (в данном случае Person@7960847b) вряд ли может служить хорошим строковым описанием объекта. Поэтому метод toString() нередко переопределяют.

1.1)Метод hashCode:

Метод **hashCode** позволяет задать некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код. По данному числу, например, можно сравнивать объекты.

1.2)Метод getClass:

Метод getClass позволяет получить тип данного объекта:

1.3)Метод equals:

Метод equals сравнивает два объекта на равенство. Метод equals принимает в качестве параметр объект любого типа, который мы затем приводим к текущему, если они являются объектами одного класса.

Оператор **instanceof** позволяет выяснить, является ли переданный в качестве параметра объект объектом определенного класса, в данном случае класса Person. Так как если объекты принадлежат к разным классам, то их сравнение не имеет смысла, и возвращается значение false.

2) 3)Методы по умолчанию в интерфейсах:

Для создания метода по умолчанию в интерфейсе, мы должны использовать ключевое слово default. Вообразим, что в представленном выше интерфейсе метод log(String str) является методом по умолчанию. Теперь, когда какой-либо класс будет реализовывать интерфейс Interface1, не является обязательным обеспечить реализацию методов по умолчанию (в нашем случае — это метод log(String str)).

Мы знаем, что Java не позволяет нам [наследоваться от нескольких классов](https://javadevblog.com/nasledovanie-v-java-prosto-i-na-primerah.html), потому что это приведет к [ромбовидной проблеме](https://javadevblog.com/mnozhestvennoe-nasledovanie-v-java-i-kompozitsiya-vs-nasledovaniya.html), где компилятор не может решить, какой метод суперкласса использовать. Теперь же с появлением методов по умолчанию, эта проблема возникнет и для интерфейсов! Все дело в том, что если класс реализует как Interface1, так и Interface2 и не реализовывает общий метод по умолчанию, то компилятор не может решить что выбрать.

Наследование нескольких интерфейсов является неотъемлемой частью Java, поэтому нам нужно следить за тем, чтобы эта проблема не возникала и в интерфейсах.

4) Коротко о главном. Методы по умолчанию в интерфейсах:

·Методы по умолчанию помогаю реализовывать интерфейсы без страха нарушить работу других классов.

·Методы по умолчанию в [Java 8](https://javadevblog.com/obzor-vozmozhnostej-java-8-funktsional-ny-e-interfejsy-lyambda-vy-razheniya-stream-i-time-api.html" \t "_blank) позволяют избежать создания служебных классов, так как все необходимые методы могут быть представлены в самих интерфейсах.

·Методы по умолчанию дают свободу классам выбрать метод, который нужно переопределить.

·Одной из основных причин внедрения методов по умолчанию является возможность коллекций использовать лямбда-выражения.

·Если какой-либо класс в иерархии имеет метод с той же сигнатурой, то методы по умолчанию становятся неактуальными. Метод по умолчанию не может переопределить метод класса java.lang.Object. Аргументация очень проста: это потому, что объект является базовым классом для всех Java-классов. Таким образом, даже если у нас есть методы класса Object, определенные в качестве методов по умолчанию в интерфейсах, это будет бесполезно, потому что всегда будет использоваться метод класса объекта. Вот почему, чтобы избежать путаницы, мы не можем писать стандартные методы, которые переопределяли бы методы класса Object.

5)Статические методы в интерфейсах:

Статические методы похожи на методы по умолчанию, за исключением того, что мы не можем переопределить их в классах, реализующих интерфейс. Этот функционал помогает нам избежать нежелательных результатов, которые могут появиться в дочерних классах. Статические методы видны только для методов интерфейса.

6) Коротко о главном. Статические методы в интерфейсах:

·Статические методы в интерфейсе являются частью интерфейса, мы не можем использовать его для объектов класса реализации.

·Статические методы в интерфейсе хороши для обеспечения вспомогательных методов, например, проверки на null, сортировки коллекций и т.д.

·Статические методы в интерфейсе помогают обеспечивать безопасность, не позволяя классам, которые реализуют интерфейс, переопределить их.

·Мы не можем определить статические методы для методов класса Object, потому что получим ошибку компиляции: «*This static method cannot hide the instance method from Object*«. Это потому, что в Java так делать нельзя . То есть Object является базовым классом для всех классов и мы не можем использовать статический метод и еще такой метод с одинаковой сигнатурой.

·Мы можем использовать статические методы интерфейса, чтобы не создавать вспомогательные классы, то есть переместить все статические методы в соответствующий интерфейс. Такой метод легко использовать и быстро находить.

7)Разрешение конфликтов с методом по умолчанию:

3) Метод, объявленный с модификатором **final** не может быть переопределен в наследниках.

Поле с модификатором **final** не может поменять своего значения после инициализации.

Если поле класса объявлено как **static**, то будет существовать ровно одно значение этого поля, не зависимо от того, сколько экземпляров класса будет создано, даже если не будет создано ни одного экземпляра.

Как видите, если обращаться к**статическим методам** и через название объекта, и название класса, код будет работать. К нестатическим методам нужно обращаться исключительно через название объектов класса. **Статические методы нельзя переопределять.**

**4) Принципы OOП Stupid:**

Это может задеть Ваше самолюбие, но вы уже, вероятно, написали много STUPID кода. Я тоже. Но, что это значит?

* Синглтон
* Сильная Связанность/Tight Coupling
* Невозможность тестирования
* Преждевременная оптимизация
* Не дескриптивное присвоение имени
* Дублирование кода

**5) Принципы OOП Solid:**

SOLID значит:

* Принцип единственной обязанности( принцип [ООП](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), обозначающий, что каждый [объект](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) должен иметь одну ответственность и эта ответственность должна быть полностью [инкапсулирована](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) в [класс](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). Все его поведения должны быть направлены исключительно на обеспечение этой ответственности.)
* Принцип открытости/закрытости
* Принцип подстановки Барбары Лисков(Пусть q(x) является свойством, верным относительно объектов x некоторого типа T. Тогда q(y) также должно быть верным для объектов y типа S, где S является подтипом типа T.)
* Принцип разделения интерфейса
* Принцип инверсии зависимостей(Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
* Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.)

**Ссылки:**<https://itnan.ru/post.php?c=1&p=273843>  
<https://itnan.ru/post.php?c=1&p=273843>