The SingleTon Pattern - порождающий шаблон проектирования, те предназначенный для описания способа создания объекта, в данном случае создание объекта с рядом ограничений, сюда же входят абстрактная фабрика, абстрактный метод, Строитель и прототип.Основная идея шаблона, гарантия уникальности создаваемых объектов

Начнем рассмотрение с ключевых вопросов:

1. Как создавать объект:

- Создать ссылочную переменную

- Выделить память

- Вызвать конструктор

2. Создать еще один объект:

- повторить вышеописанные действия

3. Как создать объект класса с приватным конструктором:

- объект можно создать изнутри:

Создаем ссылочную переменную и используем статический метод(действует на уровне всего класса, а не одного экземпляра) данного класса

public class Building {  
  
 private Building() {  
 }  
  
 public static Building getInstance(){  
 return new Building();

}  
}

Этот подход и использует шаблон Синглтон

4. Как гарантировать уникальность объектов:

- подходов много, рассмотрим 2 из них, к последующим вернемся при изучении понятия многопоточность, так как многопоточность дает возможность создавать несколько объектов, даже при использовании этого шаблона, те рассматриваемая версия потоконебезопасна и предназначена для однопоточных сред.

Наполнение класса шаблона:

1. Создаем приватное статическое поле, xранящее ссылку на наш объект

public class SingleTon {  
  
 private static SingleTon *uniqueInstance*;

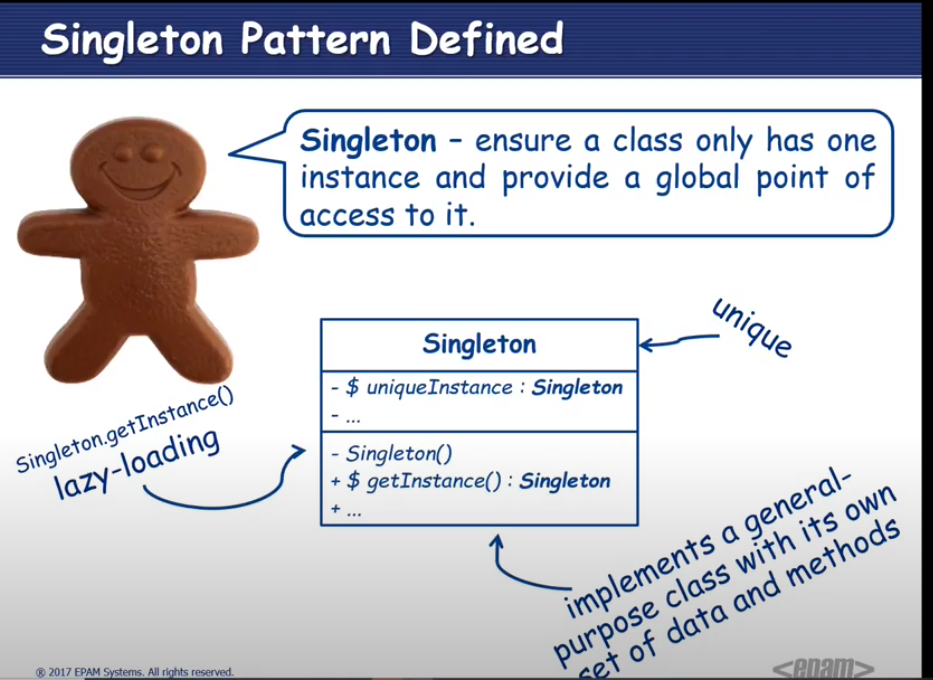
1. Скрываем конструктор

private SingleTon() {  
}

1. Создаем статический метод, который инициализирует ссылку если она еще не инициализирована и возвращает управление на нее или просто возвращает, если инициализирована

public static SingleTon getInstance() {  
 if (*uniqueInstance* == null) {  
 return new SingleTon();  
 }  
 return *uniqueInstance*;  
}

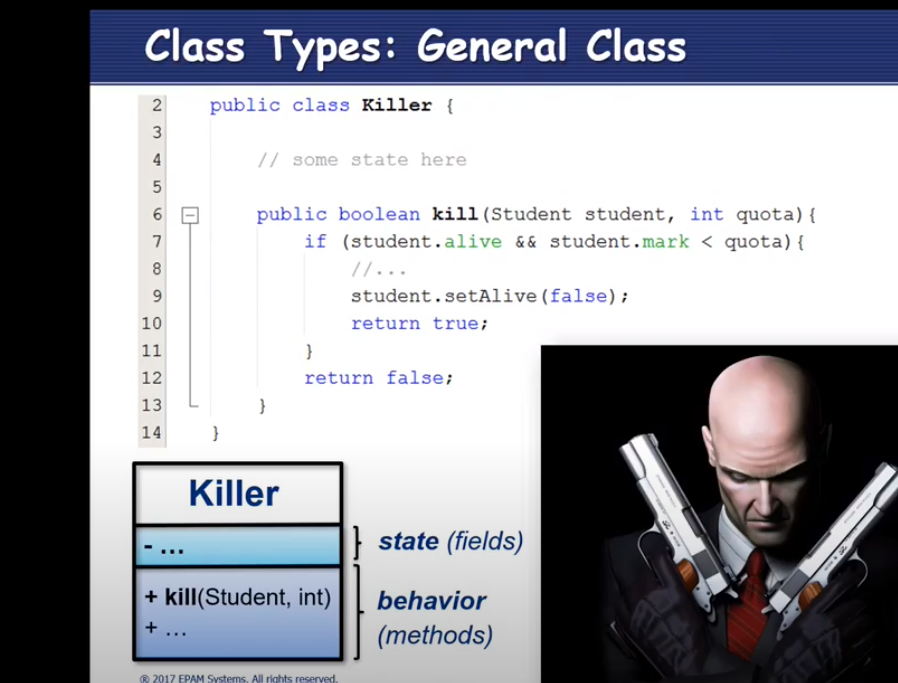
Такая инфраструктура внедряется в те классы, объекты которых должны быть единственного экземпляра, для экономии ресурсов. Если есть точная уверенность, что объект этого класса будет обязательно использоваться в системе, то можно упростить класс, сразу инициализируя поле новым объектом и возвращая его методом getInstance.



Вспомним, конструктор при создании инициализации вызывается в последнюю очередь, перед ним создается ссылка данного класса, 🡪 загружается байткод данного класса, а после начинает работать гарантированная джава инициализация, после инициализируется статика, когда объект класса Class инициализировался, выделяется память под экземпляры соответствующего класса с применением гарантированной дж-инит, далее инициализация во время объявления и только потом конструктор. Стоит упомянуть блоки инициализации безымянные {} внутри класса, которые были пропущены в данном списке, они мб динамические и статические, когда необходимо инициализироать статику (иниц-ся только один, сколько бы экземпляров класса не было создано),мы можем применить статические блоки инициализации, чтобы избежать дублирования кода, ~~а при создании экземпляров класса с тою же целью, то что должно инициализироаться всегда, иниц-ся динамическими блоками~~ и может стать заменой упрощенному подходу к обязательному созданию объекта.

Дополнительные способы использования СинглТон, тк в Java запрещены глобальные переменные, СинглТон может предоставить аналог уровня объекта, до появления интерфейсов синглтон использовали как хранилище данных (переменных).

Если вспомним наш объект киллер -функциональный класс и должен быть единственным в своем роде, можем использовать его как основу для применения шаблона СТ

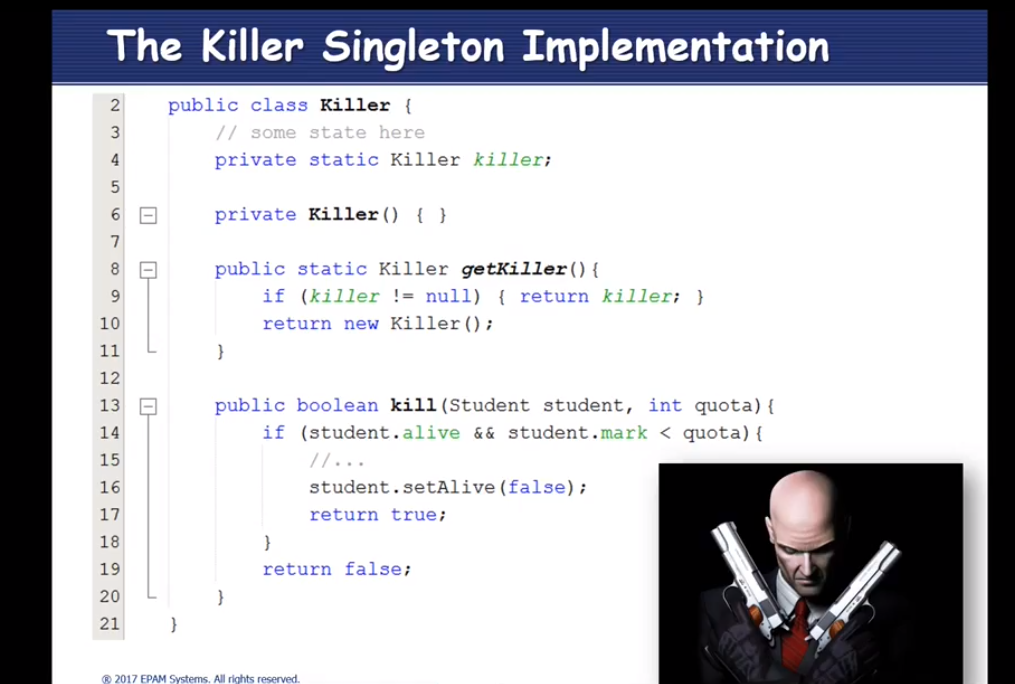
Вспомним исходную структуру

Корректировки:

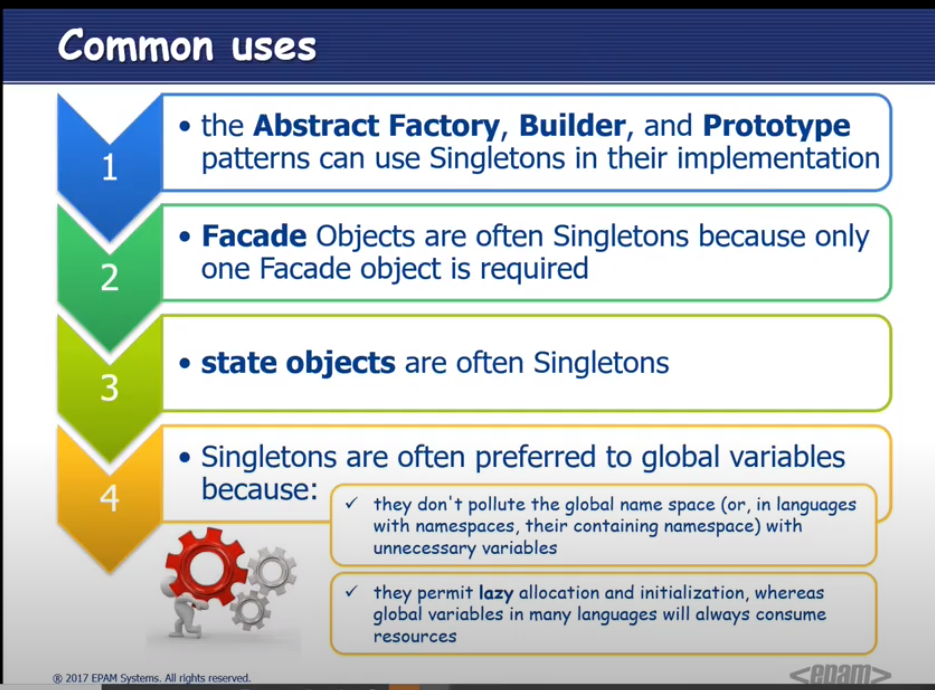
-приватный конструктор

-статический метод для создания объекта

-поле для хранения ссылки на объект



В данном случае не хватает инициализации поля killer, оно всегда будет null(((

Шаблоны на базе которых используется Синглтон

Переходим к рассмотрению перечислений ENUM

До введения перечислений существовало 2 проблемы

Экономия ресурсов и типобезопасность

Рассмотрим типобезопасность на примере статического метода. Который должен возвращать объект своего класса, экземпляр которго зависит от передаваемого параметра

public class Season {  
  
 public class Qwerty{  
 public static Season getSeason(int type){  
 return new Season();  
 }  
 }  
}

предположим у нас 4 сезона, но пользователь может передать аргумент 100, и тк это инт значение, оно будет валидно, 🡪 нужно установить защиту от дурака или сгенерировать исключение, тк до 5й версии JDK не было готовых инструментов для обеспечения потокобезопасности. С введением перечислений на уровне компиляции исключена возможность с помощью ограничения параметризированных методов.

Экономия ресурсов: ограничение создания несуществующих объектов

Реализуем структуру базового класса Season (объект для чтения): поля имя и порядковый номер, конструктор, инициализирующий все состояние, сеттеры и тоСтринг

age itStep.yandr.javaLessons.lesson39;  
  
public class Season {  
 private String name;  
 private int order;  
  
 public Season(String name, int order) {  
 this.name = name;  
 this.order = order;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public int getOrder() {  
 return order;  
 }  
  
 @Override  
 public String toString() {  
 return name + "( " + order + " )";  
 }  
}

Допустим мы создаем календарь и хотим привязать сезоны к месяцам, создаем наши сезоны

public class Main {  
 public static void main(String[] args) {  
 Season s1 = new Season("Winter", 1);  
 Season s2 = new Season("Spring", 2);  
 Season s3 = new Season("Summer", 3);  
 Season s4 = new Season("Autumn", 4);  
 }  
}

но тк в каждом по 3 месяца, то получается нужно создавать по 3 идентичных объекта, это крайне неудобно, лучше использовать шаблон СинглТон, тк сезонов 4 – Кватротон, остается проблема создания объектов которых нет (н-р 5й сезон), значит нужно ограничить эту возможность в самом классе, тк компилятор этого контролировать не может

1 создаем статическое константное поле содержащее ссылку на объект и сразу его инициализируем

public static final Season *WINTER* = new Season("Winter",1);

2 прячем конструктор

private Season(String name, int order) {  
 this.name = name;  
 this.order = order;  
}

теперь создание объекта производится через вызов статической константы, следовательно сколько объектов не будет создано, они просто будут хранить ссылку на это же поле, что позволит серьезно сэкономить ресурс памяти, повторяем действие с остальными сезонами

public static final Season *SPRING* = new Season("Spring",2);  
public static final Season *SUMMER* = new Season("Summer",3);  
public static final Season *AUTUMN* = new Season("Autumn",4);

таким образом мы одновременно и ограничили возможность создания некорректных объектов и сэкономили память.Поля в некоторых случаях необходимо делать тоже константными, но нужно будет или инициализировать их сразу или во всех конструкторах. Параметризация методов также обеспечена, тк невозможно предать объект не соответствующий данным константам

С 5й версии Джава перечисления взяли рутину создания таких объектов на себя, тк они и есть тот же класс, сравним:

package itStep.yandr.javaLessons.lesson39;  
  
public class Season {  
 public static final Season *WINTER* = new Season("Winter",1);  
 public static final Season *SPRING* = new Season("Spring",2);  
 public static final Season *SUMMER* = new Season("Summer",3);  
 public static final Season *AUTUMN* = new Season("Autumn",4);  
 private String name;  
 private int order;  
  
 private Season(String name, int order) {  
 this.name = name;  
 this.order = order;  
 }  
 public String getName() {  
 return name;  
 }  
 public int getOrder() {  
 return order;  
 }  
 @Override  
 public String toString() {  
 return name + "( " + order + " )";  
 }  
}

Все конструкторы enum по умолчанию скрыты

enum EnumSeason {  
  
 *WINTER*("Winter",1), *SPRING*("Spring",2), *SUMMER*("Summer",3), *AUTUMN*("Autumn",4);  
 private String name;  
 private int order;  
   
 EnumSeason(String name, int order) {  
 this.name = name;  
 this.order = order;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public int getOrder() {  
 return order;  
 }  
  
 @Override  
 public String toString() {  
 return name + "( " + order + " )";  
 }  
}

дефолтный конструктор вызывается по умолчанию без указания ()

WINTER, SPRING, SUMMER, AUTUMN;

EnumSeason() { }

\*valueOf() метод который позволяет возвратить объект при совпадении с полем передаваемой строки

Чаще всего enum встречается в виде класса , пречисляющего константы,без дополнительного функционала, такие перечисления называются “чистыми”

enum DayOfWeek{  
 *MONDAY*, *TUESDAY*, *WEDNESDAY*,*THURSDAY*,*FRIDAY*,*SATURDAY*,*SUNDAY*}

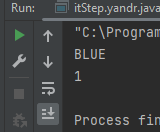
*\*попробовать в сущностях своего проекта какую-то характеристику вынести в enum, создать пользовательский тип данных на базе перечислений, например:*

enum FlowerColor {  
 *RED*, *BLUE*, *GREEN*, *YELLOW*}

Тк перечисления как и все остальные объекты относятся к классу Object,но напрямую наследуются от класса ENUM то обладают минимальным функционалом из коробки, присущим обоим родителям:

Object: toString, equals, hashCode и тд…при этом equals (сравнивает по дефолтным параметрам: имя и порядковый номер, которые присваиваюся имя – по имени константы, номер от 0 по порядку добавления, операторы неравенства тж перегружены на сравнение по этим параметрам)и hashCode уже перегружены

Enum: name, ordinal и тд

 FlowerColor color = FlowerColor.*BLUE*;  
 System.*out*.println(color);  
 System.*out*.println(color.ordinal());  
  
}

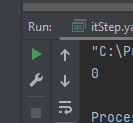
Тк Java поддерживает единственное наследование классов, ни от какого другого класса Enum больше наследоваться не может

Рассмотрим подробнее данный базовый класс:

-Имплементит Comparable🡪 значит любые объекты перечислений можно сравнивать 🡪 закидывать в контейнер, сортировать и тд

-Константы name & ordinal и одноименные методы, возвращающие их значения

-Конструкторы, неявно вызываемые и инициализирующие все дефолтное состояние объекта

-Статический метод valueOf ищет объект по совпадению полей со строкой, введенной в аргументе

FlowerColor g = FlowerColor.*valueOf*("RED");  
System.*out*.println(g.ordinal());

-Статический метод values(), который возвращает перечисление в виде массива, таким образом можно проверить на соответствие имен перечисления , обойдя вызов исключения

String name = "Red";  
for (FlowerColor color:FlowerColor.*values*()){  
 if (name.equals(color.name())){  
 FlowerColor c = FlowerColor.*valueOf*(name);  
 System.*out*.println(c);  
 }

-switch тж адаптирован под использование перечислений, при этом в кейсах можно использовать сокращенный вызов перечислений:

EnumSeason season = null;  
switch (season) {  
 case *WINTER*:  
 System.*out*.println(0);break;  
 case *SPRING*:  
 System.*out*.println(1);break;  
 case *SUMMER*:  
 System.*out*.println(2);break;  
 case *AUTUMN*:  
 System.*out*.println(3);break;  
 default:  
 throw new IllegalStateException("Unexpected value: " + season);  
}

Тж стоит отметить что наследоваться от всех классов -перечислений нельзя, тк конструктор скрыт, а при создании дочернего объекта в конструкторе , обязательно вызывается конструктор базового класса.

Вернемся к нашему классу -перечислению сезонов, обычно дефолтный конструктор в классах перечислений используется, только тогда, когда мы хотим дать возможность поменять пользовательское состояние, когда поля не константны, в нашем случае, не используем.

Обычно перечисление используется при ограниченном количестве объектов данного класса.

Вернемся к Синглтон, теперь понимая ,как его упростить при помощи перечислений. Рассмотрим на примере нашего киллера для студентов:

-Инфраструктура Синглтон

class Killer {  
 private static Killer *KILLER*;  
  
 private Killer() {  
 }  
  
 public void kill() {  
  
 }  
  
 public static Killer getKILLER() {  
 return *KILLER*;  
 }  
}

на базе enum, дефолтный конструктор и так скрыт-удаляем, объект мы получаем на имени класса, 🡪getKiller тоже удаляем, ссылка на объет уже статик и прайвет, оставляем только имя.

enum Killer {  
 *KILLER*;  
  
 public void kill() {  
  
 }  
  
}

таким образом мы рассмотрели наиболее распространенную реализацию потокобезопасного СинглТон