**Модификаторы НЕ ДОСТУПА**

Модификаторы классов (class modifiers) - **abstract, final, static, strictfp**.

О модификаторе strictfp знать не критично. Если интересно - читайте о нем в конце статьи.

Модификатор **abstract**, примененный к классу, говорит о том, что класс является (или считается) незаконченным, а задание "завершить" класс возлагается на наследников. Попытка инстанциировать такой класс приведет к ошибке компиляции, например:

**public** **abstract** **class** Expression {

Expression e = **new** Expression();

}

В результате компиляции этого кода получим ошибку: Expression is abstract; cannot be instantiated.

Обратите внимание, у абстрактного класса не обязательно должны быть абстрактные методы.

Модификатор **final** у класса говорит о том, что от него нельзя наследоваться.

**public** **final** **class** Example {

**class** Subclass **extends** Example { }

}

Попытка компиляции кода приведет к ошибке: cannot inherit from final Example.

Очевидно, класс нельзя объявить одновременно final и abstract.

Вложенные классы в Java могут быть объявлены как **static**, например:

**public** **class** Outer {

**static** **class** Inner { }

}

В этом случае класс называется статическим вложенным классом и имеет доступ к статическим полям и методам обрамляющего класса.

Модификаторы методов (method modifiers) -  **abstract, final, static, strictfp** (читайте в конце статьи), **native, synchronized**.

С методами вариантов немного больше. Методы в Java могут быть объявлены как

Метод с модификатором **abstract** может быть объявлен как метод-член (member method) в пределах абстрактного класса (или интерфейса). В этом случае тело метода отсутствует, а реализация может быть предоставлена в классах-наследниках. Если же метод объявлен как абстрактный в конкретном классе, то получим ошибку компиляции.

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

**public** **class** Clazz {

**public** **final** **void** a() { }

**class** SubClass **extends** Clazz {

**public** **final** **void** a() { }

}

}

В приведенном примере возникнет ошибка на этапе компиляции: a() in Clazz.SubClass cannot override a() in Clazz; overridden method is final

Метод с модификатором **static** относится к классу в целом, а не к его экземплярам, то есть в него не передается объект this. Такой метод может быть вызван используя имя класса. Например:

**public** **class** Sample {

**static** a() { }

**public** **static** **void** main(String...args) {

Sample.a();

}

}

Java Native Interface (JNI) — стандартный механизм для запуска кода, под управлением виртуальной машины Java (JVM), который написан на языках С/С++ или Ассемблера, и скомпонован в виде динамических библиотек, позволяет не использовать статическое связывание. Это даёт возможность вызова функции С/С++ из программы на Java, и наоборот.

Модификатор **native** сигнализирует о том, что метод реализован в платформо-зависимом коде, часто на языке С.

Этот модификатор может быть применен только к методам, но не классам и переменным.

Тело нативного метода должно заканчиваться на (;) как в абстрактных методах, идентифицируя то, что реализация опущена.

***Как я понял:*** Модификатор **Native** сигнализирует о том, что в методе будет использован программный код написанный на языке С\С++ для возможности оперирования указателями (как в опасном коде). Опасный код - это обыкновенный код, не контролируемый платформой .NET (которая следит за выполнением промежуточного кода).

Примеры объявления таких методов можно найти,например, в классе java.io.FileInputStream:

**private** **native** **void** open(String name) **throws** FileNotFoundException;

**private** **native** **int** readBytes(**byte** b[], **int** off, **int** len) **throws** IOException;

Наконец, модификатор **synchronized** у метода говорит о том, что перед его выполнением должен быть захвачен монитор объекта (для нестатического метода), либо монитор, связанный с классом (для статического метода).

Вот небольшой пример кода, который демонстрирует описание синхронизированных методов. Захват монитора так же осуществляется с помощью ключевого слова synchronized.

**public** **class** Synch {

**public** **synchronized** **void** a() { }

**public** **static** **synchronized** **void** b() { }

**final** **static** Synch s = **new** Synch();

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

**synchronized** (s) {

// lock on object

}

**synchronized** (Synch.class) {

// lock on class

}

}

}

Модификаторы полей (field modifiers) - **static, final, transient, volatile**.

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. Такие статические поля еще называют переменными уровня класса (class variable).

Поле с модификатором **final** не может поменять своего значения после инициализации. Это касается и статических, и нестатических полей (member fields).

**public** **class** Sample {

**final** **static** **int** constant = 5;

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

Sample.constant = 0;

}

}

Компиляция этого кода приведет к ошибке: cannot assign a value to final variable constant.

Для указания того, что во время сериализации объекта некоторое поле нужно игнорировать, используется модификатор **transient**. Обычно такие поля не являются частью внутреннего состояния объекта, либо хранят промежуточные значения.

С модификатором **volatile** все немного посложнее. Попытаюсь объяснить попроще. В Java потоки могут хранить значения некоторых переменных в некотором своем локальном контексте. Если один поток изменит значение такого поля, то другой поток может об этом не узнать (так как хранит копию). Для полей с модификатором volatile есть гарантия, что все потоки всегда будут видеть актуальное значение такой переменной.

Модификаторы, связанные с интерфейсами - **abstract, static, strictfp**. Интерфейс является abstract по-умолчанию. В случае со вложенным интерфейсом ключевое слово static можно не указывать - он в любом случае будет статическим.

Методы интерфейсов по-умлочанию являются public abstract, поэтому к ним не применимы модификаторы final, static и native. Синхронизированными они тоже быть не могут, так как интерфейс нельзя инстанциировать.

Поля интерфейсов по-умолчанию являются public static final, а значит должны быть проинициализированы.

Если подытожить, то два фрагмента кода ниже являются идентичными:

**public** **interface** Outer {

**int** a = 5;

**int** b();

**interface** Inner { }

}

**public** **abstract** **interface** Outer {

**public** **static** **final** **int** a = 5;

**public** **abstract** **int** b();

**public** **static** **abstract** **interface** Inner { }

}

Ну и **default** - последняя разработка в области модификаторов :). Метод интерфейса, помеченный как default, предоставляет реализацию этого метода по-умолчанию. Например:

**public** **interface** Test {

**void** method1();

**default** **void** method2() { }

}

Этот код скомпилируется без ошибок на Java версии 1.8.

Другие контексты использования модификаторов

Есть еще 2 варианта, где могут использоваться модификаторы.

Первый вариант - это локальные переменные, которые могут быть объявлены как final. В этом случае значение переменной нельзя будет изменить в в пределах метода после инициализации

Второй вариант - это статический блок инициализации, который выполняется при загрузке класса:

**public** **class** Test {

**static** **int** i;

**static** {

i = 5;

}

}

Ну и наконец, перейдем к модификатору **strictfp**. strictfp - это модификатор, введенный в java 1.2, ограничивающий точность вычислений с float и double по стандарту IEEE. Для чего это нужно? Чтобы обеспечить переносимость. Дело в том, что JVM использует всю возможную точность процессора, а она на разных системах разная, поэтому и результат может получиться разный.