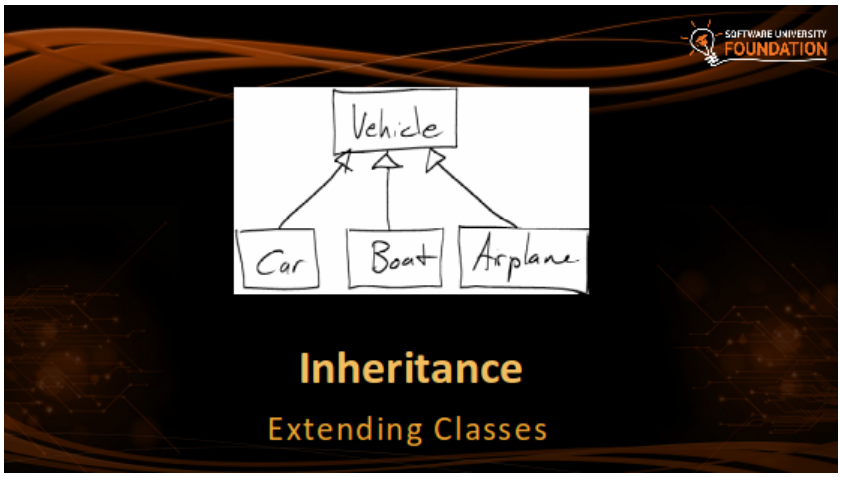
# вторник, 28.02.2017

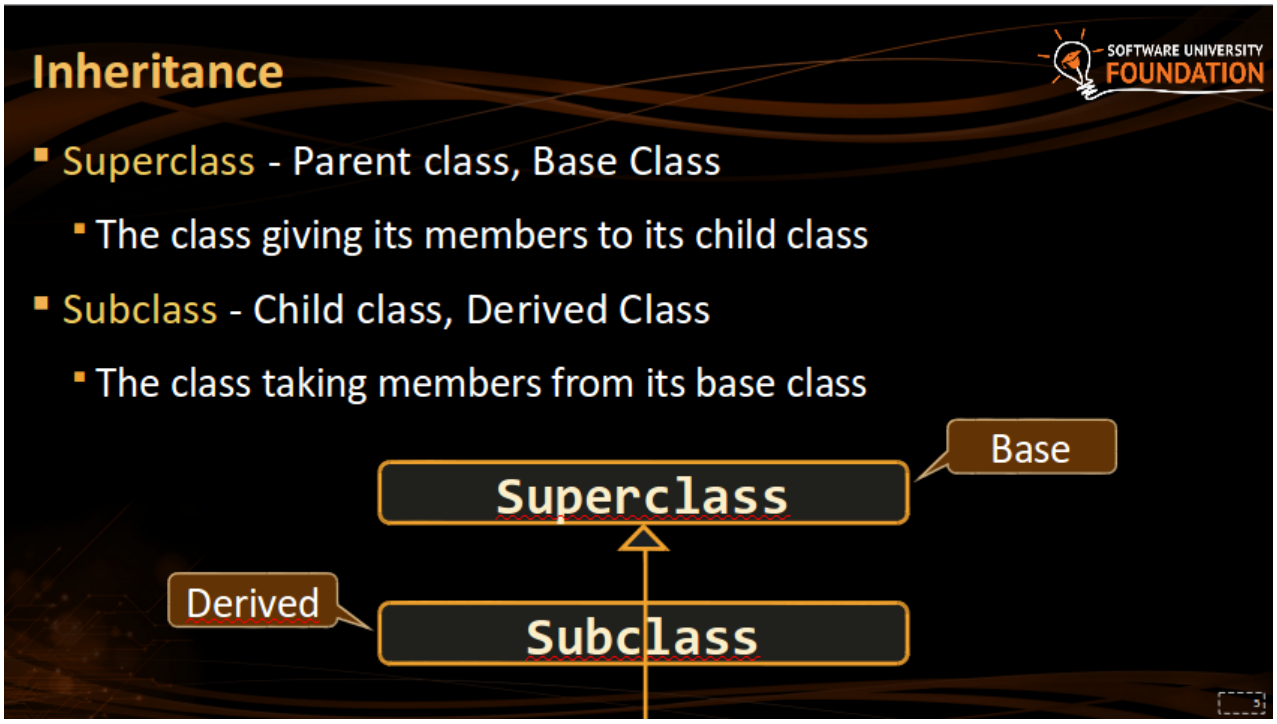
**Inheritance(Наследяване)** Един клас може да наследява друг и много класове могат да наследяват даден клас. Освен това можем да направим така че няколко класа да се наследяват един в друг и да стане верига от наследявания.

**Какво имаме предвид под наследяване ?**




Това означава един клас(примерно класа кола показан на горната снимка) да вземе всички черти на някакъв друг клас, който в случая с горния пример е Vehicle. В примера показан по-горе имам различни класове(Car, Boat, Airplane) и искаме всички те да споделят всички черти на нещо общо между тях. И то е, че всичките представляват превозни средства. И по този начин, ние можем да си изкараме всички общи характеристики между тях в един общ клас, който те като наследят, всеки един от наследниците ще може да се възмолзва от наследените членове на по-горния клас.
**ИДЕЯТА ТУК Е ДА ВЗЕМЕМ КОД, КОЙТО СМЕ ПОЛЗВАЛИ И ДА МОЖЕМ ДА ГО ПРЕИЗПОЛЗВАМЕ В НЯКАКВИ ДРУГИ КЛАСОВЕ, КОИТО СМЕ СИ СЪЗДАЛИ.**

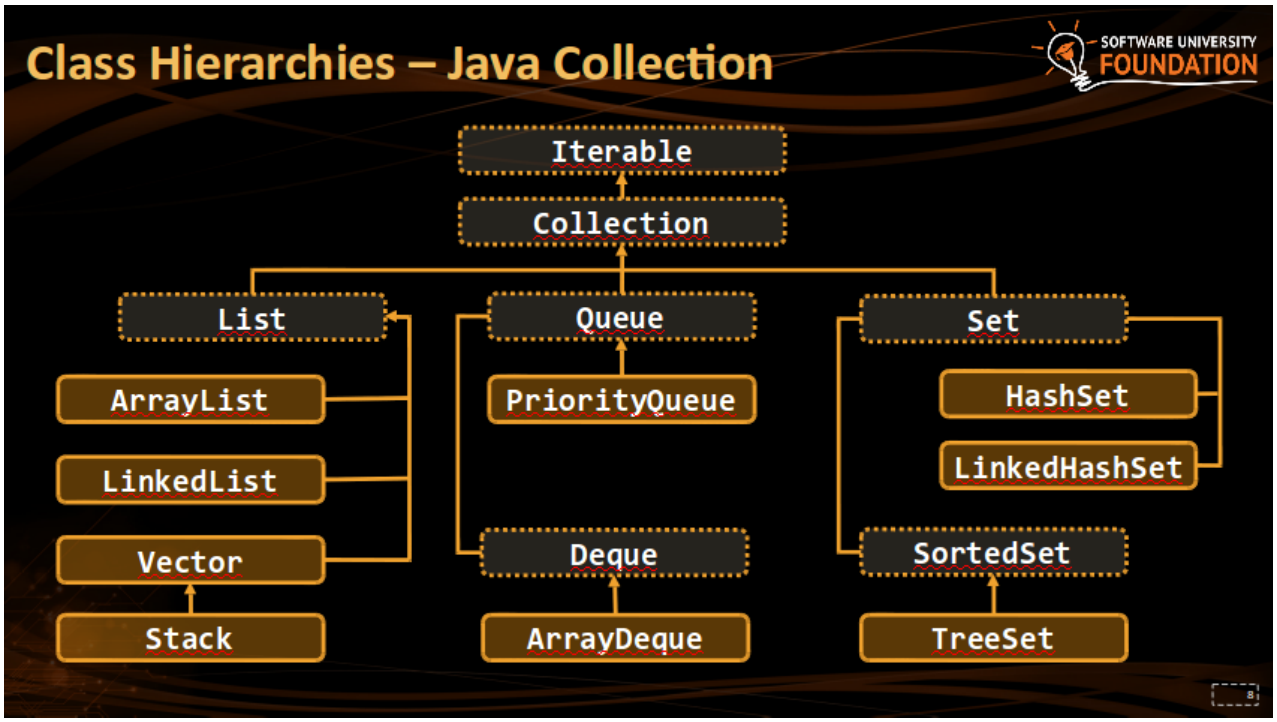
---




По терминологията на Java, всеки един базов(бащин или родителски) клас се нарича

По терминологията на Java, всеки един базов(бащин или родителски) клас се нарича **Superclass** , а всеки един който е наследник - **Subclass** (подклас). Един клас може да бъде **Superclass** и едновременно с това да бъде и **Sublass**(подклас). Примерно ако седи някъде в средата на дадена йерархия, може да бъде бащин клас на някой друг и в същото време наследник на някой друг. **ВАЖНО Е ДА СЕ ОТБЕЛЕЖИ , ЧЕ ЕДИНИЯ КЛАС ПРЕДОСТАВЯ ВСИЧКИ СВОИ ЧЛЕНОВЕ НА НАСЛЕДНИЦИТЕ СИ, А НАСЛЕДНИЦИТЕ ВЗЕМАТ ВСИЧКИ ЧЛЕНОВЕ НА РОДИТЕЛСКИЯ КЛАС.**

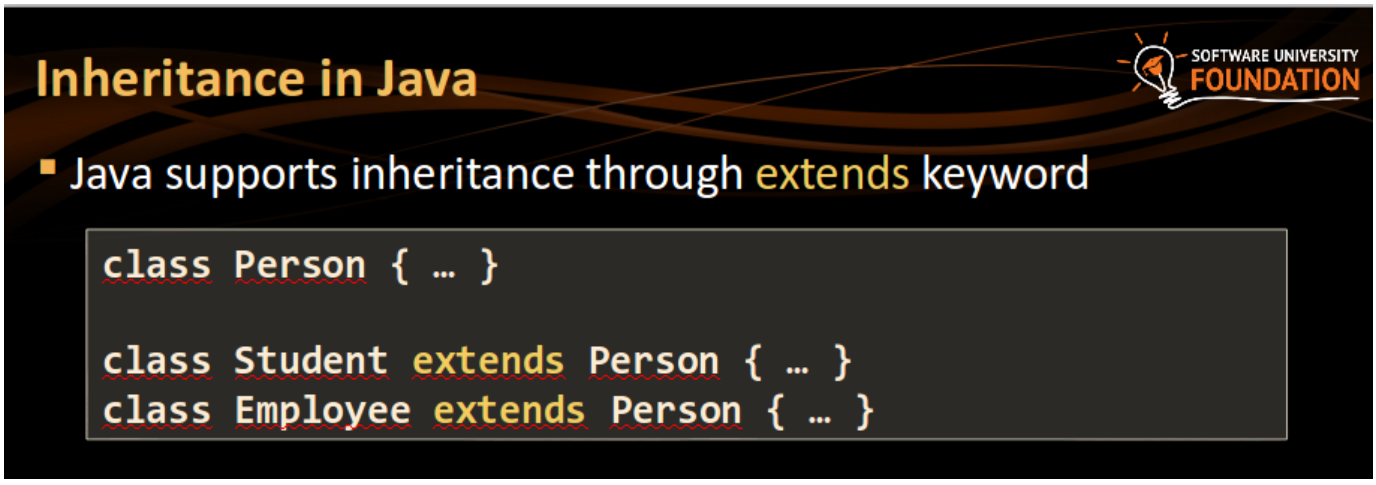
Следващият пример показва йерархия в Java.

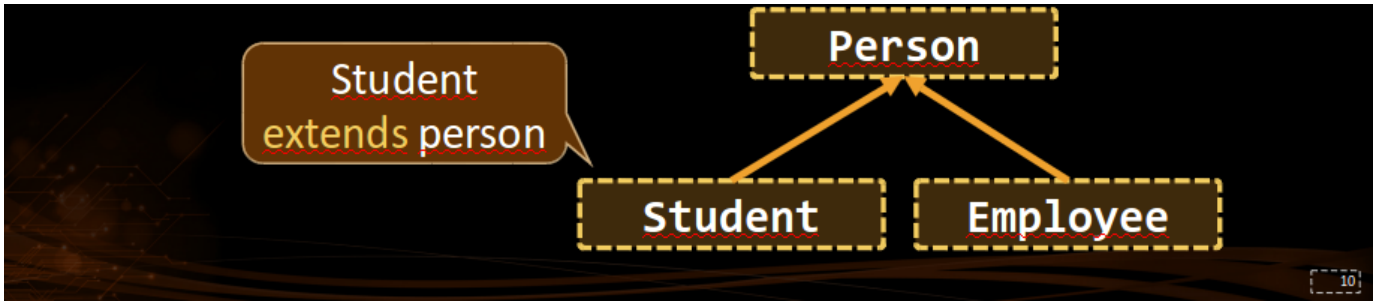



Най-отгоре имаме интерфейсът Iterable, който ни казва че тези неща, които го имплементират, всички те ще могат да бъдат обхождани. Той е общ за всички неща надолу, които го имплементират. По същия начин след това имаме някаква абстракция за колекция, която ни осигурява примерно махане на елементи, добавяне на такива, гледа дали е празна и др. подобни. Надолу имаме други интерфейси, които са имплементации на тези неща. Имаме абстракция за лист, опашка и сет. За всяко едно нещо надолу по йерархията можем да кажем със сигурност , че притежава всички характеристики на всичко, което стои над него. И примерно работейки с един ArrayList, знаем че той е лист, освен това знаем че е и колекция и Iterable.

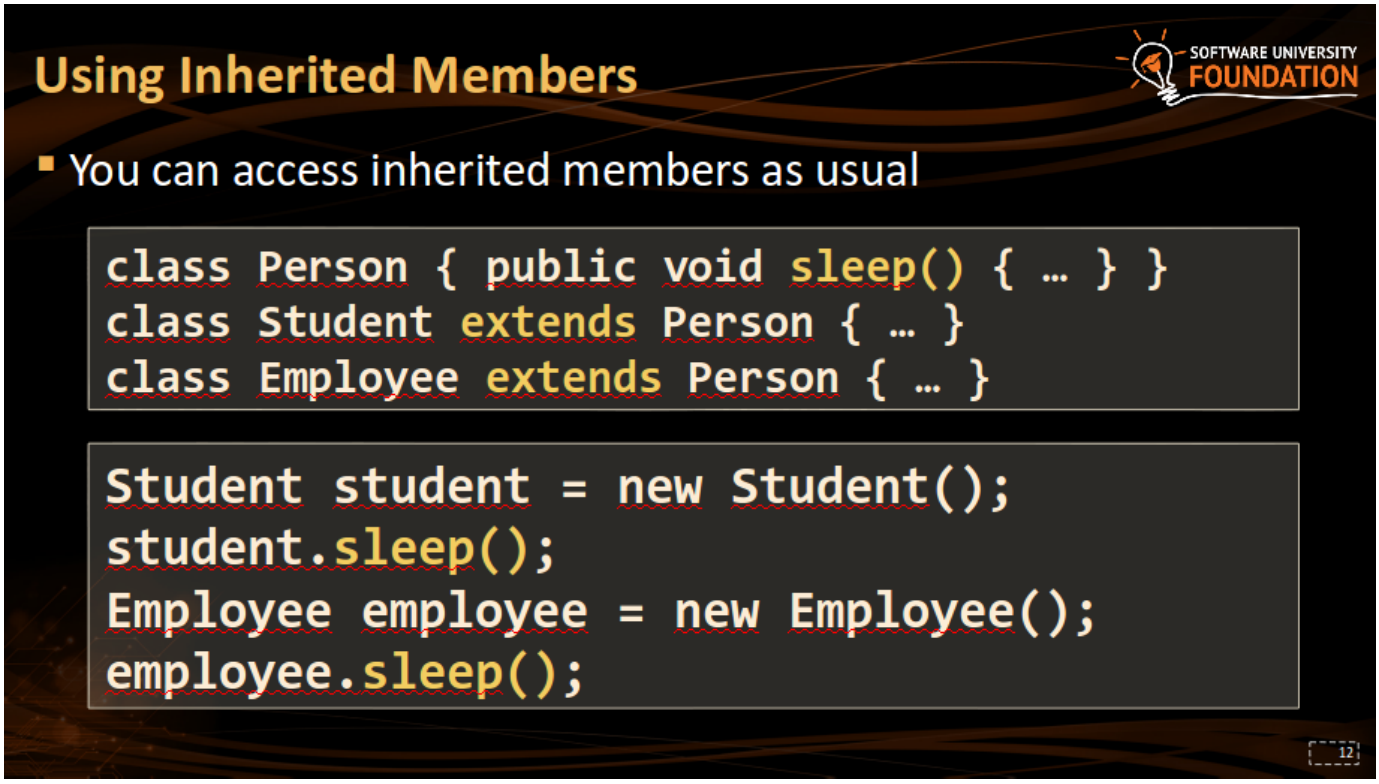
---

**Как можем да си имплементираме някаква йерархична структура ? Тоест как да си имплементираме наследяване в Java ?** Това става с ключовата дума **extends** Да кажем, че имаме един клас Person и искаме от него да бъде наследен от класа Student , и освен това да се наследи и от класа Employee. На следващата картинка е показано как се наследява, чрез ключовата дума **extends**.



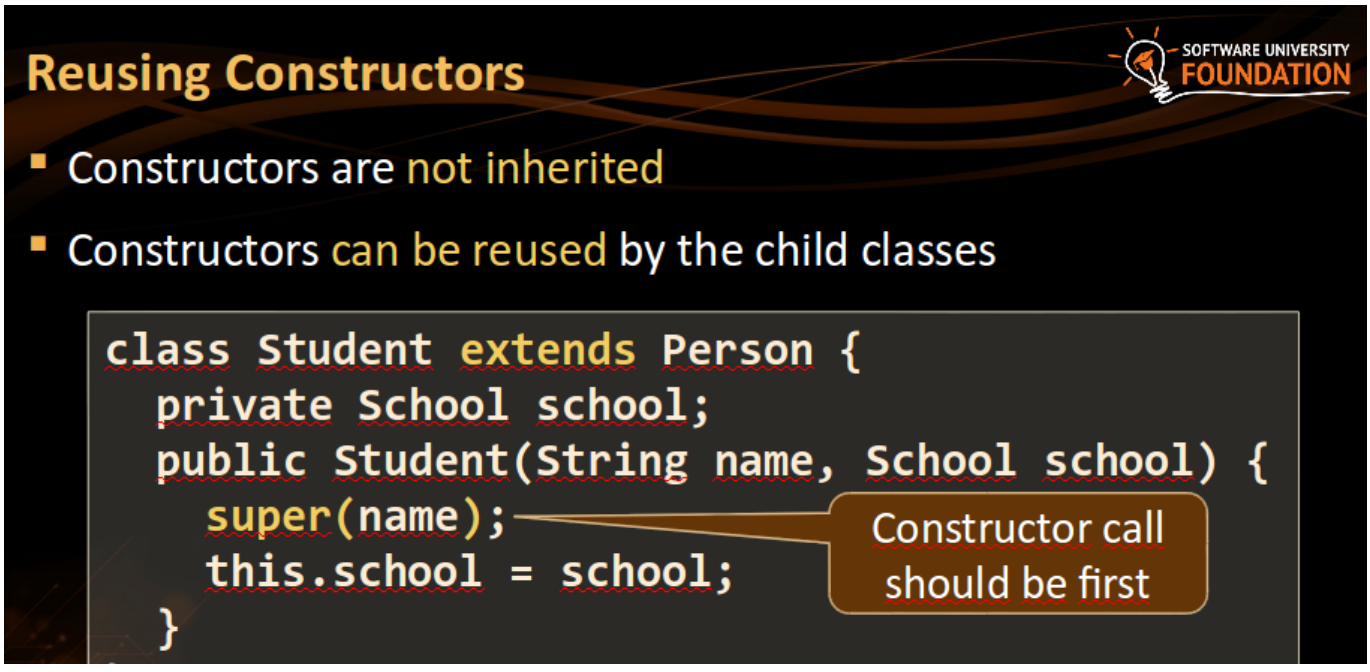



**Как можем да използваме наследените членове ?**




Това става като първо си създадем някакъв обект. Отначало си правим класа Person и си дефинираме някакъв метод в него. Примерно този човек може да спи.(Примерът е показан на горната картинка). След това си правим класа Student, който extend-ва (наследява Person) и след това още някакъв клас - Employee, който също наследява класа. И сега как можем да ползваме този наследен метод - sleep(), който сме наследили от Person класа ? Просто като си създадем една инстанция от класа Student и му извикаме методът sleep(); Той просто си го има, защото е наследен.

**Как да преизползваме конструкторите ?**



Понеже конструкторите не се смятат за членове на класа, то те **НЕ СЕ НАСЛЕДЯВАТ**. **Ако конструкторът на базовия клас е видим за наследения клас, то последния може да го вика** и да използва(показано е на горната картинка)

**По какъв начин се създават новите обекти, тоест като създадем нов обект от класа Student по какъв начин ще се пази той ? Дали ще бъде нов обект от клас, който комбинира Student и класа Person или ще е някакъв хибриден обект ?**

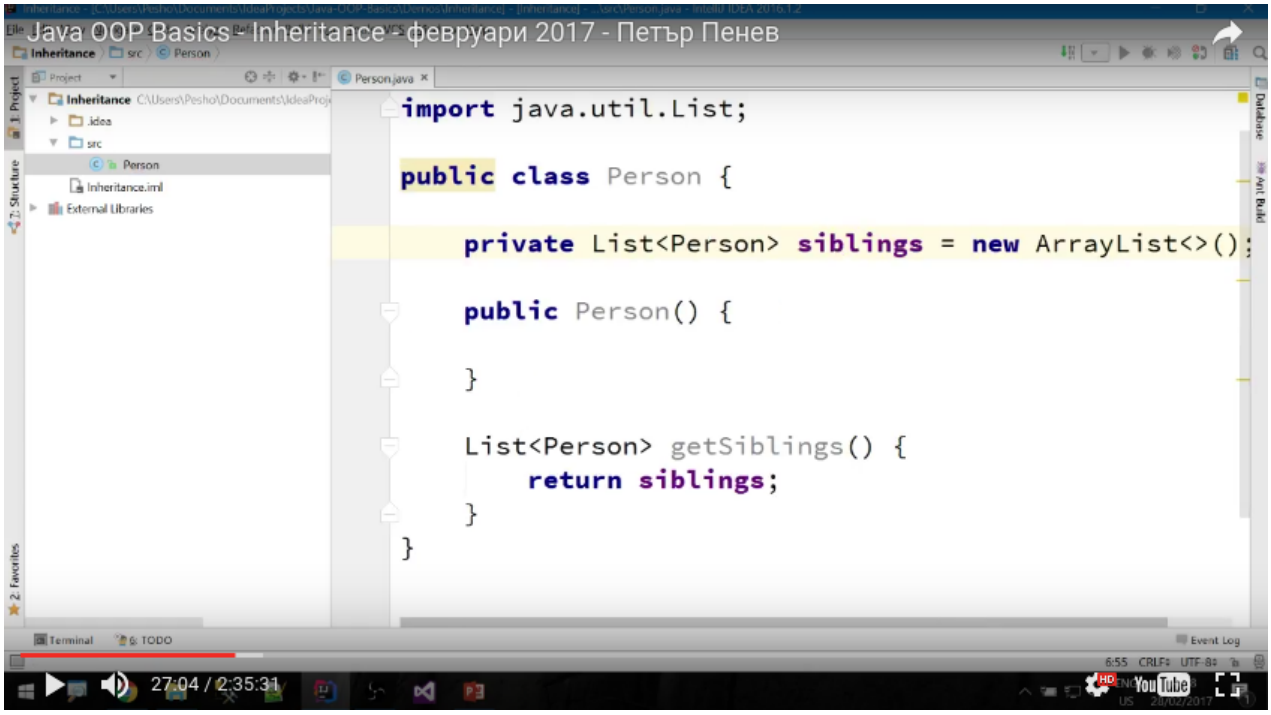
Студентът като един клас , като обект ще се създаде с всички неща, които са присъщи на него, обаче в себе си ще съдържа един допълнителен обект, който е от класа Person. И по тази причина, за да можем да създадем един обект от класа Student, ние първо трябва да сме създали обект от класа Person. **И ПОРАДИ ТАЗИ ПРИЧИНА ВИКАНЕТО НА СУПЕР КОНСТРУКТОР В ТЕКУЩИЯ ВИНАГИ ТРЯБВА ДА СТАВА НА ПЪРВИЯ РЕД**. Компилаторът няма да ни позволи super(name) да бъде на различен ред освен първия. **Просто защото това поле this.school = school няма къде да отиде. Обекта от клас Student все още не е създаден**. Затова първо трябва да се създаде Person-а и след това се създава обектът Student и вече можем да му запишем всичко, което му трябва.

---

**Кога е хубаво да си инициализираме разни обекти ?**

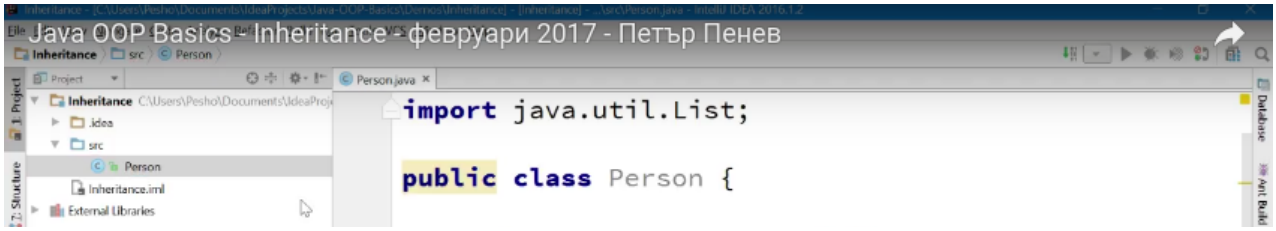
Имаме няколко алтернативи(те ще бъдат показани на следващото изображение)... Да кажем , че имаме клас Person и той има примерно някакъв List от Person-и (List<Person>). И сега оттук имаме няколко варианта:

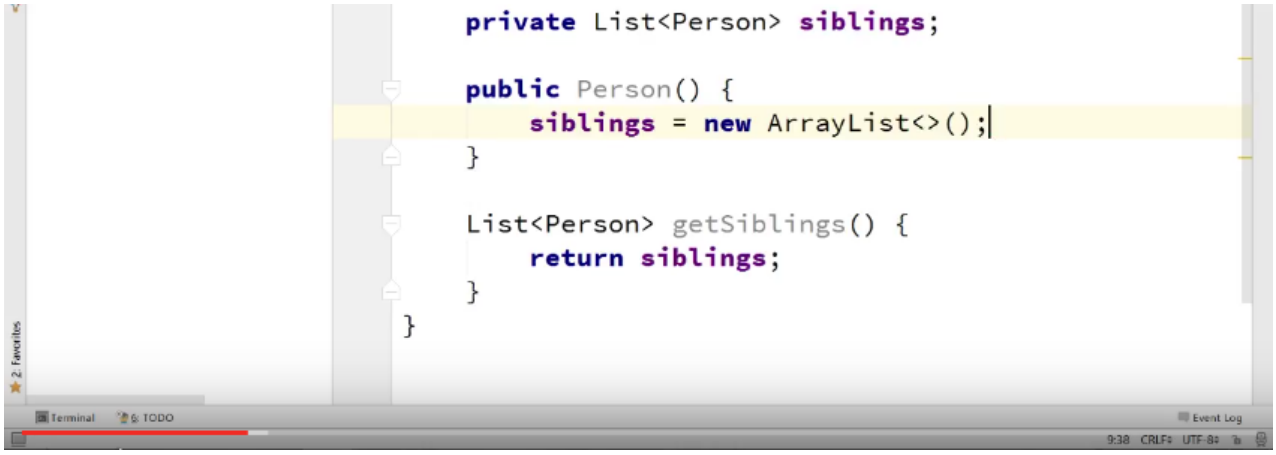
- **Да си инициализираме този лист директно в блока с променливи**.




  При този вариант, в момента в който се създава обекта от този клас, още преди да се изпълни каквото и да е в конструктора, първо ще се инициализират всички променливи, които сме декларирали над него. За тези променливи така или иначе трябва да се задели памет без значение от какъв тип са те. Ако са примитивни така или иначе ще се задели памет за тях в стека. Обаче ако са обекти в стека ще се задели място за техните референции, а в heap-а ще се задели памет и за новия обект. И при този вариант всички тези полета трябва да са инициализирани преди да се извика конструктора.

- **Другият вариант е да се инициализира в конструктора**.

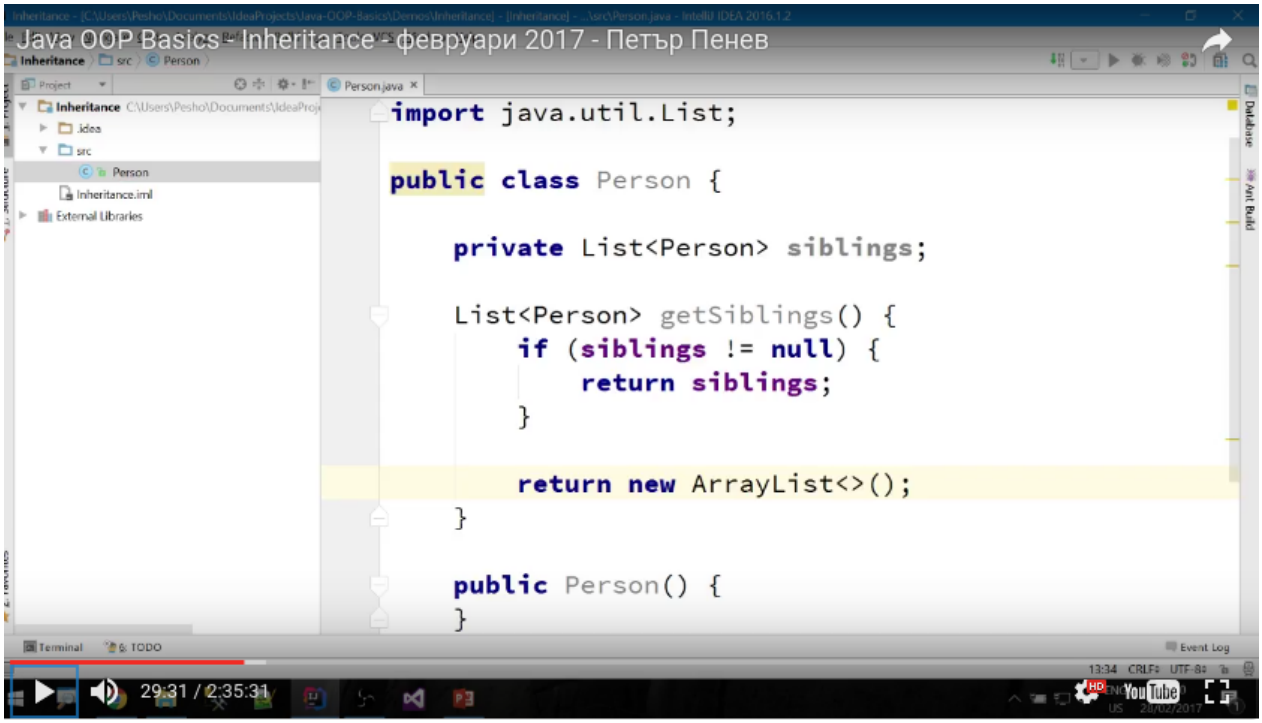





**При този вариант какво ще стане, когато тръгнем да създаваме обект от този клас ?**

Преди да се изпълни конструктора, първо ще бъде заделена памет в стека за всички променливи, които са декларирани над конструктора. Обаче обектите ще се създадат, чак в конструктора. Тоест това е една крачка по-напред във времето.

- **Третият вариант е в метод**. - **Нарича се още мързелива инициализация (Lazy initialization)**.



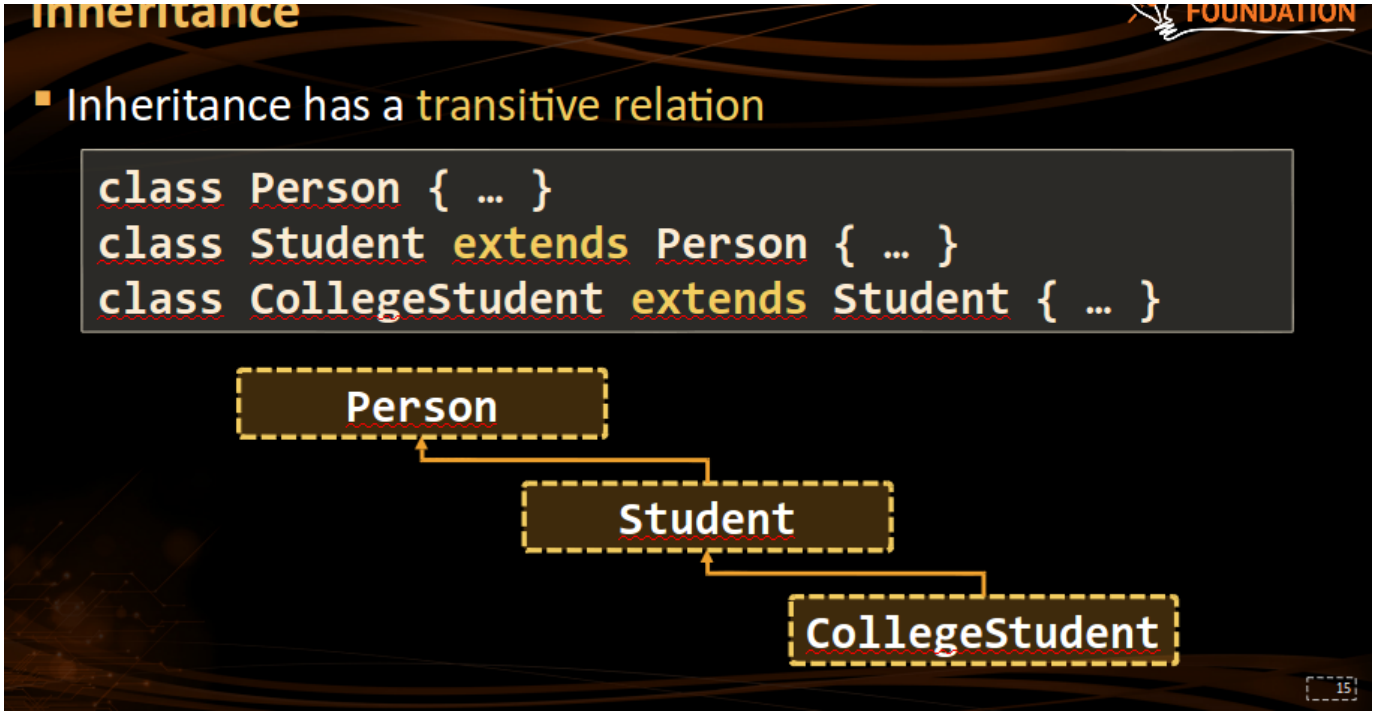

При него инициализацията не е нито в блока с променливите, нито в конструктора, а в момента в който ни потрябва. Както е показано на горната снимка в този случай листът ще се създаде в последния възможен момент.

**Кой от всички тези начини да използваме ?**

Зависи от ситуацията. Ако примерно имаме някакви много тежки обекти, които съдържат много информация в тях, най-добрият начин е да използваме мързеливата инициализация. (Lazy initialization). При примитивните типове данни ние нямам избор, защото щом са декларирани в този блок, за тях така или иначе ще се отдели памет. А за обикновените обекти е добре възможно най-късно, **НО КАТО ЦЯЛО Е ДОБРА ПРАКТИКА ДА СЕ ИНИЦИАЛИЗИРАТ В КОНСТРУКТОРА.**

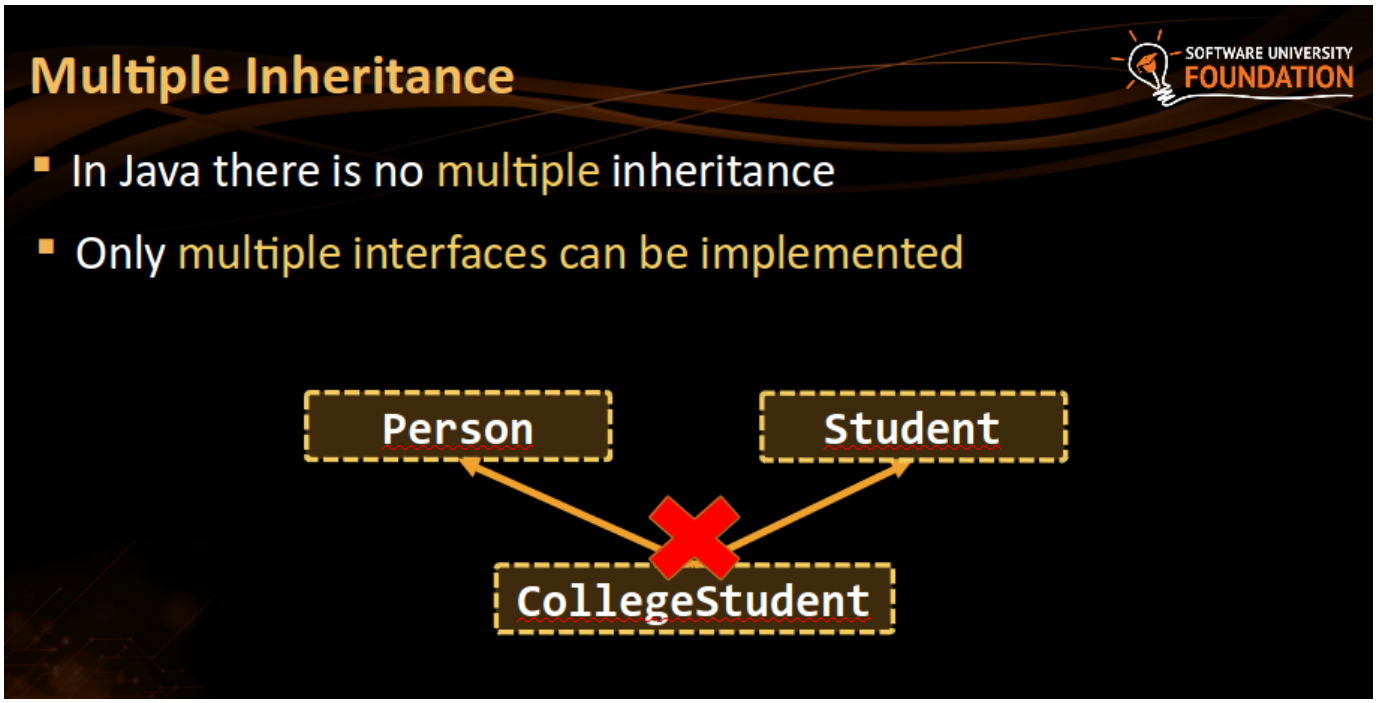
---

**ЕДНО ОТ ДРУГИТЕ МНОГО ВАЖНИ НЕЩА ЗА НАСЛЕДЯВАНЕТО Е, ЧЕ ТО ИМА ТРАНЗИТИВНА ВРЪЗКА. ТОЕСТ АКО ЕДИН КЛАС НАСЛЕДЯВА ДРУГ, ВТОРИЯТ НАСЛЕДЯВА ТРЕТИ, ТО ТРЕТИЯТ ЩЕ НАСЛЕДЯВА ВСИЧКО ОТ ПЪРВИЯТ**. Пример в следващия слайд.




В примера от горната картинка , класа Student наследява Person, а CollegeStudent наследява Student. Следователно щом CollegeStudent наследява Student, то следователно наследява и Person.

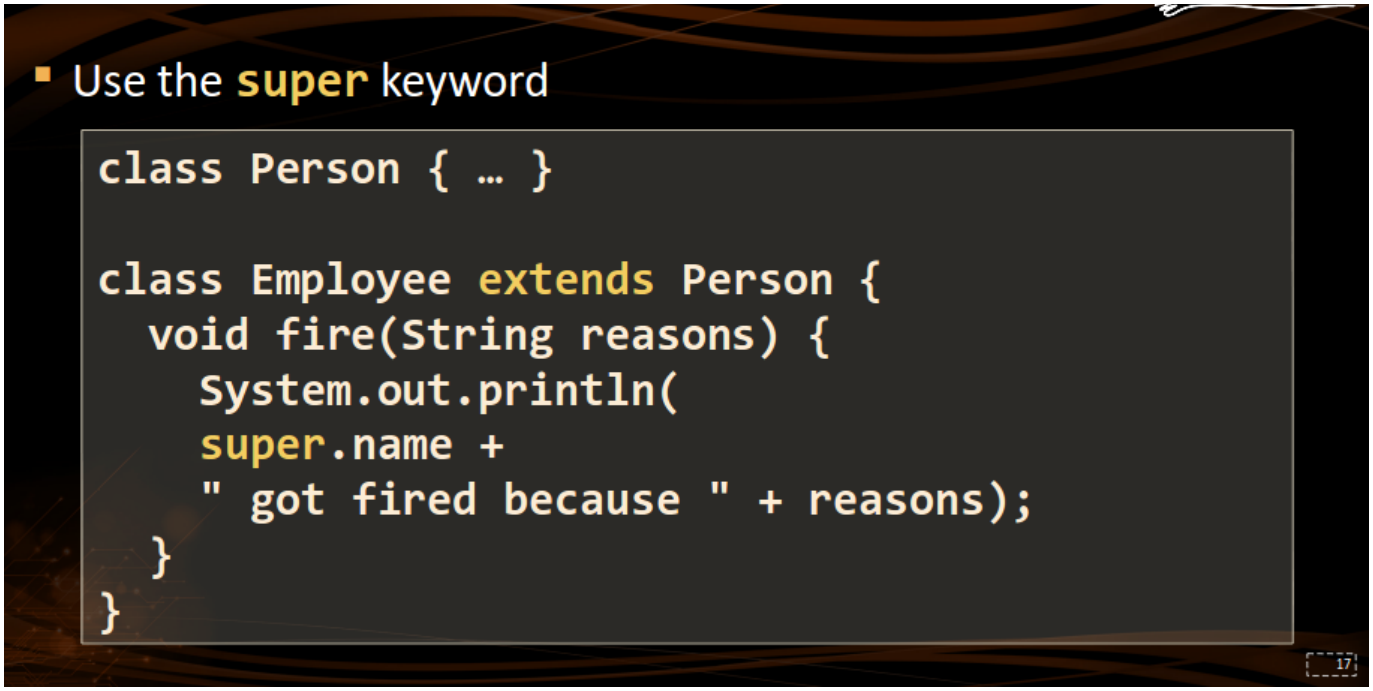
**Multiply Inherence(Множествено наследяване)**




**ТРЯБВА ДА СЕ ЗНАЕ, ЧЕ В JAVA НЯМА ТАКОВА НЕЩО КАТО МНОЖЕСТВЕНО НАСЛЕДЯВАНЕ**. (Показано е в примера на горната илюстрация). **В JAVA ИМА МЕХАНИЗЪМ ДА СЕ СЪЗДАДЕ НЕЩО ПОДОБНО, НО НЕ Е СЪЩОТО И СТАВА ЧРЕЗ МНОЖЕСТВЕНО НАСЛЕДЯВАНЕ НА ИНТЕРФЕЙСИ.** Примерно в горната илюстрация да кажем , че Person и Student не са класове , а интефейси. В този случай класът StudentCollege може да ги имплементира и двата едновременно. Интерфейс можем да си го представим като нещо подобно на договор, който един клас се съгласява да изпълни.

---

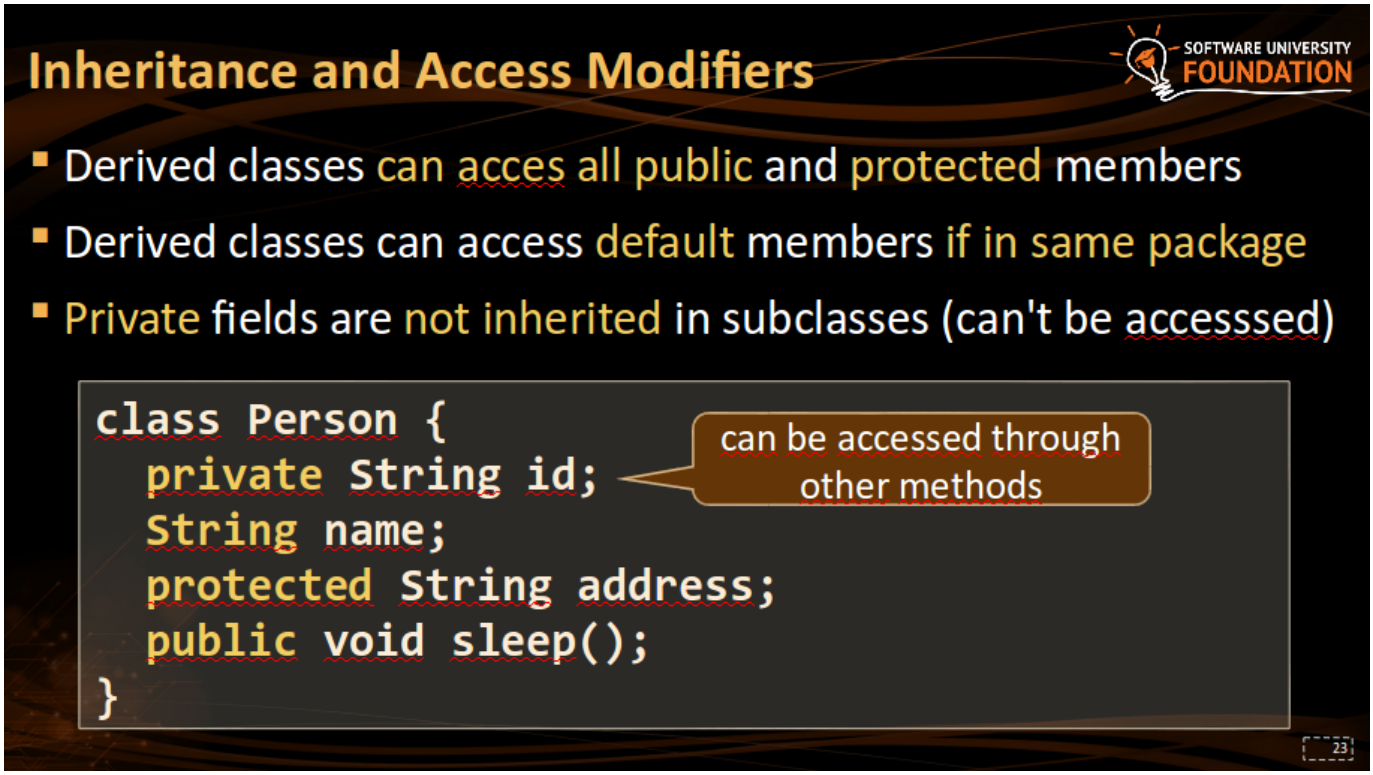
**Access to Base Class Members(Достъп до членовете от базовия клас)**




Извикването на членове(полета и методи) от базовия клас става посредством ключовата дума super. Примерно на горната картинка имаме клас Person, а също така и друг клас Employee, който наследява Person. И сега примерно класът Employee добавя един метод наречен fire(уволнение в този контекст) и приема като параметър String reason. И примерно сега искаме на конзолата да изпишем името на човека и че е уволнен. Това става като просто напишем super.name и след него всички останало, което искаме.
**ТОЕСТ ЗА ДА ДОСТЪПИМ ИМЕТО НА ЧОВЕКА, КОЕТО Е В БАЗОВИЯ КЛАС, ТРЯБВА ДА ИЗПОЛЗВАМЕ СУПЕР.**

---

**Reusing Classes(Преизползване на код)**

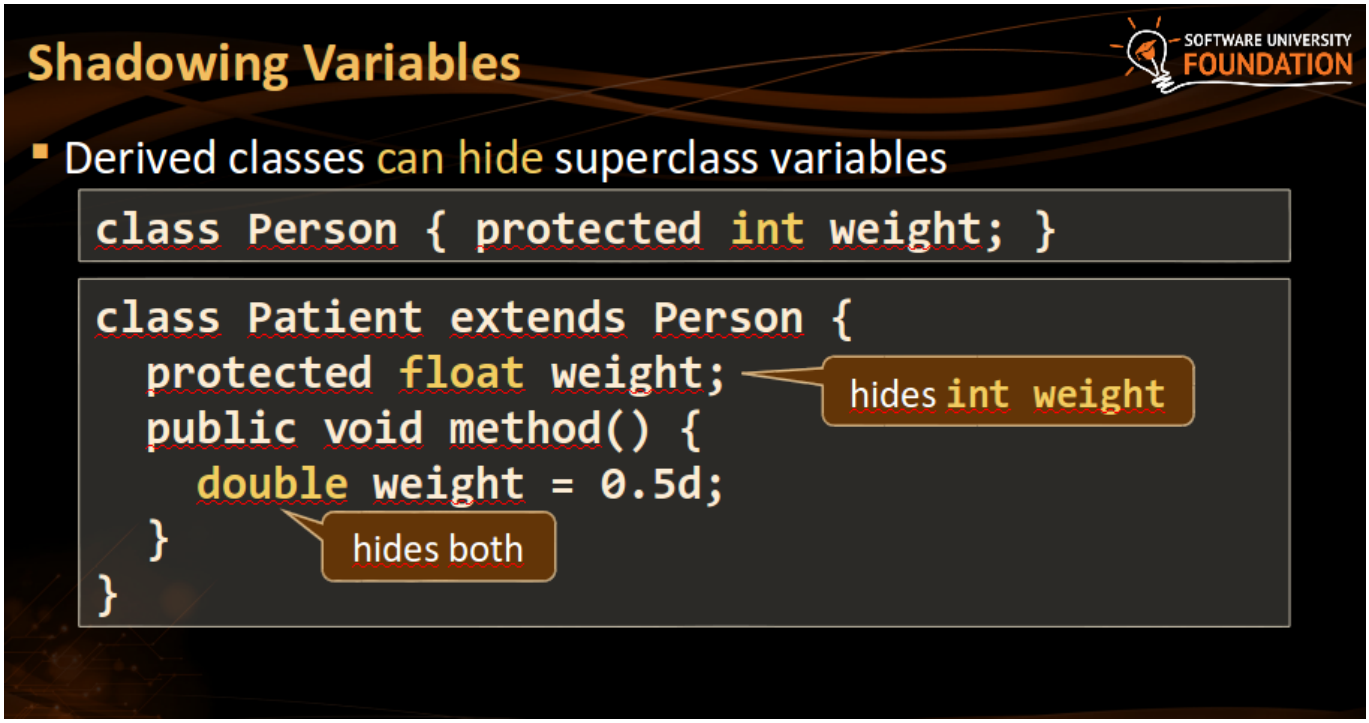



За самите наследници има значение базовите членове как са дефинирани. Ясно е, че ако са публични, то те са достъпни за всички и от навсякъде. Обаче какво се случва ако имаме дадено поле, което е дефинирано като private? (На примера от горната картинка такова е полето private String Id). По принцип родителския клас не е длъжен да предоставя всички свои полета на своите наследници. Private полетата на бащиния клас не могат да се достъпят от наследниците директно. В случая с примера отгоре, никой от наследниците няма да може да вижда полето id. Наследниците все пак могат да променят тези полета ,но не пряко, а косвено. Не може да се каже на дадено private поле просто super.име на полето и да бъркинат по него. Те могат единствено ако полето си

има getter или setter. С modifier-а , който е по default(по подразбиране) , това е полето String name в примера на горното изображение, с него дори един клас да наследява друг, то той няма да може да достъпва членовете с този modifier, освен ако не са в един и същ пакет(package). Член от класа, който е деклариран с modifier protected ще бъде достъпен за всички наследници в този клас, плюс всички други класове, които се намират в този package. Трябва да се използва единствено, когато искаме да дадем достъп на наследниците. В другия случай може да се използва default-ния modifier.

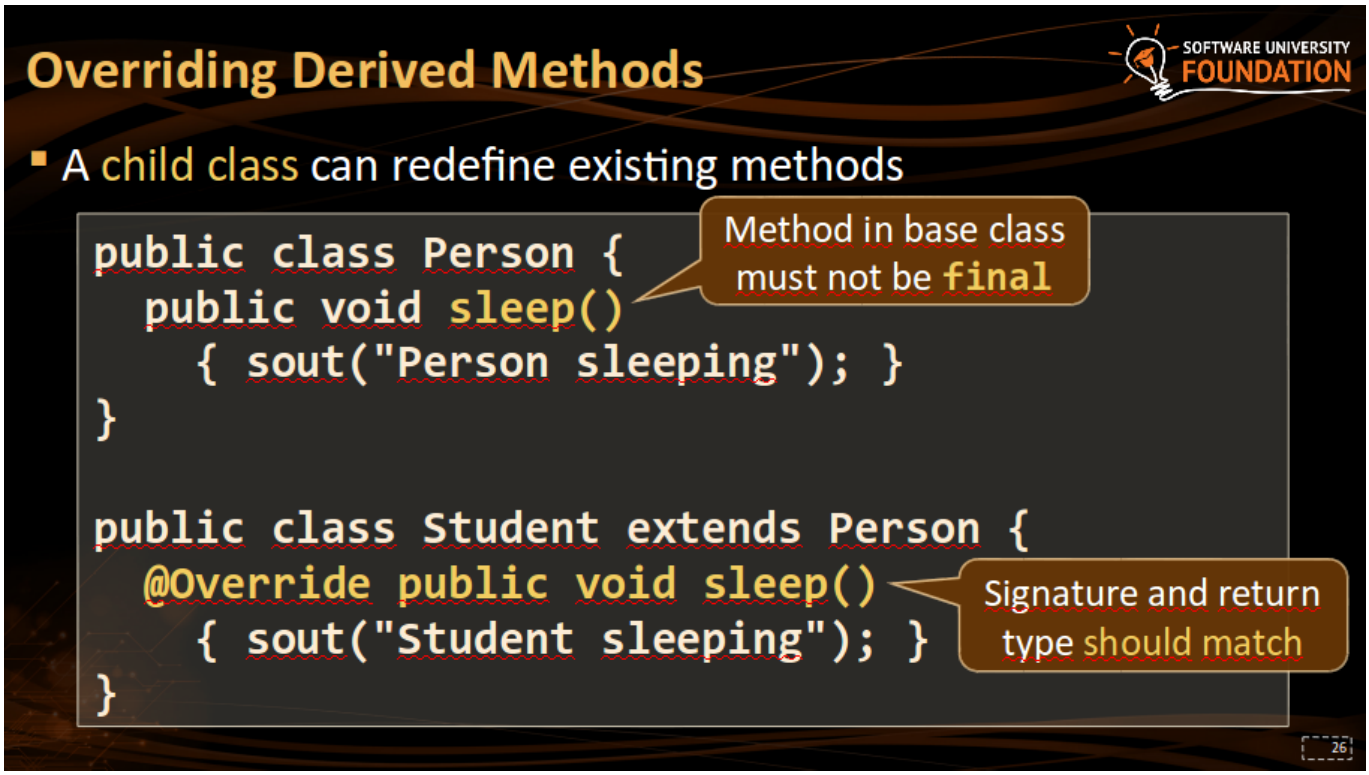
---

**Shadowing Variables(Скриване на променливи)**



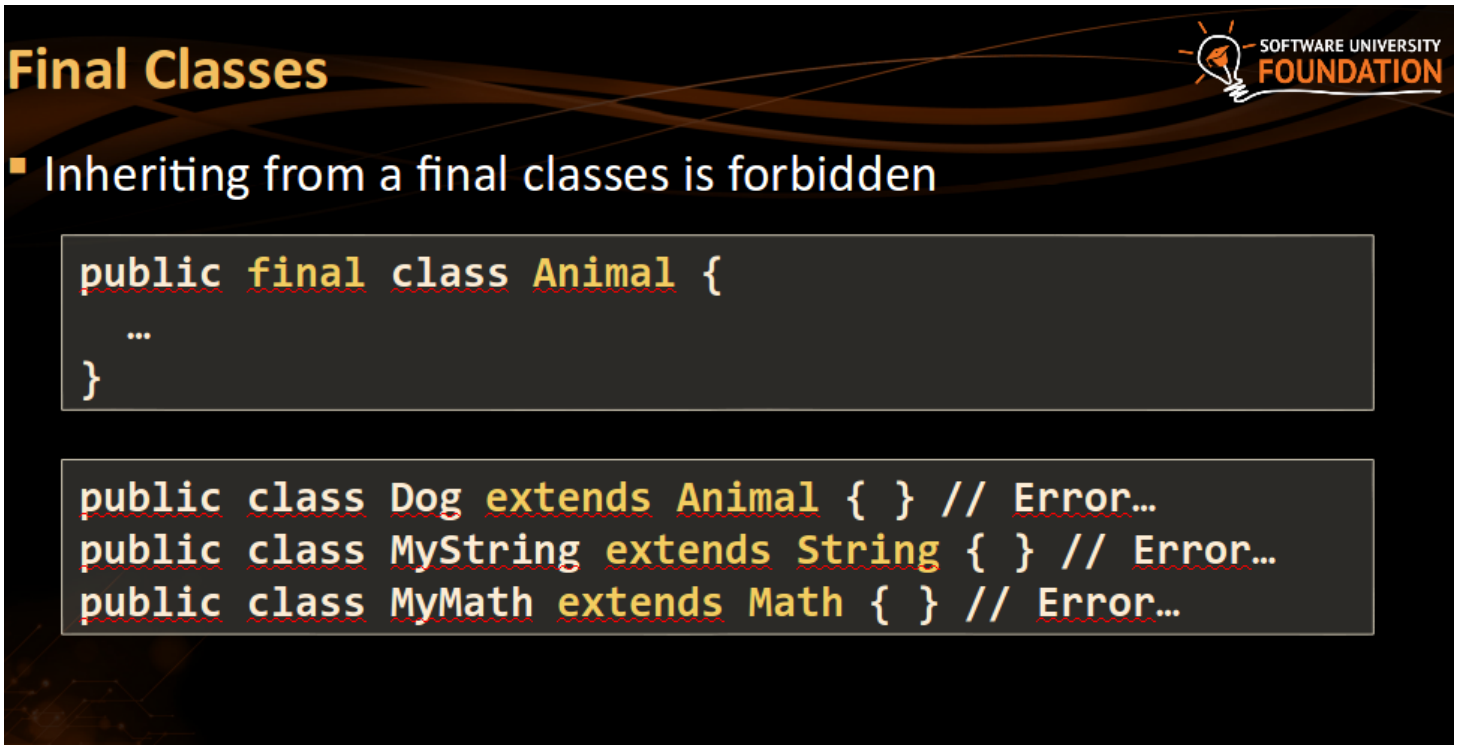

Когато един клас наследява друг, то той може да декларира променлива със същото име като такава в бащиния. И така скрива родителската променлива. Има още едно ниво на скриване, което в local scope(обхвата) на даден метод , може да имаме още една променлива със същото име, която пък ще скрие другите две. (Показано е на горния слайд). Малко е като последния да затвори вратата... :D

**Overriting Derrive Methods(Скриване на методи)**




По същия начин, по който скриваме променливите, можем да скриваме и методи. Това

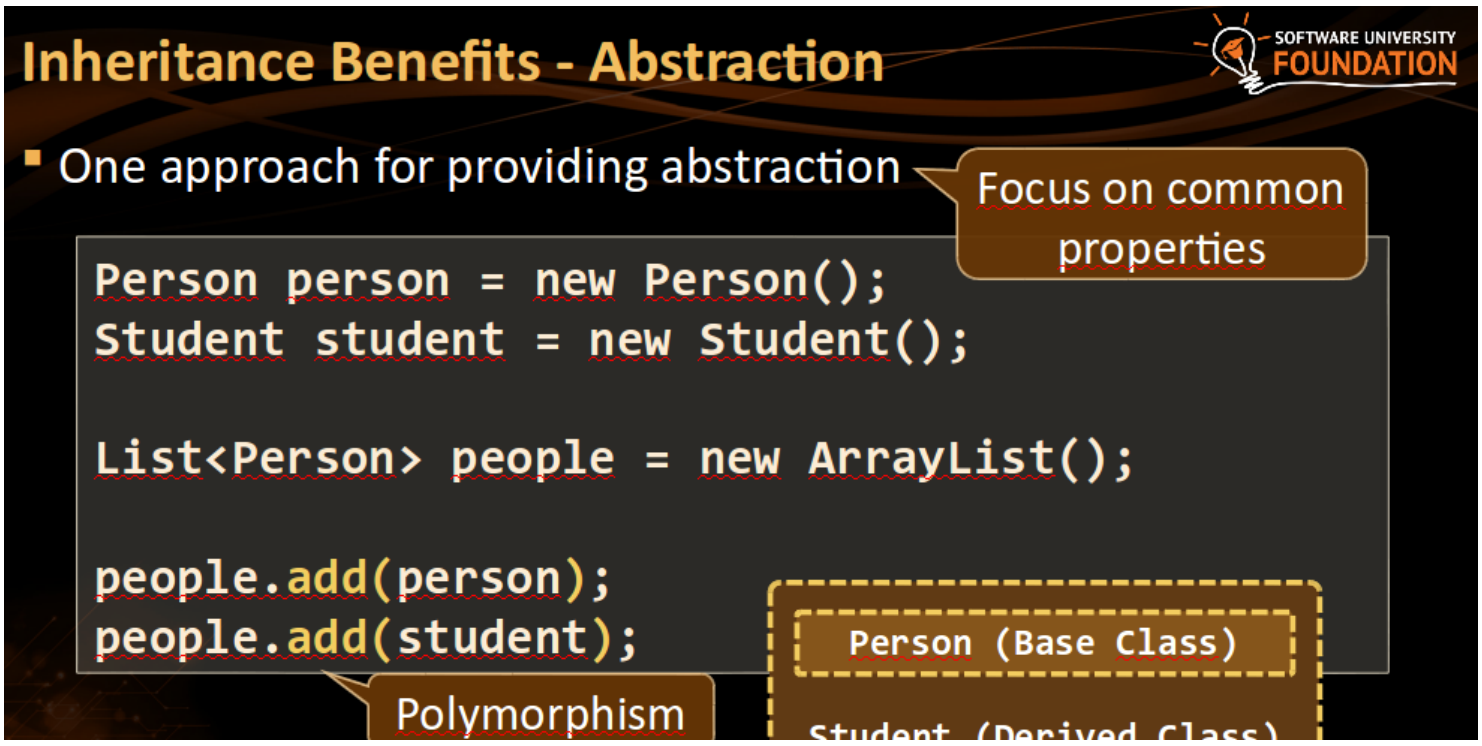
се нарича overrite-ване (презаписване) на методите. Тоест един клас-наследник може да overrite-не същият метод, който се намира на родителския клас. Тоест може да хване методът на родителят си и да подмени съдържанието му. Можем да говорим за чисто наследяване тогава, когато два класа са абсолютно едни и същи, тоест имат същите членове, но просто наследникът променя поведението на методите на своя родител. Това е най-добрият и чист вид наследяване, който можем да имаме.
За да може overrite-ването да е правилно трябва и двата метода да имат еднакви return type и сигнатура(име на метода плюс нещата, които се подават като аргументи). Анотацията @Override , когато я сложим ако сигнатурата или return type-а не отговарят IDE-то ще го подчертае в червено и няма да се компилира, тоест ще ни спаси от някакъв бъг, който трудно бихме могли да хванем. За да можем да презаписваме методи обаче трябва да бъде спазено още едно условие. Те да не бъдат декларирани като final в родителския клас както е показано в следващата илюстрация.




Когато презаписваме методи, изпълнението им има обратна последователност, тоест от най-конкретния клас, към най-абстракния(най-горния базов родителски клас). Работи по този начин, защото иначе overrite-ването нямаше да има смисъл.

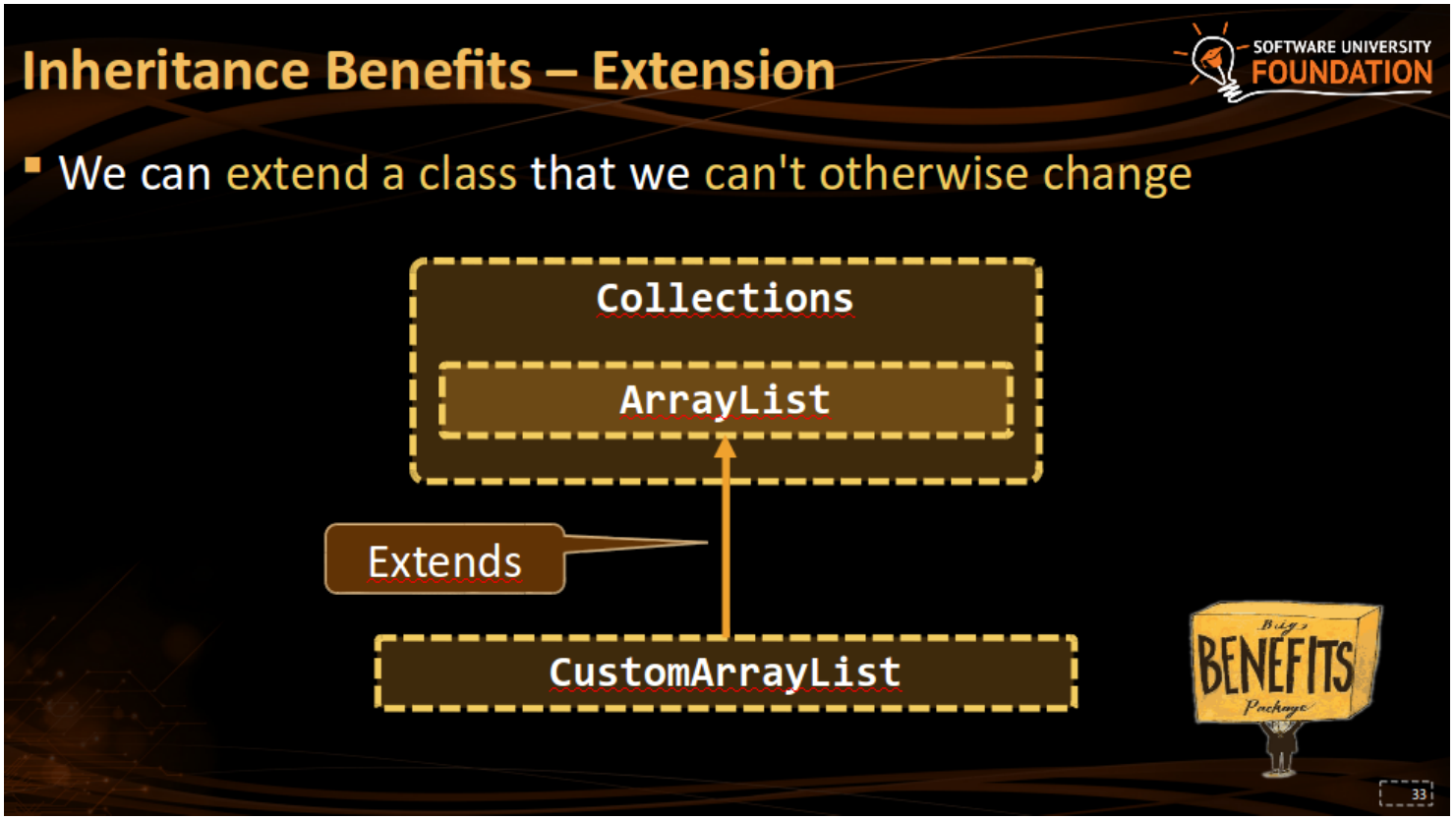
---

**Inheritance Benefits(Ползи от наследяването)**



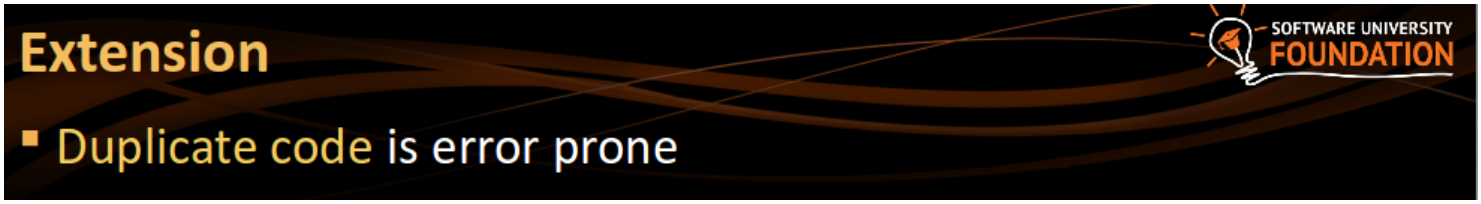

Една от ползите освен че ни дава достъп до всички тези неща по-горе(полиморфизъм, който ще учим по-подробно следващата лекция) , ни дава добра възможност и да си правим абстракция* на своите инстанции на класове. Абстракцията ни позволява на какво ниво на абстракция да работим с два различни обекта, които по принцип са от различни типове, но са свързани йеархично. Примерно в горната картинка, имаме един човек, имаме и един студент и можем да си направим един лист от хора и в него да слагаме едновременно и хора, и студенти, защото знаем, че последните са хора. Това е полиморфизъм. Добавяйки студента в листа от хора и след това изваждайки го оттам, ние ще можем да работим с него като с клас човек, защото няма да знаем , че е студент. Можем да го кастнем към клас студент, но това не е добра практика.

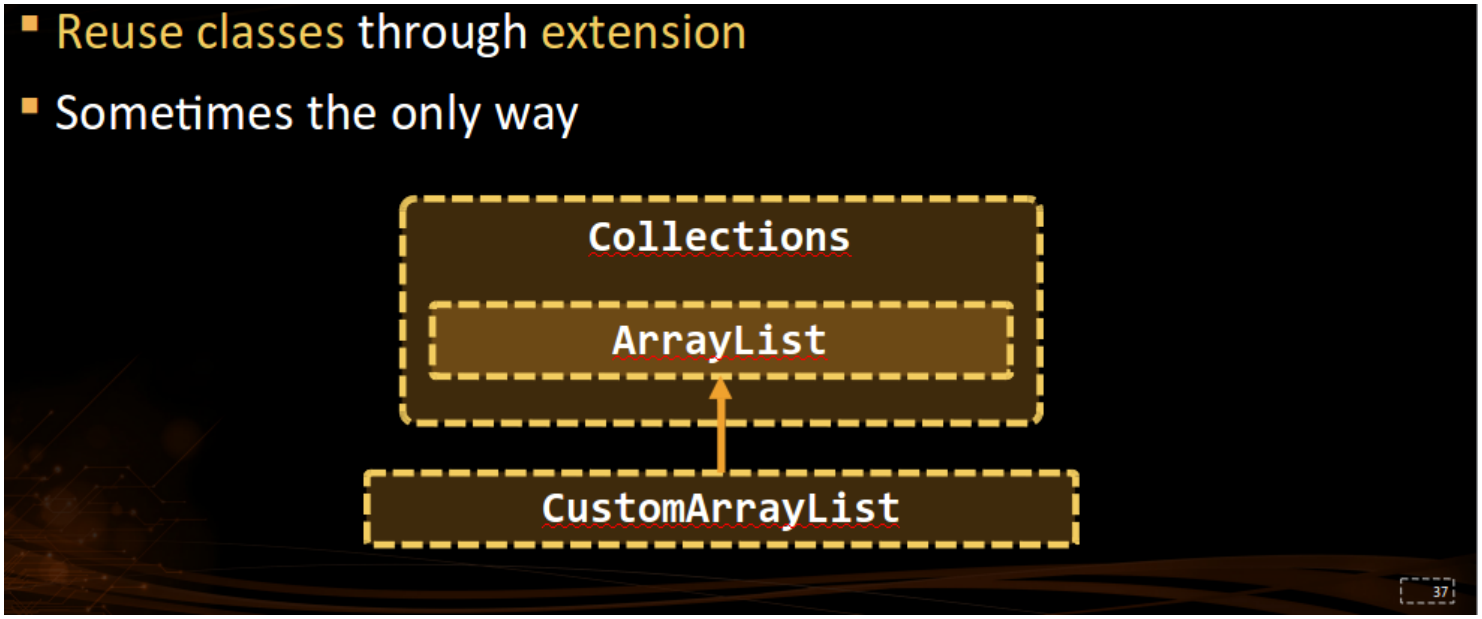
***Абстракция** - Примерно не казваме , че човекът е Пешо, а казваме че е човек. Обръщаме се към точно този обект(ако е в контекста на програмирането) с неговата абстракция, тоест с нещо което стои по-общо от него. Друг пример за абстракция е ако имаме един квартал примерно. В него имаме много къщи, много улици, много дървета и т.н. Обаче той е квартал. **ОБЕДИНЯВАМЕ НЕЩО С ОБЩИТЕ МУ ХАРАКТЕРИСТИКИ**. В текущия пример съвкупност от дървета, къщи и улици и хора на едно място е един квартал. И когато в ежедневието си говорим с някого и вместо да му кажем примерно - "Вчера ходих до тези 60 улици, къщи и хора, които са събрани на едно място" , ние му казваме - "Ходих до онзи квартал с това име".




**Другата голяма полза, която ни дава наследяването е преизползване, разширяване на класовете.** Да кажем , че имаме някаква колекция. Примерно ArrayList. И искаме да и добавим някаква функционалност. Например от този ArrayList винаги да теглим случайни елементи. И затова просто си extend-ваме ArrayList-а, което ще рече че наследникът е клас, който ще има всичката функционалност на ArrayList, тоест ще можем да слагаме вътре обекти, да ги махаме, да проверяваме дали не е празен листа и др. Обаче освен всичко това, ние можем да си добавим някакъв си свой метод.

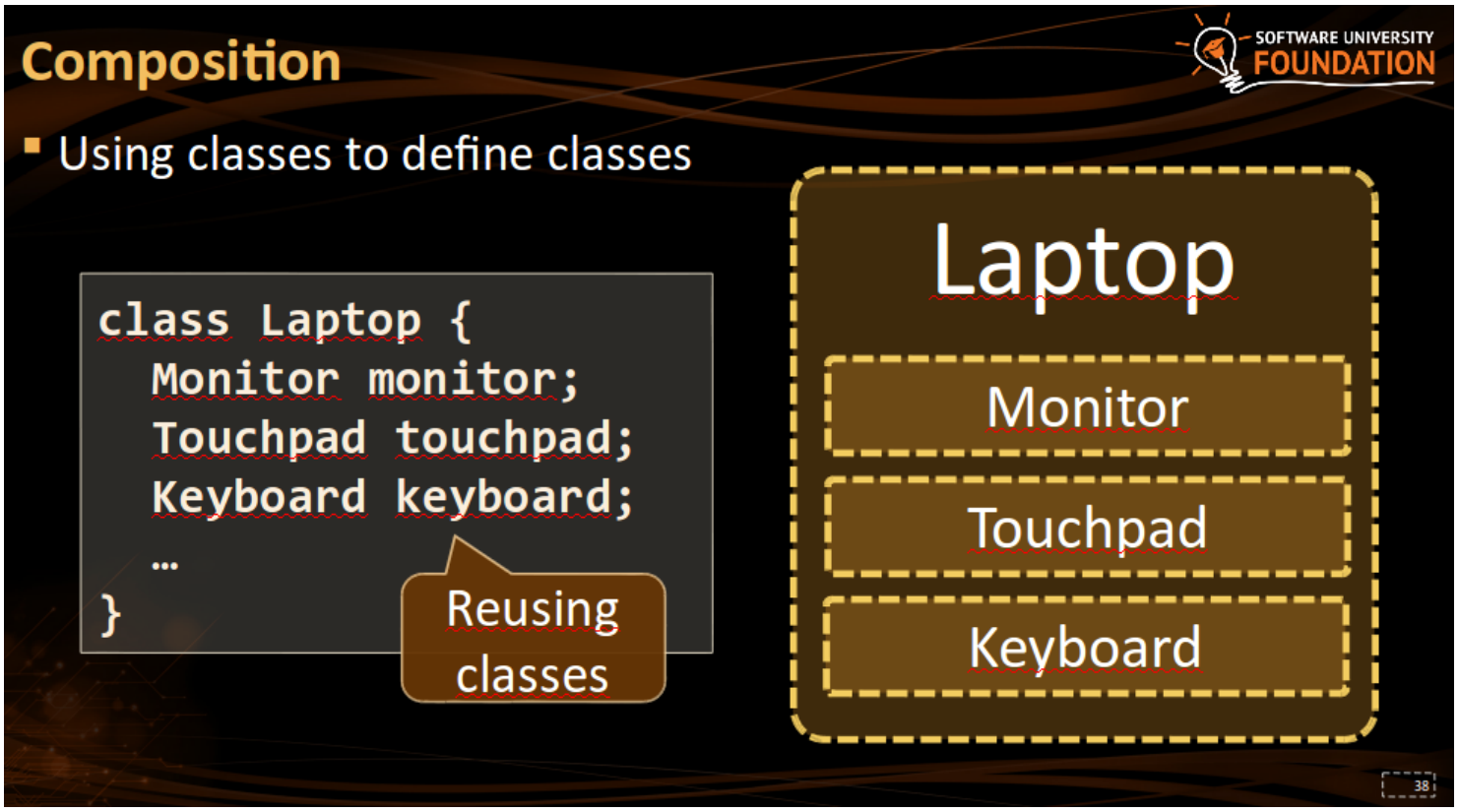
**Types of Class Reuse(Видове преизползване на код)**
**Extension, Composition, Delegation.**





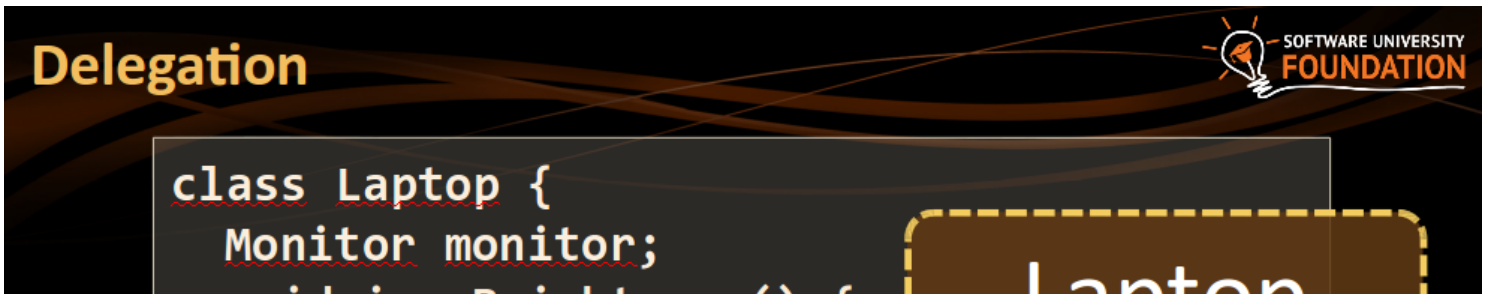

**Когато преизползваме код сме сигурни че един отрязък от код присъства само на едно място. Тоест ако трябва да правим промени не трябва да обикаляме навсякъде из много класове , много методи и тн.**

**Compositon**(Композиция)



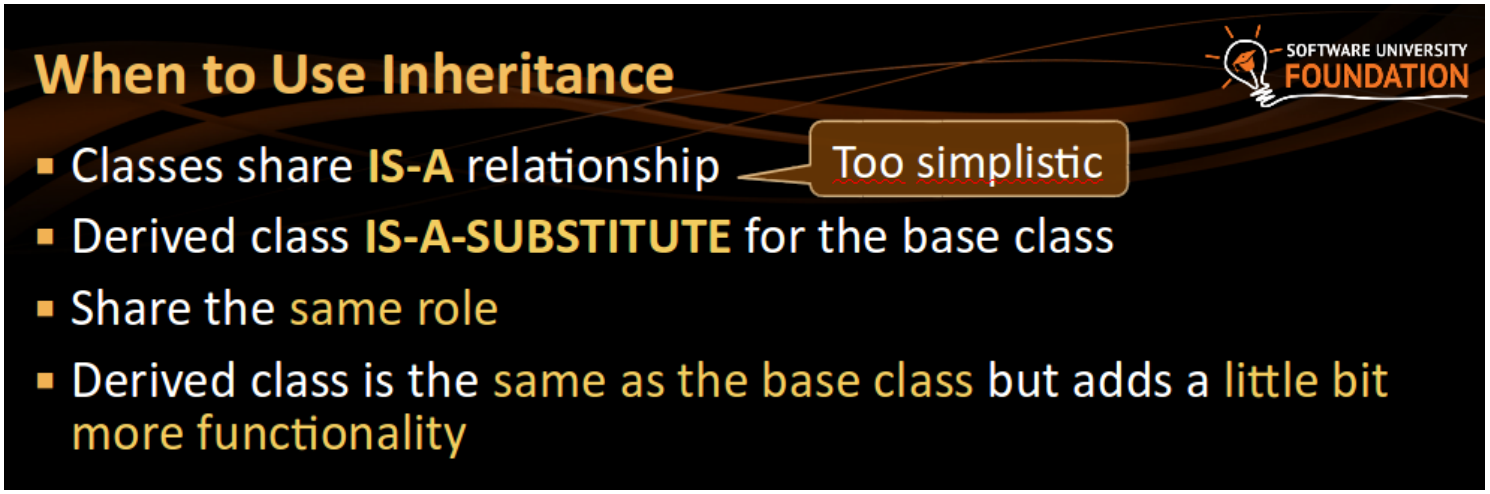

**В един клас можем да си нареждаме обекти от различни други класове и по този начин да си преизползваме кода.**

**Delegation(Делегация)**




Това е този случай, когато един обект предоставя на друг обект да го управлява. Примерно на картинката горе имаме един лаптоп, както и функции за увеличаване и намаляване на яркостта. Между тях имаме и един монитор, който не е показан на външния свят. **НЕ СЛУЧАЙНО ВЗЕМАМЕ ТЕЗИ НЕЩА. ПО ПРИНЦИП НАСЛЕДЯВАНЕТО КОЛКОТО И ХУБАВО ДА ИЗГЛЕЖДА АКО МОЖЕМ ДА МИНЕМ БЕЗ НЕГО.. СЕ СМЯТА ЗА ПО-ДОБРЕ. ПРОСТО ЗАЩОТО УСЛОЖНЯВА КОДА.**

**When to use Inherence?(Кога да използваме наследяване).**




**IS-A** връзка означава, че едното нещо е точно другото. Това е хубаво единствено в началото, за да се разбере по-добре концепцията, обаче в реалността се оказва, че това е прекалено просто и това нещо може да ни докара бъгове, които са много гадни и трудни за хващане. В реалността е хубаво да търсим връзка **IS-A-SUBSTITUDE** между класовете. Тя означавам , че един от класовете - наследникът, трябва да може да замести базовия клас във всеки един случай, и във всеки един контекст. Да вземем пример с човека и студента. Единия има име, другия има училище. Може ли един студент навсякъде да замести човека ? Ами може, защото той просто има още едно име. Друг начин, по който можем да си го представим, е просто като си мислим дали двата класа, които искаме да бъдат родител и наследник, дали имат една и съща роля. Хубаво е също да мислим и по този начин - **двата класа трябва да са абсолютно еднакви, обаче този, който е наследника да добавя нещо много малко**. **Това обаче също не е перфектният случай за наследяване.**

**НАЙ-ДОБРИЯТ ВАРИАНТ, ТОЕСТ СЛУЧАЙ В КОЙТО ИМАМЕ АБСОЛЮТНО ЧИСТО НАСЛЕДЯВАНЕ И КЪМ КОЙТО ТРЯБВА ДА СЕ СТРЕМИМ, МАКАР ЧЕ В МНОГО СЛУЧАИ НЯМА КАК, Е КОГАТО НАСЛЕДНИКЪТ Е НАИСТИНА АБСОЛЮТНО СЪЩИЯ КАТО РОДИТЕЛЯ И ПРОСТО ПОДМЕНЯ, ЗАМЕНЯ НЯКОИ ОТ ФУНКЦИИТЕ МУ СЪС СВОИ СОБСТВЕНИ, КОИТО ПОДЛЕЖАТ НА СЪЩИТЕ ОГРАНИЧЕНИЯ КАТО ТЕЗИ НА РОДИТЕЛЯ.**