Повторное использование классов.

**Повторное использование кода** (англ. code reuse) — методология проектирования компьютерных и других систем. Она заключающаяся в том, что система (компьютерная программа, программный модуль) частично либо полностью должна составляться из частей, написанных ранее компонентов и/или частей другой системы, и эти компоненты должны применяться более одного раза (если не в рамках одного проекта, то хотя бы разных). **Повторное использование** — основная методология, которая применяется для сокращения трудозатрат при разработке сложных систем.

Суть идеи, которую обозначают словом reuse, можно выразить следующим образом:

При решении задач используйте существующий стандартный набор инструментов и предоставляйте результаты своего труда в виде максимально общего набора инструментов, оформленного в соответствии со стандартами. То есть в виде простых, но достаточно мощных и общих функций (классов), снабженных документацией, с простыми очевидными примерами использования и обозначенными путями интеграции с другими инструментами (системами).

Самый распространённый случай повторного использования кода — **библиотеки программ**. Библиотеки предоставляют общую достаточно универсальную функциональность, покрывающую избранную предметную область. Примеры: библиотека функций для работы с комплексными числами, библиотека функций для работы с 3D-графикой, библиотека для использования протокола TCP/IP, библиотека для работы с базами данных. Разработчики новой программы могут использовать существующие библиотеки для решения своих задач и не «изобретать велосипеды».

Одной из наиболее притягательных возможностей языка Java является возможность повторного использования кода. Но что действительно "революционно", так это наличие возможности выполнять не только простое копирование и изменение этого кода.

# Композиция

**Композиция** — это объединение объектов, которые общаются между собой с помощью только внешних интерфейсов (black box reuse). Друг для друга объекты являются «чёрными ящиками».

Гибкость композиции объектов обеспечивает делегирование — передача ответственности за выполнение метода связанному объекту. Есть две разновидности делегирования:

**Методы-заглушки**, явно передающие вызов связанному объекту.

**Методы-делегаты**, передаваемые из связанного объекта в главный. Связанный объект при этом, как правило, не принадлежит главному, а существует сам по себе (с заглушками ситуация обратная). Можно даже сказать, что главный объект является вспомогательным для связанного.

Разумеется, при передаче делегата должна сохраняться привязка к свободным переменным в первоначальном контексте метода (в т.ч. переменных-членов его класса), иначе это не делегат, а просто указатель на функцию. По сути, процедура, сохраняющая привязку к свободным переменным в своём лексическом контексте, является замыканием (closure). Подразумевается, что процедура не обязана иметь имени**. В Java делегатов нет**, но есть возможность получить аналогичный эффект с помощью механизма Reflection, или с помощью внутреннего класса (inner class).

**Синтаксис композиции:** До сих пор, композиция достаточно часто использовалась, Вы просто помещали ссылку на объект внутрь нового класса. Для примера, представьте себе, что Вы хотите получить объект, который хранит различные объекты типа String, пару примитивных типов и объект другого класса. Для не примитивных объектов Вы помещаете ссылки внутри вашего класса, но примитивные типы Вы определяете напрямую.

***Пример1.***

# Наследование

**Наследование** (inheritance, white box reuse) — это формирование подкласса, который имеет доступ ко всем данным и методам родительского класса и может перегружать методы. О «чёрном ящике», как при композиции, речи нет. Наследование часто считают одним из основных принципов ООП. Наследование, к тому же, (a) нарушает принцип инкапсуляции, т.к. родительский класс открывает свои данные подклассу, и (b) действует без гибкости, «раз и навсегда»: после создания объекта наследованного класса его базовый класс уже не изменить. В известной книге «Банды четырёх» даётся совет предпочитать композицию наследованию. Создатель Java Джеймс Гослинг также признался, что считает наследование не лучшей техникой.

**Синтаксис наследования**: Наследование является неотъемлемой частью Java, впрочем, как и других ОО языков программирования. Это очевидно - Вы всегда осуществляете операцию наследования, когда создаете класс, даже если ваш класс не является наследником какого-либо другого, потому, что Вы неявно наследуете стандартный корневой класс Java Object. Синтаксис наследования похож на композицию, но процедура выполнения заметно отличается. Когда Вы наследуете, Вы "говорите": "Этот класс такой же, как тот старый класс!" Вы излагаете эту фразу в коде давая классу имя, как обычно, но до того, как начнете работать с телом класса, добавляете ключевое слово extends следующее до имени базового класса. Когда вы сделаете это, вы автоматически получите все поля данных и методы базового класса.

***Пример2.***

**Инициализация базового класса**: Осуществить инициализацию в конструкторе, путем вызова конструктора базового класса, который имеет все необходимые сведения и привилегии для осуществления инициализации самого базового класса. Java автоматически вставляет вызов базового класса в конструктор произошедшего (наследуемого) от этого класса.

***Пример3.***

**Конструктор с аргументами**: Если Ваш класс не имеет аргументов по умолчанию или если Вы хотите вызвать конструктор базового класса, который имеет аргументы, Вы должны просто использовать ключевое слово super и передать ему список аргументов.

***Пример4.***

**Объединение композиции и наследования**: Совместное использование композиции и наследования часто и широко используется при программировании.

***Пример5.***

**Выборочная композиция против наследования**: Композиция в основном используется, когда Вам нужно использовать возможности существующего класса, но не использовать его интерфейс. Это значит, что Вы внедряете объект, так, что вы можете использовать его для получения доступа к функциональности внедряемого объекта в вашем новом классе, но пользователь вашего нового класса видит интерфейс вашего нового класса раньше, чем интерфейс внедряемого объекта. Что бы добиться такого эффекта, Вы должны включать private объекты существующих классов внутрь вашего нового класса. Иногда требуется разрешить пользователю класса получить доступ к вашему новому классу напрямую; что бы сделать это, нужно сделать объекты public. Эти объекты используют реализацию скрытия самих себя, так что такой подход достаточно безопасен. Если пользователь знает, что Вы собрали этот класс из различных частей, то интерфейс этого класса будет для него более легок в понимании.

***Пример6.***

При наследовании, Вы берете существующий класс и создаете специальную его версию. В основном это означает, что Вы берете главный, целевой класс и приспосабливаете его для частных нужд. Немного поразмыслив, Вы увидите, что нет разницы при создании класса car используя объект vehicle - car не содержит vehicle, он и есть vehicle. Отсюда связь он и есть используется в наследовании, а содержит при композиции.

# Protected

В идеальном мире, **private** объекты всегда являются действительно **private**, но в реальных проектах, где вы пытаетесь во многих местах скрыть от внешнего мира нечто, Вам часто нужна возможность получить к нему доступ из классов наследников. Ключевое слово **protected** поэтому не такая уж, и ненужная назойливость или догма. Оно объявляет "Этот объект частный (**private**), если к нему пытается подобраться пользователь, но он доступен для всех остальных находящихся в том же самом пакете(**package**)". То есть, **protected** в Java автоматически означает **friendly**.

# Final

В Java ключевое слово final имеет слегка разные значения в зависимости от контекста, но в основном, оно определяется так "Это не может быть изменено". Вы можете хотеть запретить изменения по двум причинам: дизайн или эффективность. Поскольку эти две причины слегка различаются, то существует возможность неправильного употребления ключевого слова final.

## Данные final

При использовании final с объектами, а не с примитивными типами получается несколько не тот эффект. С примитивами, final создает константу значения, а с объектами - ссылку, final создает ссылку - константу. Как только ссылка инициализируется на какой-то объект, она уже не может быть впоследствии перенаправлена на другой объект. Однако сам объект может быть модифицирован; Java не предоставляет способа создать объект - константу. (Однако, Вы можете написать свой собственный класс с эффектом константы.) Эти же ограничения накладываются и на массивы, поскольку они тоже объекты.

***Пример7.***

## Пустые final

Java позволяет создавать пустые (чистые) final объекты (*blank final)*, это такие поля данных, которые были объявлены как **final** но при этом не были инициализированы значением. Во всех случаях, пустая final переменная *должна* быть инициализирована до ее использования, и компилятор обеспечивает это условие. Тем не менее, пустые final поля предоставляют большую гибкость при использовании модификатора **final**, к примеру, **final** поле внутри класса может быть разным для каждой копии объекта.

***Пример8.***

## Аргументы final

Java позволяет Вам так же создавать и аргументы final определением их таким образом прямо в списке аргументов. Это означает, что внутри метода Вы не сможете изменить этот аргумент или его ссылку.

## Final методы

Существует две причины для **final** методов. Первая - закрытие методов, от возможной модификации при наследовании класса. Такой подход применяется если Вы хотите быть уверенны, что этот метод не будет переопределен в дочерних классах и поведение класса не изменится.

Вторая причина - **final** методы более эффективны. Если Вы делаете метод с модификатором **final**, Вы тем самым разрешаете компилятору все вызовы этого метода превратить во *внутренние (inline)* вызовы. Но, все-таки не следует слишком уж доверять компилятору и создавать **final** методы, только, если они действительно небольшие и Вы действительно хотите запретить их изменение при наследовании.

## final и private

Любой **private** метод косвенным образом **final**. Поскольку Вы не можете получить доступ к **private** методу, Вы не можете переопределить его (даже если компилятор не выдаст сообщения об ошибке при переопределении, Вы все равно не сможете переопределить его, Вы просто создадите новый метод). Вы можете добавить спецификатор **final** к **private** методу, но это не добавит ему никаких дополнительных возможностей.

## Final классы

Когда Вы объявляете целый класс **final** (путем добавления в его определение ключевого слова **final**), Вы тем самым заявляете, что не хотите наследовать от этого класса или что бы кто-то другой мог наследовать от него. Другими словами, по некоторым причинам в вашем классе не должны делаться какие-либо изменения, или по причинам безопасности не могут быть созданы подклассы. В другом же случае, причиной сделать класс final может послужить эффективность выполнения кода класса, но здесь нужно быть уверенным, что все, что внутри класса уже оптимизировано как можно максимально.

# Резюме

Оба метода, и наследование и композиция позволяют Вам создать новый тип из уже существующего типа. Обычно, Вы используете **композицию для повторного использования существующих типов как части имплементации нового типа**, а **наследование когда Вам необходимо повторно использовать интерфейс**. В силу того, что дочерний класс имеет интерфейс базового класса, он может быть приведен к базовому типу.

Относитесь с осторожностью к наследованию в объектно-ориентированном программировании, когда Вы начинаете разработку нового проекта используйте лучше композицию, а после того, как ваш код будет доведен до совершенства, измените композицию на наследование, если это конечно необходимо. **Композиция имеет тенденцию к большей гибкости**. Вы так же можете изменять поведение объектов с композицией во время исполнения.

Повторное использование композиции и наследования оказывает огромную помощь для быстрой разработки проектов, Вы обычно хотите изменить иерархию ваших классов, до того, как другие программисты станут работать с вашим проектом и с вашими классами. Вашим преимуществом при этом будет наследование, где каждый класс не большой по размеру (но и не слишком маленький, что бы не потерять функциональность) и выполняет узкую задачу.