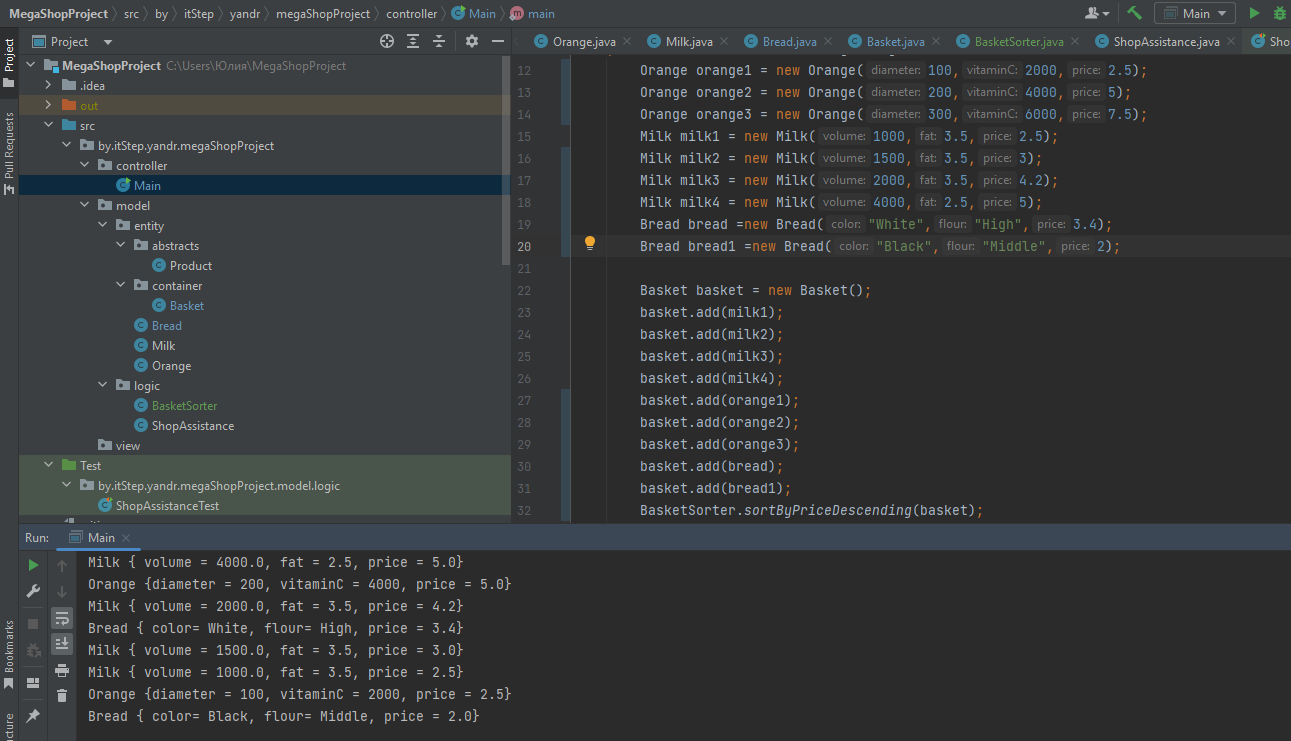
Модернизация системы MegaShopProject:

Для реализации сортировок элементов нашей, добавим метод для замены элемента через индекс:

public void setProduct(int index, Product product){  
 products[index]=product;  
  
}

Теперь создаем класс BasketSorter и реализовываем сортировки ( пузырьковая)по цене, по убыванию и по возрастанию:

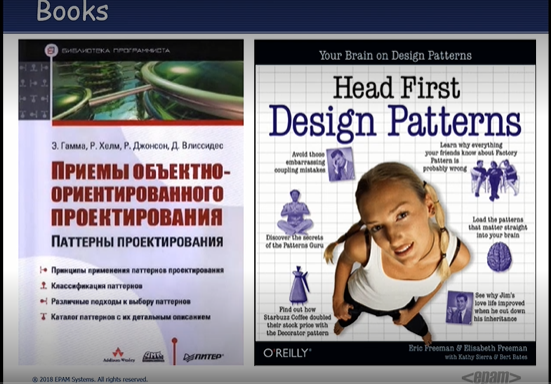
package by.itStep.yandr.megaShopProject.model.logic;  
import by.itStep.yandr.megaShopProject.model.entity.abstracts.Product;  
import by.itStep.yandr.megaShopProject.model.entity.container.Basket;  
  
public class BasketSorter {  
 public static void sortByPriceAscending(Basket basket) {  
 for (int i = 0; i < basket.getProductsCount() - 1; i++) {  
 for (int j = 0; j < basket.getProductsCount() - 1 - i; j++) {  
 if (basket.getProduct(j).getPrice() > basket.getProduct(j + 1).getPrice()) {  
 Product temp = basket.getProduct(j);  
 basket.setProduct(j, basket.getProduct(j + 1));  
 basket.setProduct(j + 1, temp);  
 }  
 }  
 }  
 public static void sortByPriceDescending(Basket basket) {  
 for (int i = 0; i < basket.getProductsCount() - 1; i++) {  
 for (int j = 0; j < basket.getProductsCount() - 1 - i; j++) {  
 if (basket.getProduct(j).getPrice() < basket.getProduct(j + 1).getPrice()) {  
 Product temp = basket.getProduct(j);  
 basket.setProduct(j, basket.getProduct(j + 1));  
 basket.setProduct(j + 1, temp);  
 }  
 }  
  
}

На этом занятии рассмотрим шаблон Strategy , стратегию изменения поведения, без переписывания кода:

Выделил этот шаблон Александр Кристофер, вместе со своей командой –бандой 4х(взяв за основу шаблон проектирования домов)

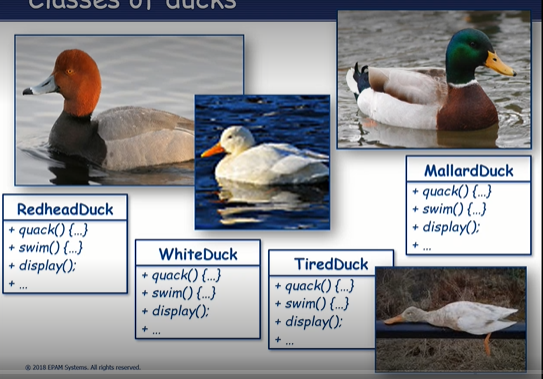


GOF шаблоны строятся на SOLID принципах

Рекомендуемая литература:

Шаблон Стратегия используется для динамических изменений поведения, без переписывания кода, при правильной архитектуре проекта,🡪 относится к поведенческим шаблонам проектирования

Предположим мы создаем симулятор уток:



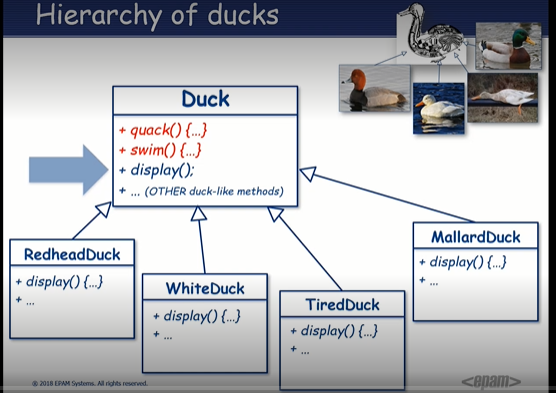
Поведение:

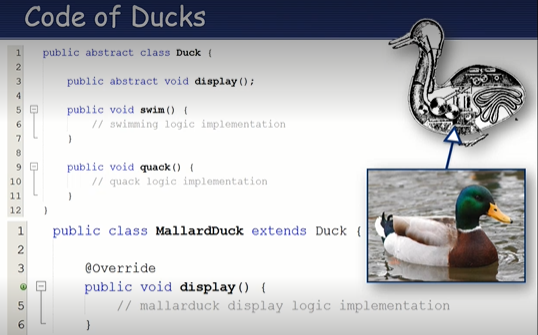
-плавать

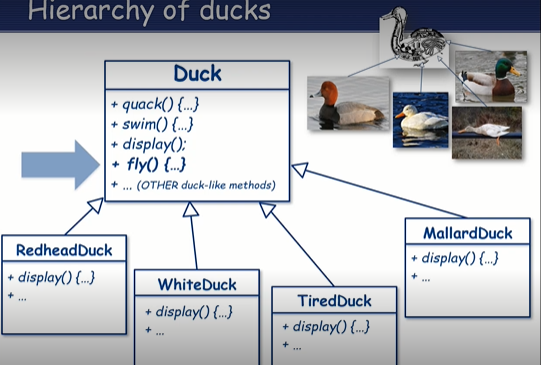
-крякать

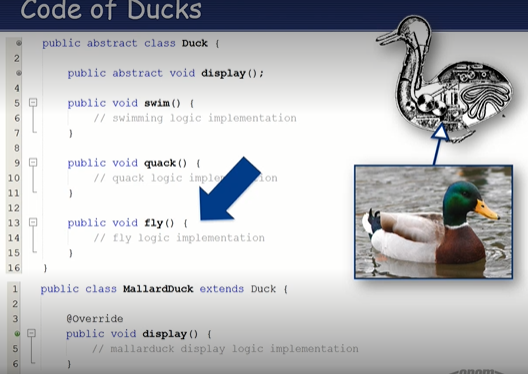
-отображаться

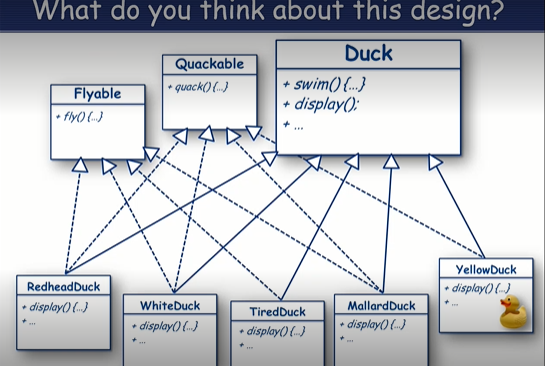
И если отображение будет индивидуальным, то остальное поведение явно должно быть вынесено в абстракцию, на уровень выше





 Однака при расширении функционала базового класса, мы сталкиваемся с проблемой, что его могут прbобрести и те производные классы для которых он не предназначался:



Один из минусов наследования, это то что добавляя в родительский класс поведение или состояние, оно наследуется всеми производными без исключения. Требуется найти способ декомпозировать систему. Одно из решений вынести поведение в IF ,но тогда мы теряем повторное использование кода, тк потребуется реализовать поведение в каждом производном классе

Решение описывается в принципах дизайна архитектуры: выделить те аспекты объектов, которые различны и вынести их за пределы основной системы. Сейчас у утки свои крылья и голос, теперь мы хотим вынести эти аспекты за пределы ее сущности.У нас будут 2 типа поведения: класс для существ которые будут летать и класс для тех, кто будет крякать.

Basic Type У утки должны будут быть поля зависящие от описанного  
 в данных классах поведения, создаем объекты:

Flyable

fly()

1.полет с крыльями

2.не летающий

Теперь утка летает не сама, а с помощью объекта который

FlyNoWay

fly()

FlyWithWings

fly()

дает ей такую возможность

Basic Type

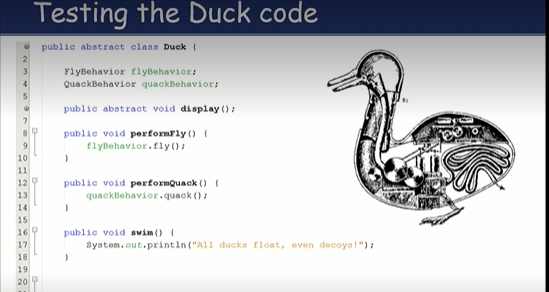
Flyable flyable = ссылка на объект поведения

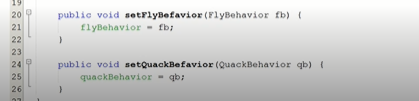
Quackable

quack()

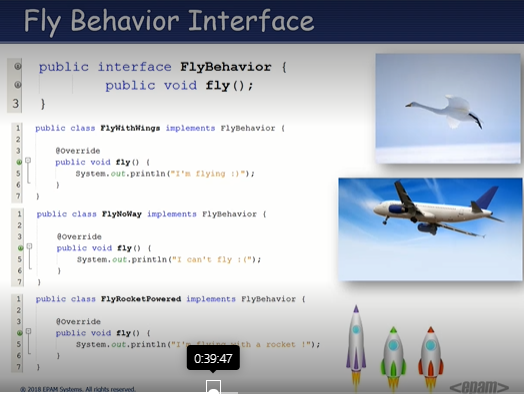
На базе IF создаем классы реализующие поведение.

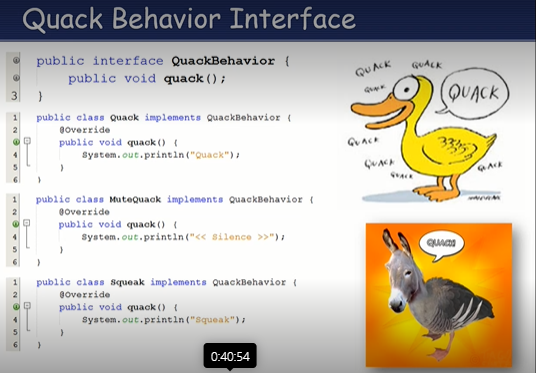
Теперь мы можем изменять вынесенный аспект не изменяя кода, тк тепрь просто передаем его исполнителям,заменив наследование на композицию(реализация полей других типов), на которой и основан шаблон Стратегия

Добавляем ссылки на объеты для реализации поведения в базовом классе

Тж добавим методы, которые будут принимать новые объекты реализации поведения и заменять ими старые

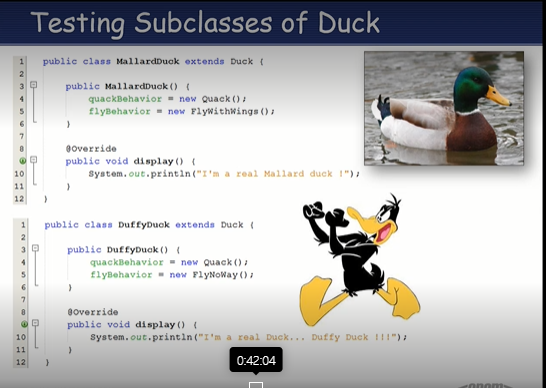
Реализуя возможность динамического изменения поведения, соответствующую O/CP

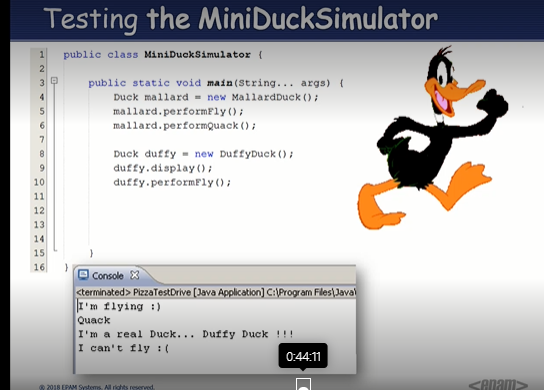
 Добавляем классы инкапсулирующие поведение



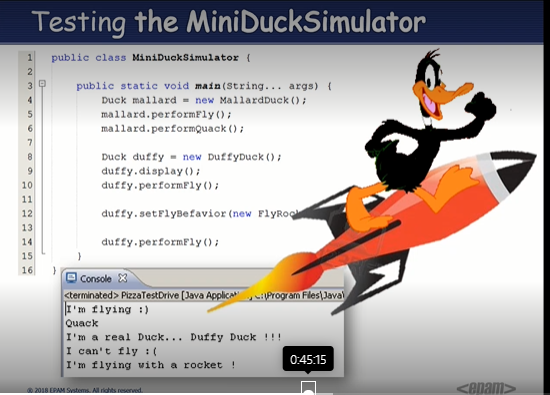
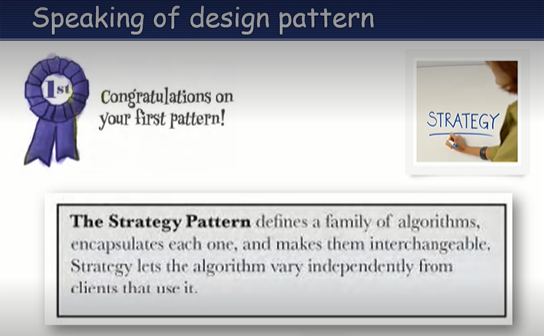
Создавая объекты этих классов. Мы внедряем конкретную имплементацию поведения

Переходим к производным классам, наследуясь от Duck, они приобретают объекты поведения в виде композиции и требуется их реализовать:

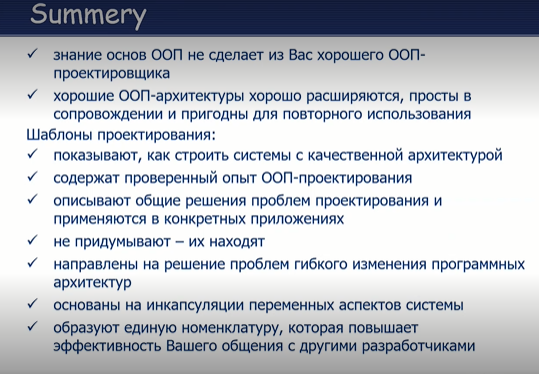




Принципы шаблона Стратегия:

* Выноси изменяющиеся аспекты поведения объектов за пределы системы
* Предпочитай интерфейс имплементации
* Используй композицию перед

Подведем итог: шаблон Стратегия определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми



В случае с MegaShopMarket один аспект поведения тж требует применения данного шаблона: сортировка – повозрастанию, убыванию цены, каллоража и пр….Нам необходимо определить семейство алгоритмов сравнения 2х продуктов по определенному параметру.

Для этого в папке logic создаем новую папку sortStrategy, в которой создаем IF ProductSortable, в который добавляем абстрактный метод сравнения:

package by.itStep.yandr.megaShopProject.model.logic.sortStrategy;  
  
import by.itStep.yandr.megaShopProject.model.entity.abstracts.Product;  
  
public interface ProductSortable {  
 boolean compare (Product pr1,Product pr2);  
}

Теперь добавляем класс с имплементациейв которую и выносим логику сравнения по цене по возрастанию

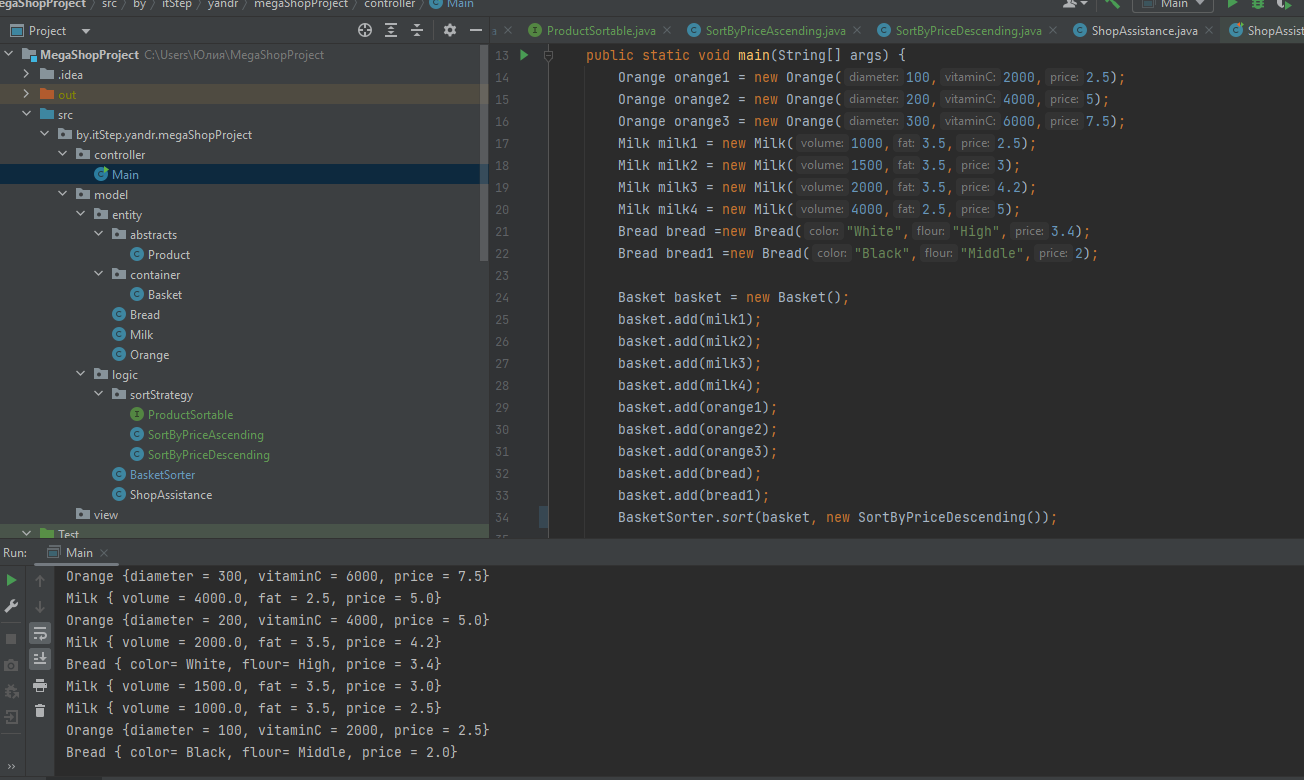
package by.itStep.yandr.megaShopProject.model.logic.sortStrategy;  
  
import by.itStep.yandr.megaShopProject.model.entity.abstracts.Product;  
  
public class SortByPriceAscending implements ProductSortable {  
 @Override  
 public boolean compare(Product pr1, Product pr2) {  
 return pr1.getPrice() > pr2.getPrice();  
 }  
}

Аналогично для сортировки по убыванию

package by.itStep.yandr.megaShopProject.model.logic.sortStrategy;  
  
import by.itStep.yandr.megaShopProject.model.entity.abstracts.Product;  
  
public class SortByPriceDescending implements ProductSortable {  
 @Override  
 public boolean compare(Product pr1, Product pr2) {  
 return pr1.getPrice() < pr2.getPrice();  
 }  
}

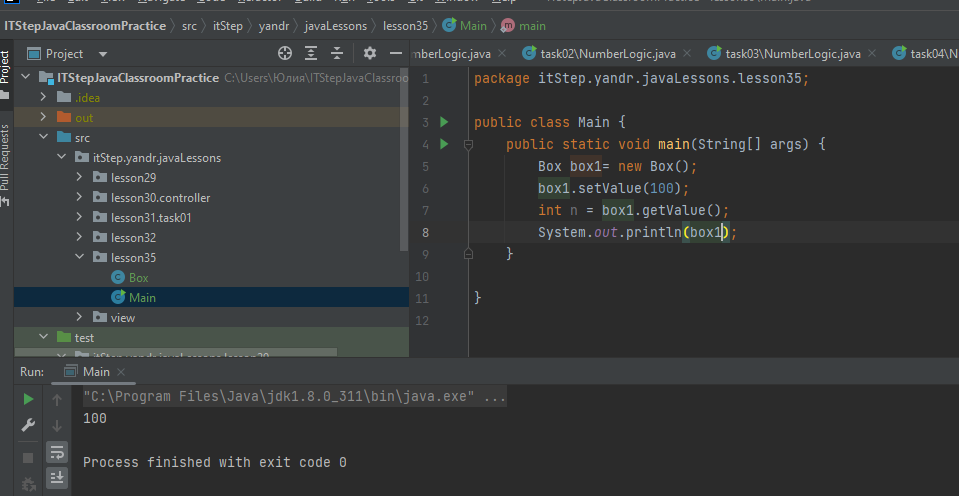
Теперь в класс сортировки добавляем аргумент, в виде объекта сортировки и заменяем на него условие в условной конструкции

package by.itStep.yandr.megaShopProject.model.logic;  
  
import by.itStep.yandr.megaShopProject.model.entity.abstracts.Product;  
import by.itStep.yandr.megaShopProject.model.entity.container.Basket;  
import by.itStep.yandr.megaShopProject.model.logic.sortStrategy.ProductSortable;  
  
public class BasketSorter {  
 public static void sort (Basket basket, ProductSortable sortable) {  
 for (int i = 0; i < basket.getProductsCount() - 1; i++) {  
 for (int j = 0; j < basket.getProductsCount() - 1 - i; j++) {  
 if (sortable.compare(basket.getProduct(j), basket.getProduct(j + 1))) {  
 Product temp = basket.getProduct(j);  
 basket.setProduct(j, basket.getProduct(j + 1));  
 basket.setProduct(j + 1, temp);  
 }  
 }  
 }  
 }

Теперь мы можем изменять логику сравнения, тк она инкапсулирована в объектах, передаваемых в качестве аргумента

Предположим нам заказали программу с контейнерами для хранения целочисленных значений:

package itStep.yandr.javaLessons.lesson35;  
  
public class Box {  
 private int value;  
  
  
 public Box(int value) {  
 this.value = value;  
 }  
  
 public Box() {  
  
 }  
  
 public int getValue() {  
 return value;  
 }  
  
 public void setValue(int value) {  
 this.value = value;  
 }  
  
 @Override  
 public String toString() {  
 return value + "";  
 }  
}

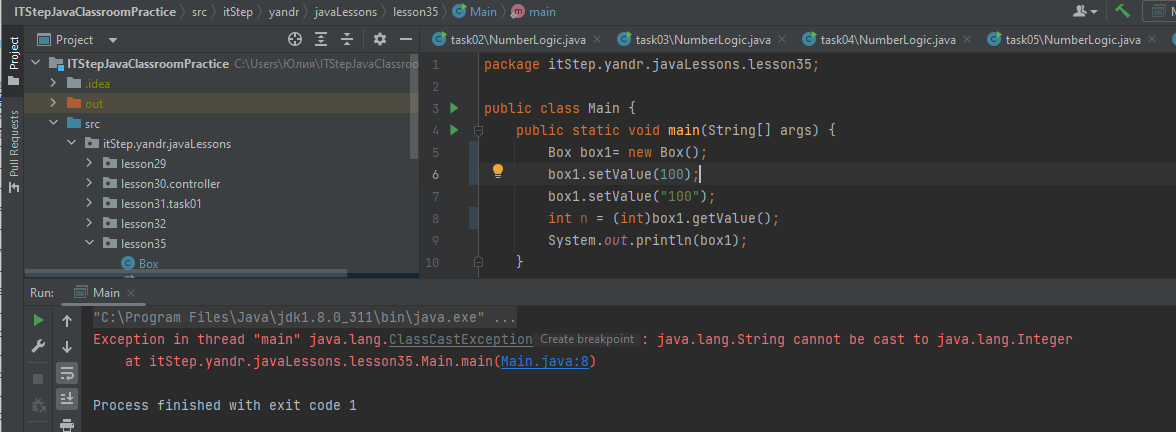


Если же нам понадобится создать такое же приложение для другого типа данных, мы просто скопируем и поменяем тип данных

package itStep.yandr.javaLessons.lesson35;  
  
public class Sbox {  
  
 private String value;

public Sbox(String value) {  
 this.value = value;  
 }  
  
 public Sbox() {  
  
 }  
  
 public String getValue() {  
 return value;  
 }  
  
 public void setValue(String value) {  
 this.value = value;  
 }  
  
 @Override  
 public String toString() {  
 return value + "";  
 }  
}

Для того чб избежать CP до 5.0 JVM использовали класс Object как максимально базовый, но этот способ не обеспечивал типобезопасности

package itStep.yandr.javaLessons.lesson35;  
  
public class Box {  
 private Object value;  
  
  
 public Box(Object value) {  
 this.value = value;  
 }  
  
 public Box() {  
  
 }  
  
 public Object getValue() {  
 return value;  
 }  
  
 public void setValue(Object value) {  
 this.value = value;  
 }  
  
 @Override  
 public String toString() {  
 return value + "";  
 }

Для ее обеспечения возможно проверять соответствие типу при помощи оператора Inctanceof

package itStep.yandr.javaLessons.lesson35;  
  
public class Main {  
 public static void main(String[] args) {  
 Box box1 = new Box();  
 box1.setValue(100);  
 box1.setValue("100");  
 Object o = box1.getValue();  
 if (o instanceof Integer) {  
 int n = (int) o;  
 } else if (o instanceof String) {  
 String s = (String) o;  
 }  
 System.*out*.println(box1);  
 }  
  
}

Но при необходимости расширения, код понадобится дополнять, следовательно система не соответствует OC/P.Именно по-этому с JVM 5.0 ,была введена технология Generic (на базе которой строится Collection FrameWork). Это по сути трафарет типа, параметризация. При реализации инфракструктуры класса, необходимо абстрагироваться от параметра типа объекта (стираем его).Создание дженерализированного класса, который будет работать с объектом того типа , который будет передан ему в качестве параметра, при создании объекта этого класса, происходит путем добавления “французских скобок, с указанием в них типа TYPE” при объявлении класса, по сути тип объекта также становвится параметром :

package itStep.yandr.javaLessons.lesson35;  
  
public class Box <T> {  
 private T value;  
  
  
 public Box(T value) {  
 this.value = value;  
 }  
  
 public Box() {  
  
 }  
  
 public T getValue() {  
 return value;  
 }  
  
 public void setValue(T value) {  
 this.value = value;  
 }  
  
 @Override  
 public String toString() {  
 return value + "";  
 }  
}

Дженерализированный класс, это класс в котором заменяется тип объекта, с которым должен работать данный класс. Дефолтное значение типа объекта данного класса – Object.

Указывать имя переменной типа рекомендуется с большой буквы, тк это плейсхолдер, чв T для универсальных классов или E element , K – key и пр

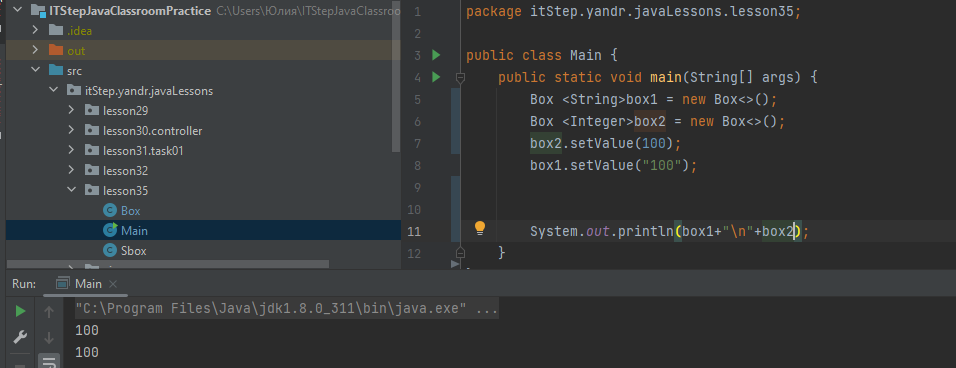
Теперь при создании объекта данного класса, мы просто указываем тип объекта с которым будем работать, используя diamond-operator

Box <String>box1 = new Box<String>();

До JDK 7.0 необходимо было дублировать тип в самом объекте, после только в ссылке

Box <String>box1 = new Box<>();

При этом класс , использующий объект данного типа создается “на лету”, стирая значение Дженерик.



Примечание:

InstanceOf определяет принадлежность (какому классу удоблетворяет)класса с учетом полиморфности, те будет определять все производные также , как базовый

class A{  
  
}  
class B extends A{  
  
}  
class C extends B {  
  
}  
  
public class Main {  
 public static void main(String[] args) {  
 A a = new C();  
 System.*out*.println(a instanceof A); true  
 System.*out*.println(a instanceof B); true  
 System.*out*.println(a instanceof C); true

Следовательно при переопределении возникает риск сбоя системы при попытке вызова поля или поведения более ссылкой базового класса.В отличии от getClass (нельзя перреопределять final), который указывает конкретно на тип объекта

System.*out*.println(a.getClass()==new A().getClass());false  
System.*out*.println(a.getClass()==new B().getClass()); false  
System.*out*.println(a.getClass()==new C().getClass());true