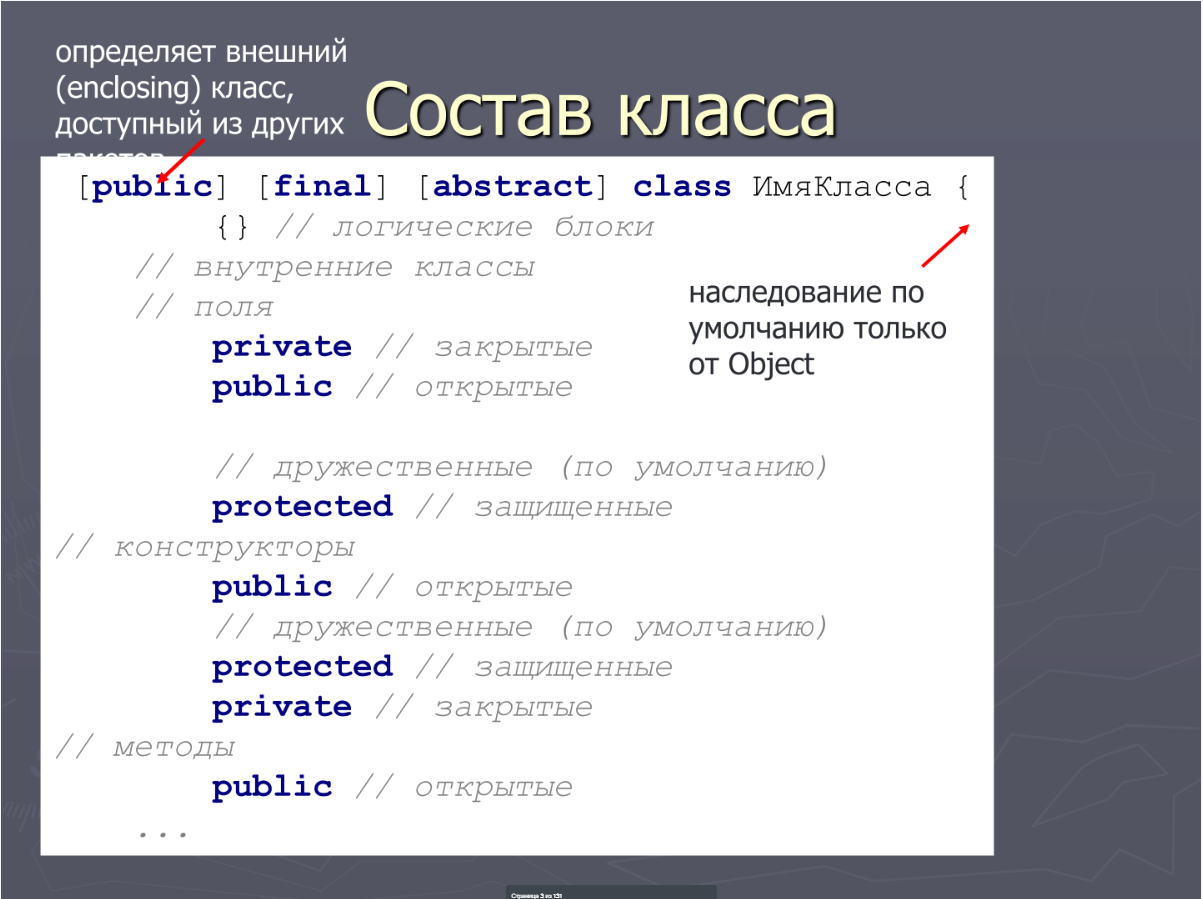
**АБСТРАКТНАЯ ФАБРИКА**

Предположим, что разрабатывается некое приложение, интерфейс пользователя которого содержит три модуля, представленных соответствующими классами: Locale (весь текст интерфейса), Images (картинки, баннеры) и Help (справка пользователя). Представим себе, что приложение предназначено для США и Японии. И в зависимости от того, какая страна выбрана в его настройках, должен загружаться соответствующий интерфейс: английская или японская локализация, баннеры, справка. Шаблон абстрактной фабрики подсказывает нам решение задачи в том, чтобы представить Locale, Images и Help интерфейсами, создать для них нужные нам реализации, а затем написать интерфейс фабрики UIAbstractFactory, в котором будут указаны фабричные методы, инициализирующие необходимые нам объекты. Далее мы пишем две реализации фабрики, в котором имплементируем эти методы, а в коде нашего приложения просто выбираем, какую из фабрик мы хотим использовать, основываясь на параметре.

который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

Есть единственный глобальный объект **LogManager,** который используется, чтобы поддержать ряд общего состояния о службах журнала и Регистраторах. Глобальный объект LogManager может быть получен, используя LogManager.getLogManager (). Объект LogManager создается во время инициализации класса и не может впоследствии быть изменен. **Log4j** — библиотека журналирования Java программ

1. 1.

Ограничение доступа

-private—только членам данного класса;

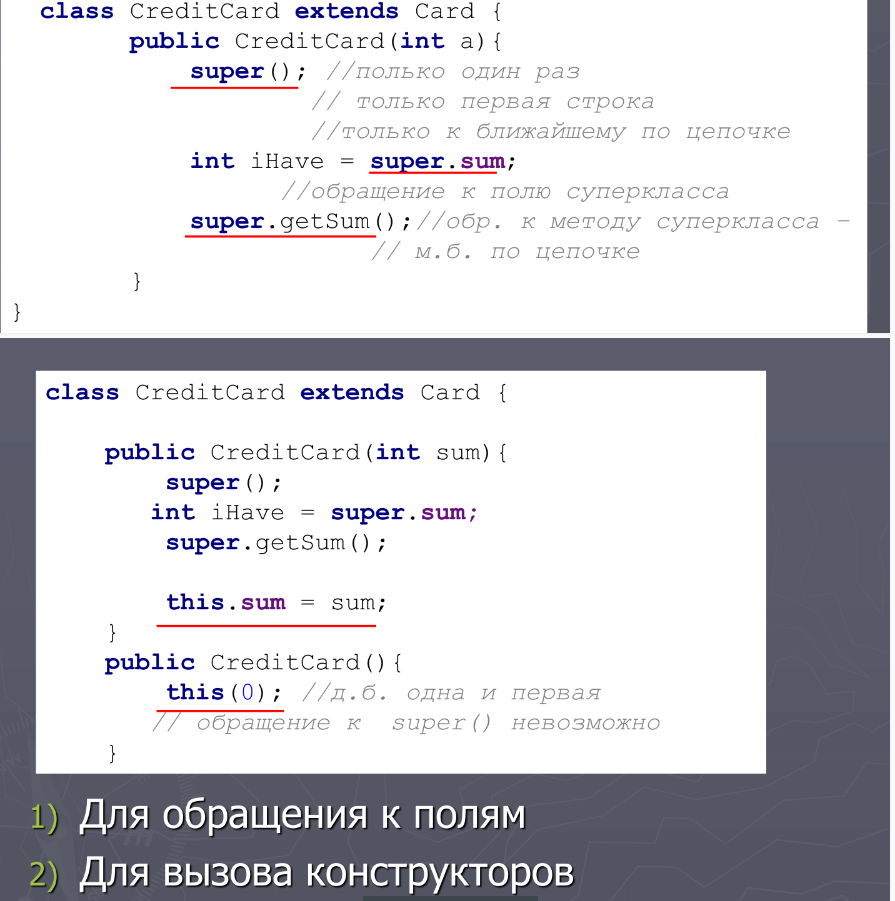
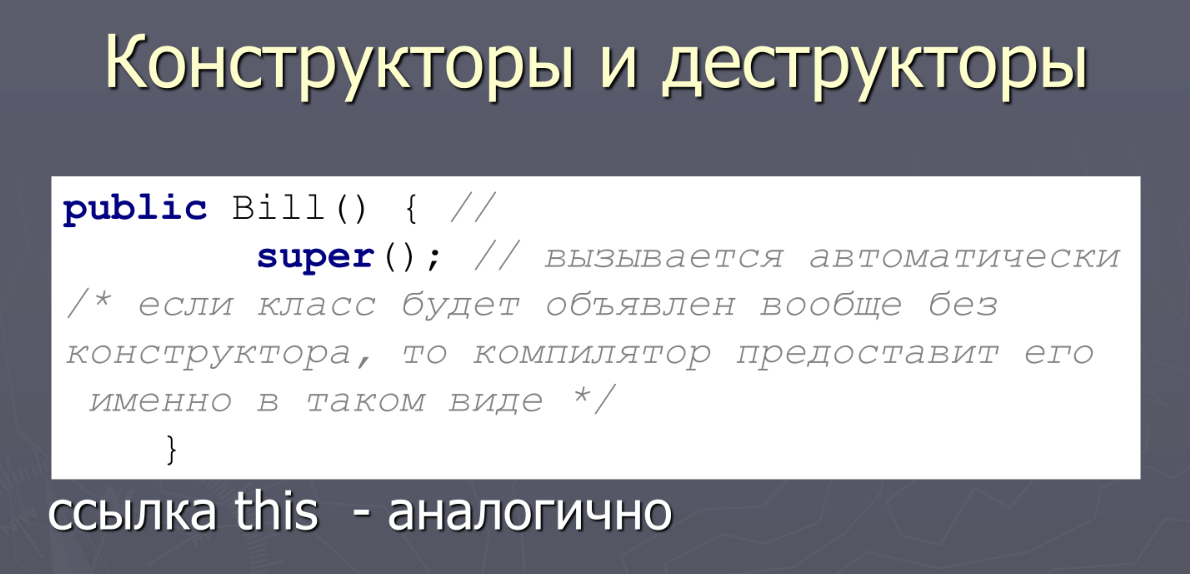
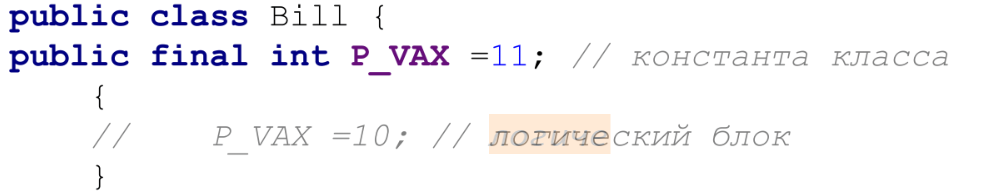
-protected—подклассам, находящимся в

том же пакете/в другом пакете

-public—для всех классов в этом и других пакетах.

1. **static int** *bonus// переменная класса*

**public final static int *P\_TAX*** = 6 *// константа класса*

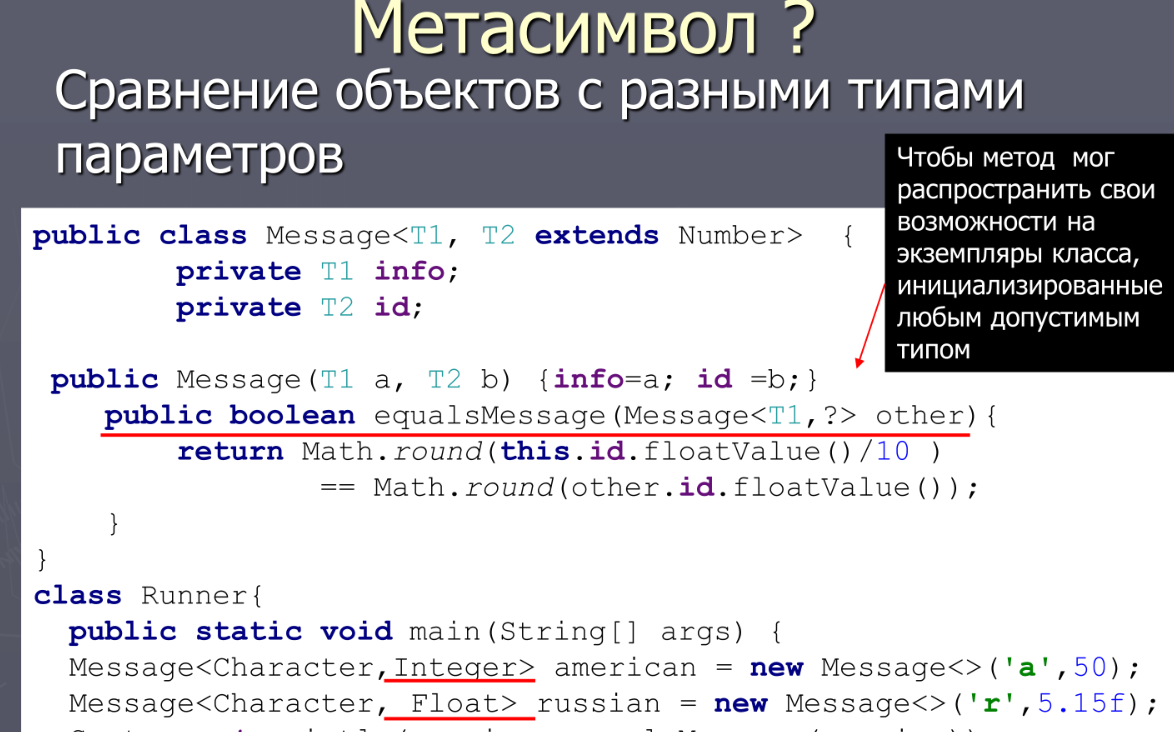
1. 
2. Иногда требуется написать подпрограмму, написанную на языке, отличном от языка Java, для повышения скорости выполнения. В Java предусмотрено ключевое слово **native**, которое используется для объявления машинно-зависимых методов. Это даёт возможность вызова функции С/С++ из программы на Java, и наоборот. сообщает компилятору, что метод реализован в другом месте
3. При описании класса могут быть использованы логические блоки. Логическим блоком называется код, заключенный в фигурные скобки и не принадлежащий ни одному методу текущего класса. Логические блоки чаще всего используются в качестве инициализаторов полей, но могут содержать вызовы методов и обращения к полям текущего класса. При создании объекта класса они вызываются последовательно, в порядке размещения, вместе с инициализацией полей как простая последовательность операторов, и только после выполнения последнего блока будет вызван конструктор класса. 
4. Параметризация позволяет создавать классы, интерфейсы и методы, в которых тип обрабатываемых данных задается как параметр. **public** **class** Subject <T1, T2> {

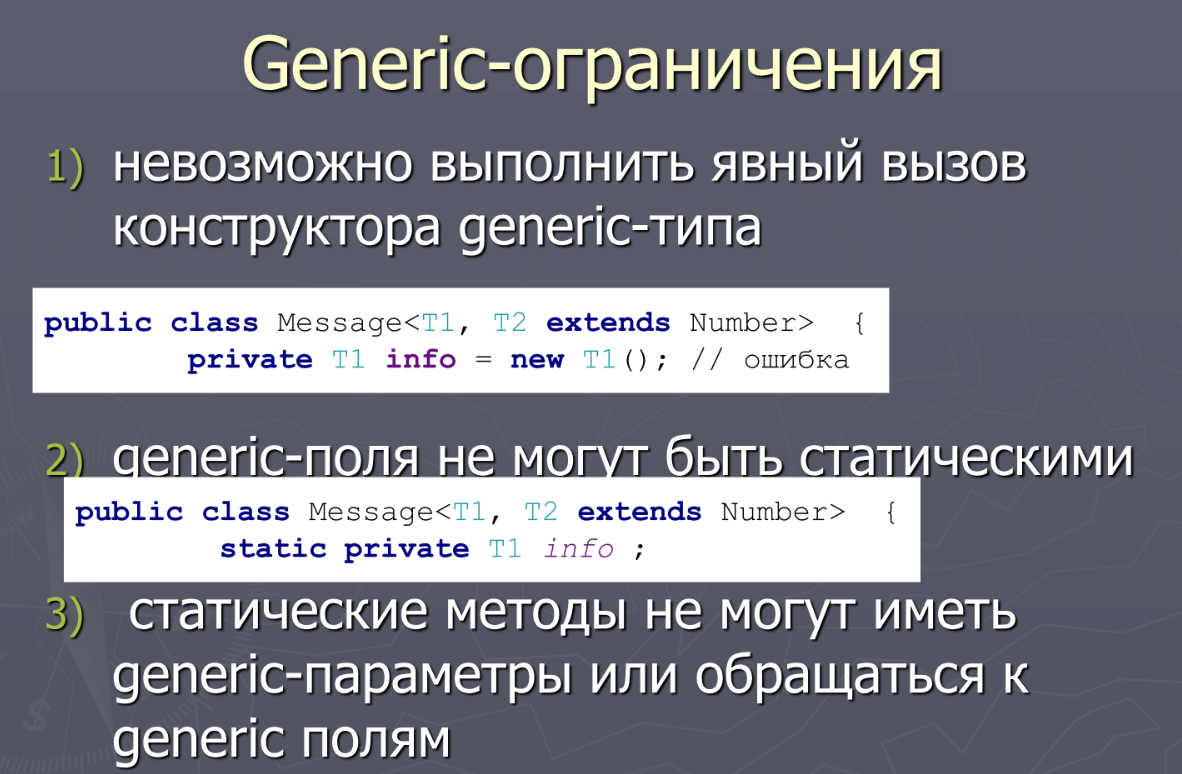
**private** T1 name;

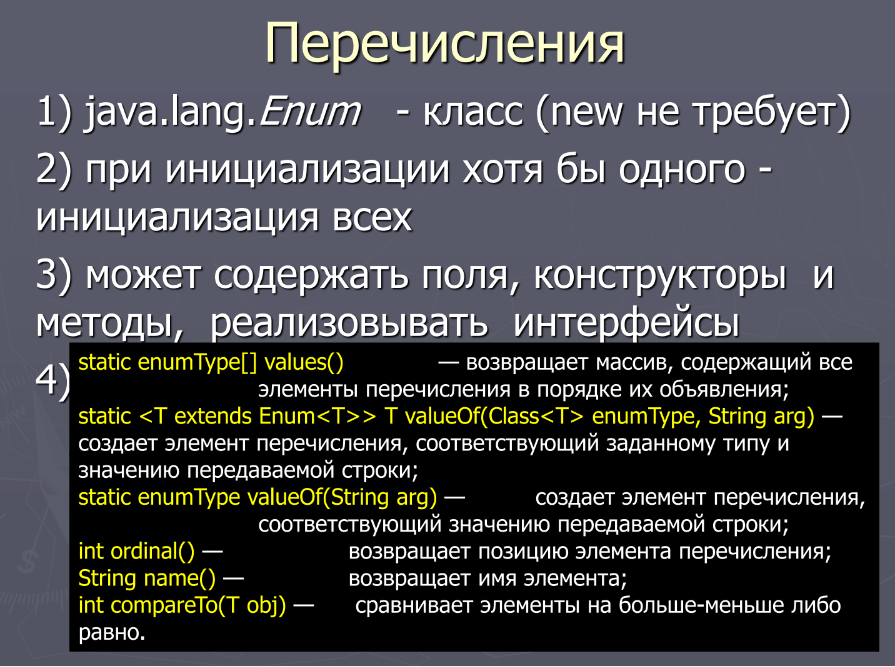
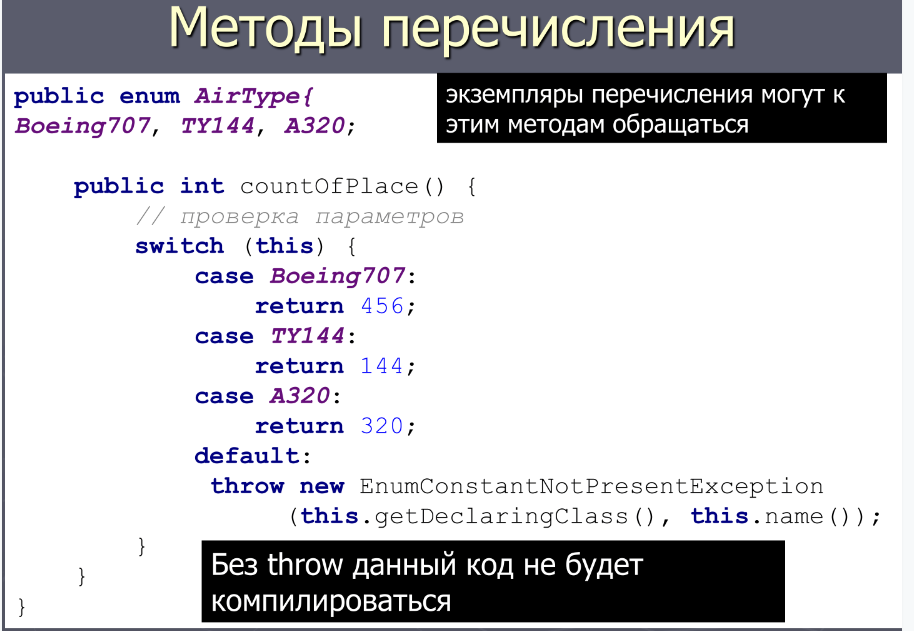
**private** T2 id;

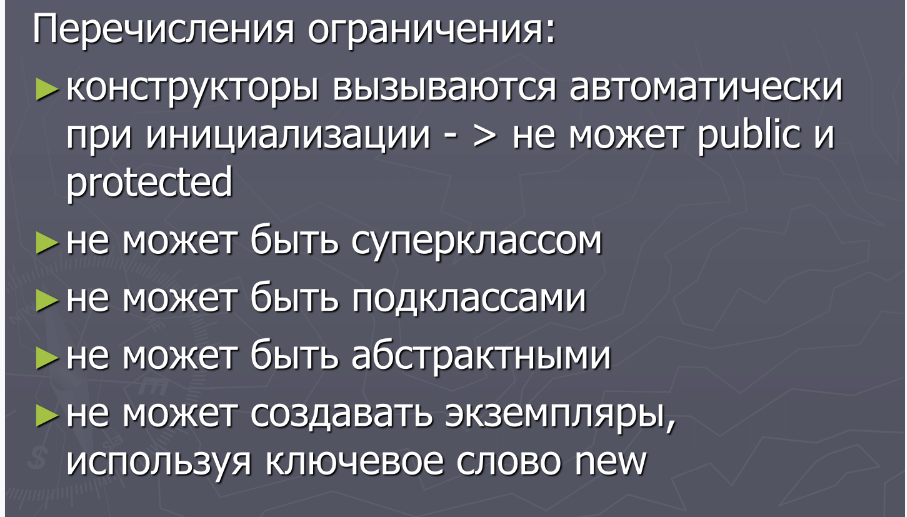
**public** Subject() {

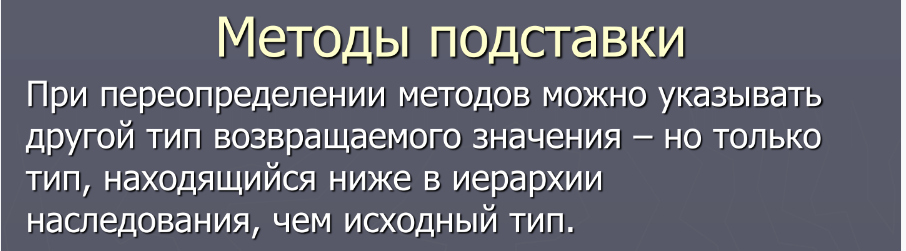
}

1. **?**     - (знак вопроса) означает, что предшествующий ему символ является необязательным (РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ) 

8.В качестве параметров классов запрещено применять базовые типы.

9. Перечисления представляют набор логически связанных констант 

10. 

**11.** 

### 12. toString

Метод toString служит для получения представления данного объекта в виде строки. При попытке вывести строковое представления какого-нибудь объекта, как правило, будет выводиться полное имя класса

**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.

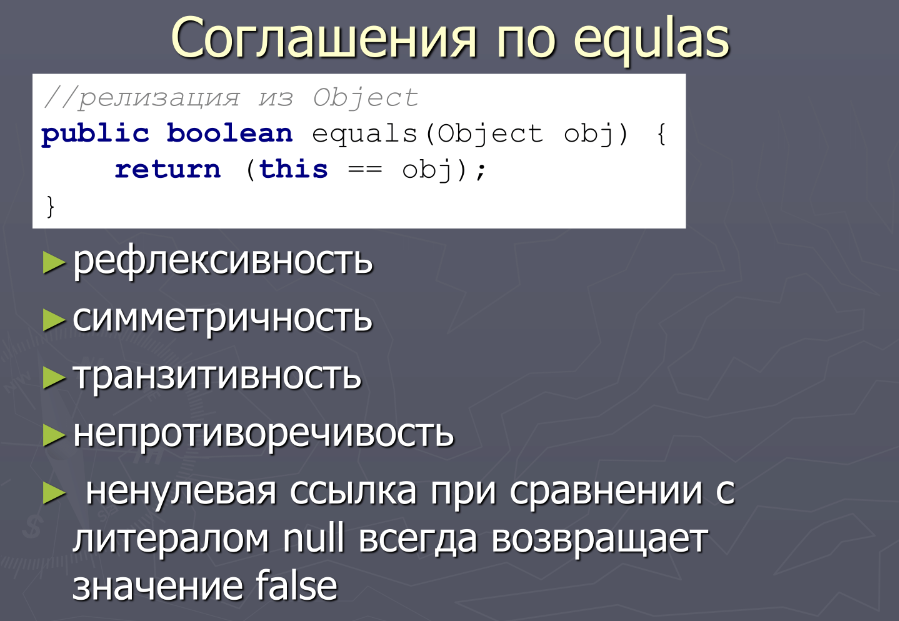
**Получение типа объекта и метод getClass**

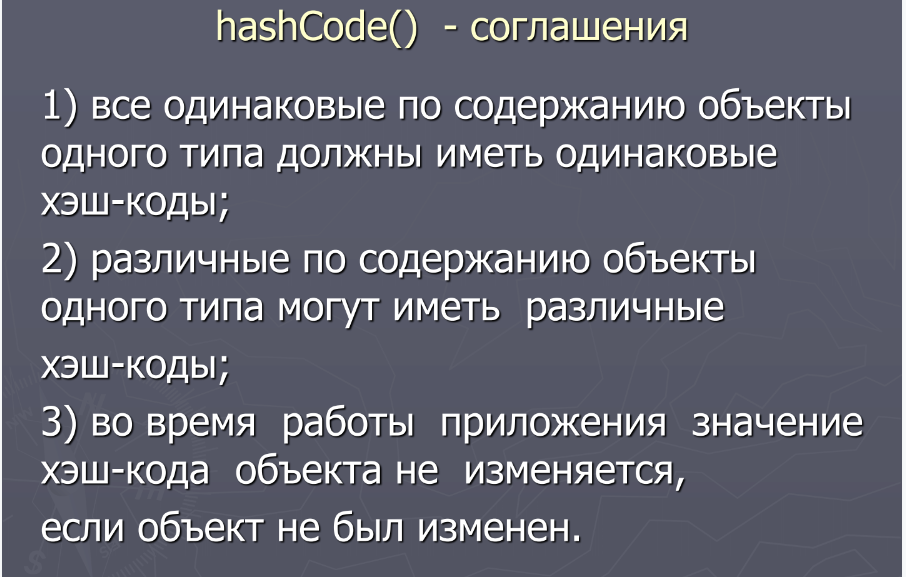
Метод getClass позволяет получить тип данного объекта

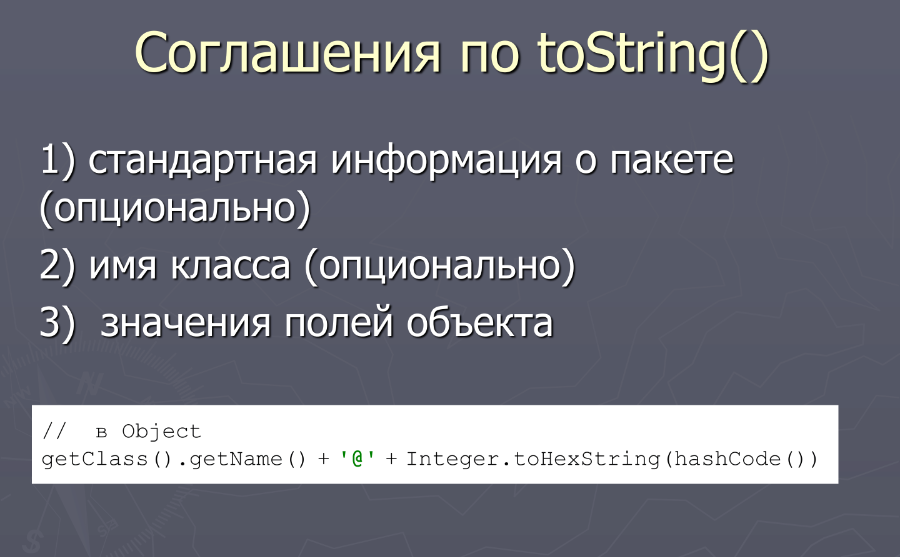
**Метод equals**

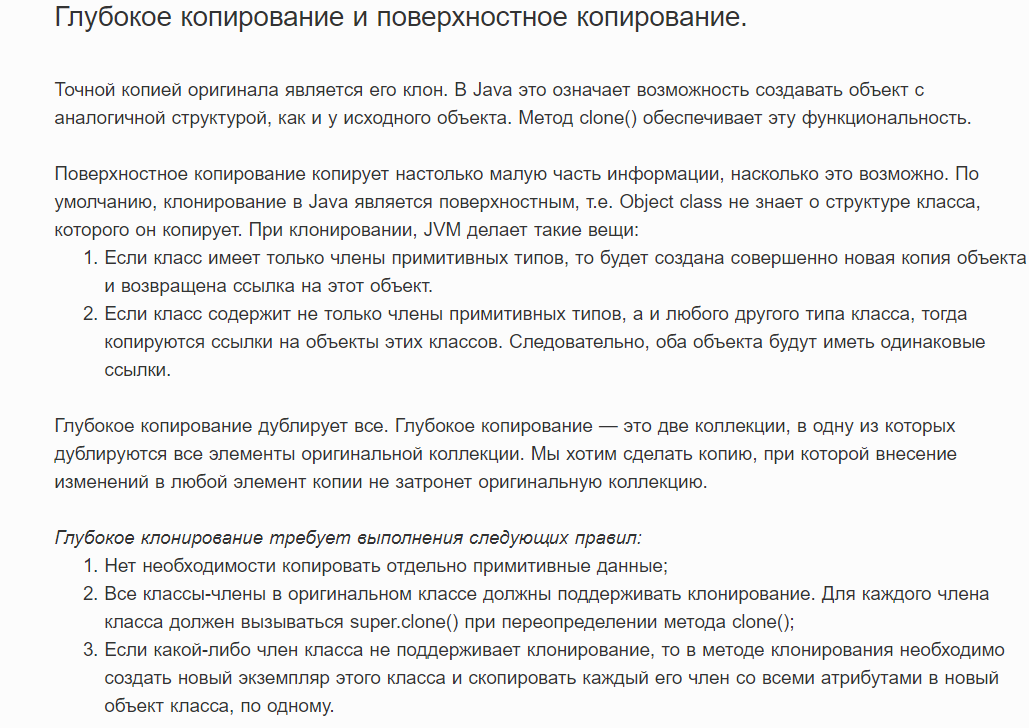
Метод equals сравнивает два объекта на равенство

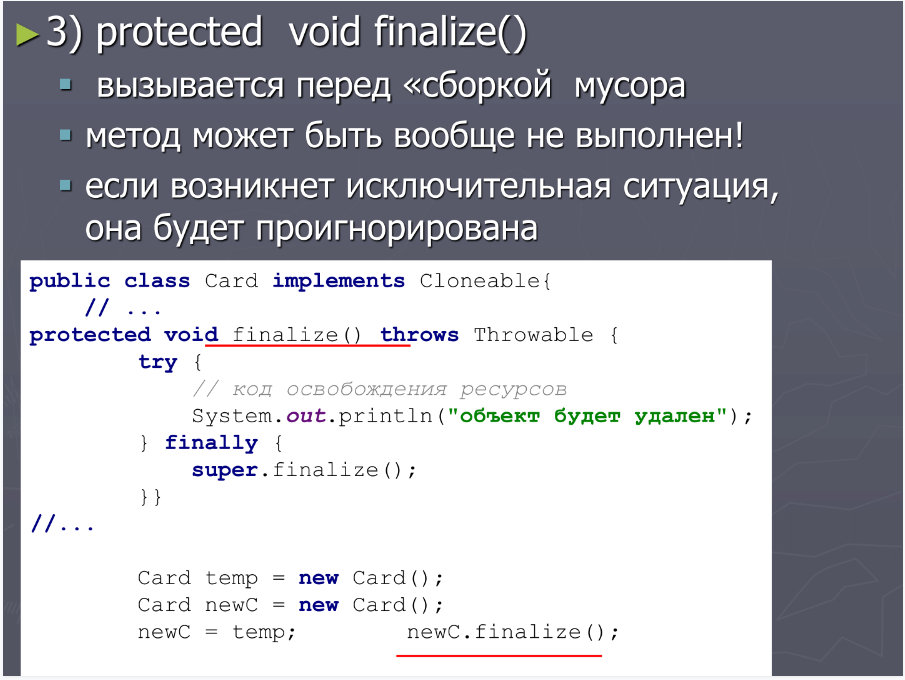
* Object clone() - создаёт новый объект, не отличающий от клонируемого
* boolean equals(Object obj) - определяет, равен ли один объект другому
* void finalize() - вызывается перед удалением неиспользуемого объекта
* Class<?> getClass() - получает класс объекта во время выполнения
* int hashCode() - возвращает хеш-код, связанный с вызывающим объектом
* void notify() - возобновляет выполнение потока, который ожидает вызывающего объекта
* String toString() - возвращает строку, описывающий объект
* void wait() - ожидает другого потока выполнения

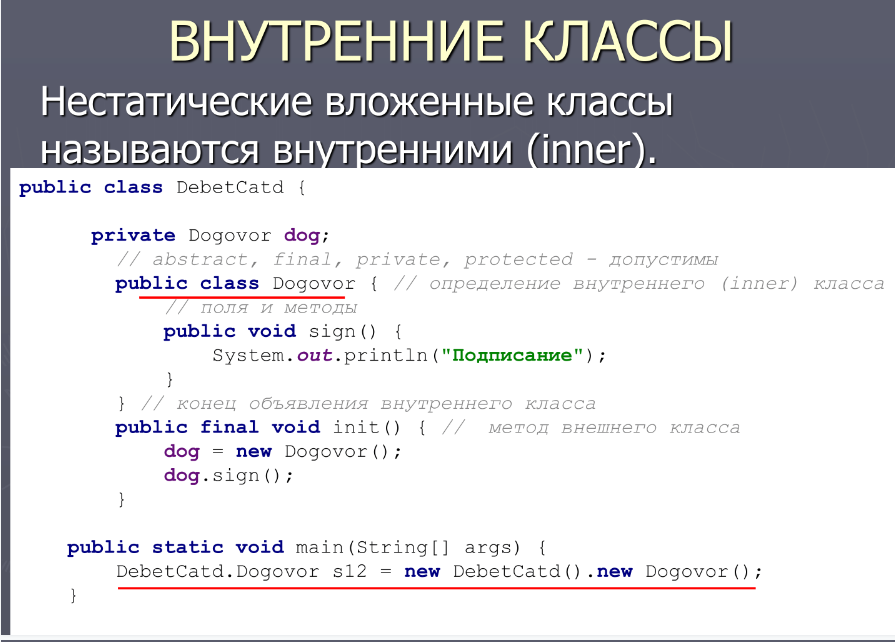
**13.** 

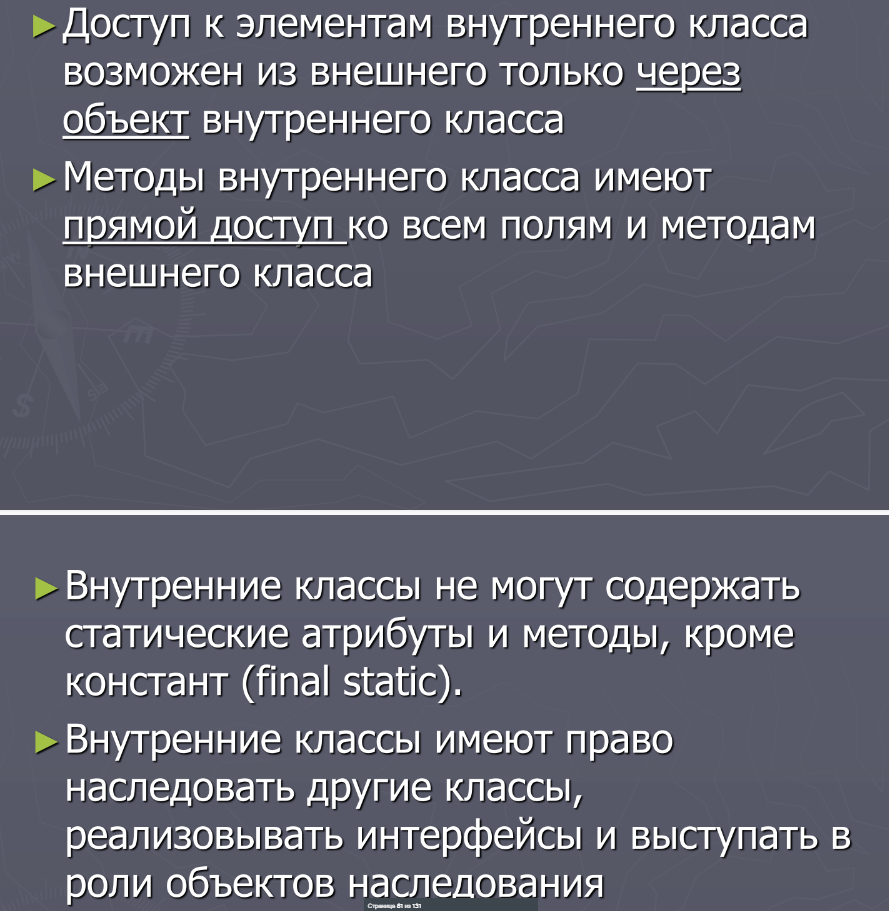
**14.** 

**15.** 

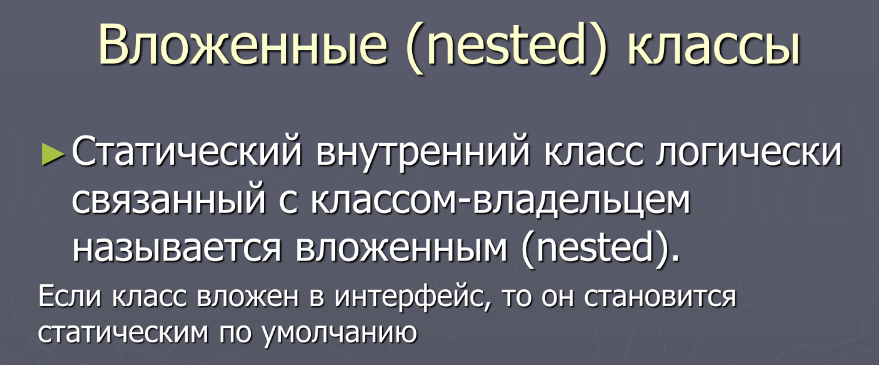
**16.** Когда вызвается метод clone родительского класса Object, происходит, так называемое, поверхностное копирование, в этом случае копируются значения простых полей и ссылочные значения. Если же нужно клонирование составного объекта, то это достигается через рекурсивное клонирование составляющих его объектов, так называемое, глубокое копирование.

**17.** 

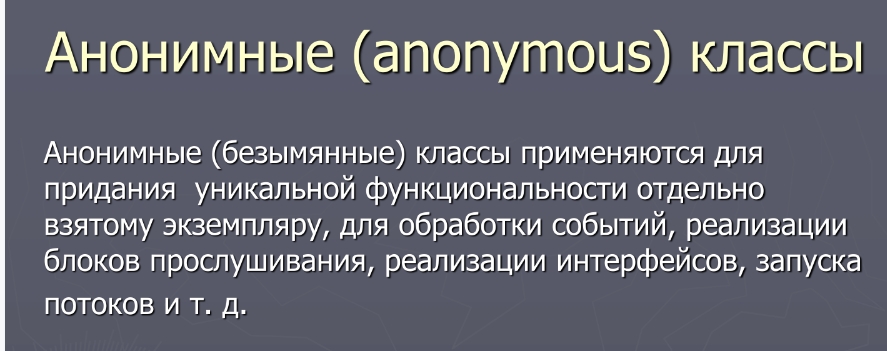
**18.** 

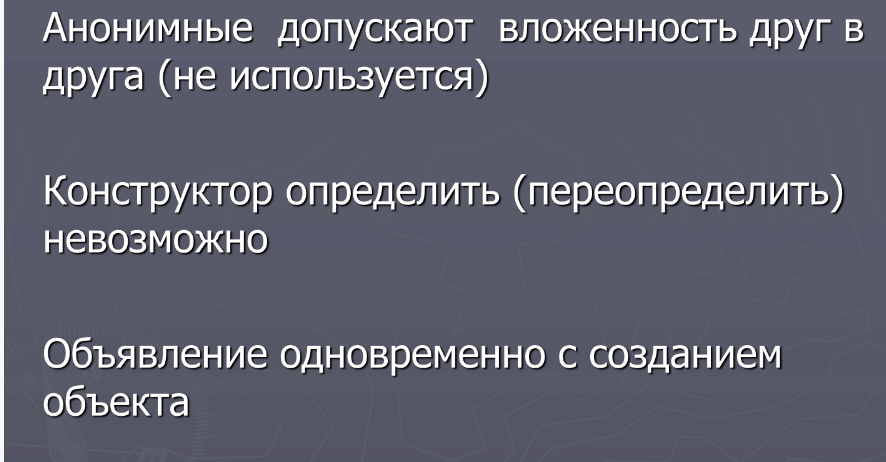
Внутренний класс может быть объявлен внутри метода или блока инициализации внешнего класса. 

## 19. Статические вложенные классы (static nested classes)



Статические вложенные классы, не имеют доступа к нестатическим полям и методам обрамляющего класса. Доступ к нестатическим полям и методам может осуществляться только через ссылку на экземпляр обрамляющего класса. Кроме этого, static nested классы имеют доступ к любым статическим методам внешнего класса, в том числе и к приватным.

**20.** 



**21.** Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. И один класс может применить множество интерфейсов.Как и в случае с классами, вы можете добавить перед словом **interface** спецификатор доступа **public** (но только если интерфейс определен в файле, имеющем то же имя) или оставить для него дружественный доступ, если он будет использоваться только в пределах своего пакета. Интерфейс может содержать поля, но они автоматически являются статическими (**static**) и неизменными (**final**). Все методы и переменные неявно объявляются как **public**.

Класс, который собирается использовать определённый интерфейс, использует ключевое слово **implements.**Все методы интерфейса не имеют модификаторов доступа, но фактически по умолчанию доступ **public**, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

Интерфейсов у класса может быть несколько, тогда они перечисляются за ключевым словом **implements** и разделяются запятыми. Интерфейсы могут вкладываться в классы и в другие интерфейсы. Если класс содержит интерфейс, но не полностью реализует определённые им методы, он должен быть об ъявлен как **abstract**. Интерфейсы — это не классы. С помощью ключевого слова **new** нельзя создать экземпляр интерфейса

**22.** 

**Unchecked**исключения в Java - наследованные от RuntimeException, checked - от Exception (не включая unchecked). Пример unchecked исключения - NullPointerException, checked исключения – IOException

В Java есть пять ключевых слов для работы с исключениями:

1. **try**- данное ключевое слово используется для отметки начала блока кода, который потенциально может привести к ошибке.
2. **catch**- ключевое слово для отметки начала блока кода, предназначенного для перехвата и обработки исключений.
3. **finally**- ключевое слово для отметки начала блока кода, которое является дополнительным. Этот блок помещается после последнего блока 'catch'. Управление обычно передаётся в блок 'finally' в любом случае.
4. **throw**- служит для генерации исключений.
5. **throws**- ключевое слово, которое прописывается в сигнатуре метода, и обозначающее что метод потенциально может выбросить исключение с указанным типом.

Исключения делятся на несколько классов, но все они имеют общего предка — класс Throwable. Его потомками являются подклассы Exception и Error. Исключения (Exceptions) являются результатом проблем в программе, которые в принципе решаемы и предсказуемы. Ошибки (Errors) представляют собой более серьёзные проблемы, которые, согласно спецификации Java, не следует пытаться обрабатывать в собственной программе, поскольку они связаны с проблемами уровня JVM.Checked исключения, это те, которые должны обрабатываться блоком catch или описываться в сигнатуре метода. Unchecked могут не обрабатываться и не быть описанными. Наличие\обработка **Checked** исключения проверяются **на этапе компиляции**. Наличие\обработка **Unchecked** исключения происходит **на этапе выполнения**.