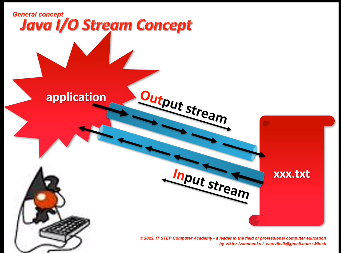
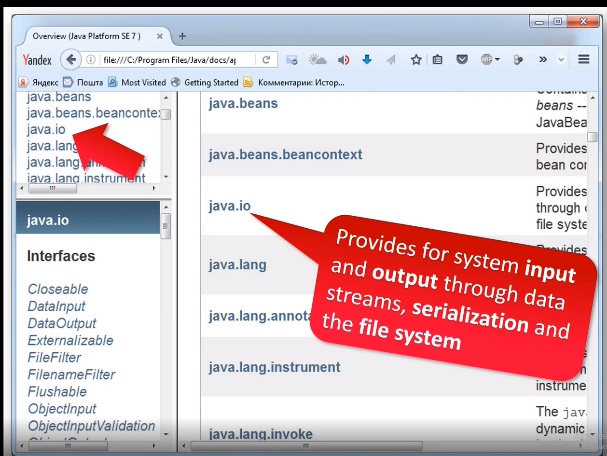
Потоки ввода вывода Java I/O Stream

Основная концепция:

- Возможность записи чтения данных, базируется на принципе единственной ответственности, есть семейство классов которое базируется на функционале чтения, есть семейство классов, которое базируется на записи данных.



Вся библиотека будет основана на абстракциях, реализация которых зависит от источника (фаил, именованные и неименованные каналы и пр ). Необходимые библиотеки находятся в пакете java.io, основана на паттерне декоратор или java.nio , который отличается по концепции – основана на буферах.



Вспомним существующие типов файлов:

-текстовые – набор символов(байт код, ассоциирующийся с символами)

-бинарные –все остальные, аудио ,видео, выполняемые и пр

Все стримы, помимо того, что делятся на читающие и записывающие, делятся по типу файлов. Рассмотрим отличия типов файлов, с точки зрения программы:

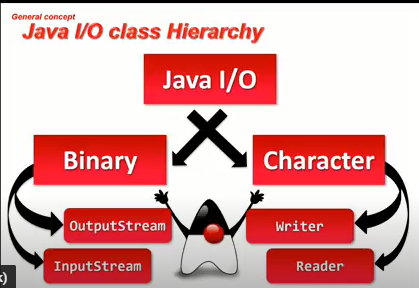
Текстовые- все есть текст, а значит фаил состоит из строк, они в свою очередь из символов,на этом и основана абстракция для парсинга(анализа и расшифровки ) данных, будем посимвольно бежать по файлу, если построчно – до символа новой строки. Текстовая информация универсальна, с точки зрения переноса ее с устройства на устройство, разными ОС, тк кодировка шифрования может быть установлена везде и при удалении части данных мы сможем прочитать оставшееся содержимок, тк нет никаких алгоритмов записи.С другой стороны их сложнее структурировать-🡪 их сложнее парсить, несмотря на легкость чтения.

Бинарные – имеют определенную структуру, возьмем любую картинку и прочитаем ее содержимое с помощью программы Lister, мы увидим байты расшифровать с помощью стандартных кодировок мы не сможем, только программа -редактор сможет распознать данный тип данных, так как знает структуру этого файла и способы его чтения, исходя из его расширения иногда используя кодеки- таблицу кодировки. Открыв дан памяти мы увидим зашифрованные данные, набор байт, содержащий конфигурационные настройки, ширину, высоту, глубину цвета, палитру сам рисунок- зашифрованные пиксели итд

Особенности:

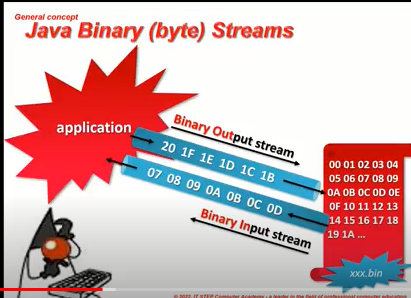
-Структура орпеделена🡪 легкий парсинг

-Низкая безопасность, при повреждении ключевых компонентов структуры, остальные данные тоже теряют смысл



Подведем итог:

Все потоки в Джава делятся на 2 направления: бинарные и символьные, которые в свою очередь делятся на входящие(чтение) и исходящие(запись). Это очень удобно с точки зрения читабельности кода.



Рассмотрим работу с бинарными потоками:

Единицей информации потока будет один байт, читаем 20 байт, значит 20 вызовов потока.Стоит отметить, что если использовать низкоуровненые реализации записи, любое число, значение которого выходит за пределы 1 байта(256) пришлось бы разрезать перед записью, а потом обратно склеивать…

Создаем тестовый проект: класс BinaryWorker в котором будет 2 метода: для чтения и записи данных. Подключаем java.io.В методе для записи используем OutputStream, заоглянем в структуру класса:

Все объекты данного класса имплеменят:

Closeable- с 8й версии расширяется от AutoCloseable для возможности использования try with Resources, как и все что работает с внешними источниками внешних данных , генерирует базовое исключение уровня библиотеки IOexception , можно специфицировать

Flushable-(только для записывающих данные потоков) гарантия буферизации перед закрытием источника, тоже генерирует IOexception

Основная задача данного класса записывать данные реализуется методом write (2 перегруженные версии) главный метод абстрактен и принимает интовское значение(тк система лучше всего работает с эти типом, на самом деле из 4х байт при передаче используется только последний, по-этому единица записи по-прежнему равна 1 байту, 0-255),перегруженные его вызывают (при объявлении не указана абстрактность)

public abstract class OutputStream implements Closeable, Flushable {  
 public abstract void write(int b) throws IOException;

public void write(byte b[]) throws IOException {  
 write(b, 0, b.length);}

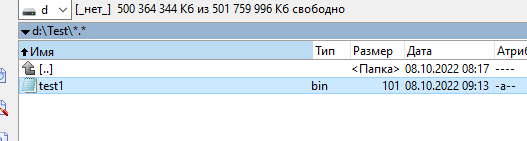
смещение и длина записи  
 public void write(byte b[], int off, int len) throws IOException {  
 if (b == null) {  
 throw new NullPointerException();  
 } else if ((off < 0) || (off > b.length) || (len < 0) ||  
 ((off + len) > b.length) || ((off + len) < 0)) {  
 throw new IndexOutOfBoundsException();  
 } else if (len == 0) {  
 return;}  
 for (int i = 0 ; i < len ; i++) {  
 write(b[off + i]);  
 }  
 }

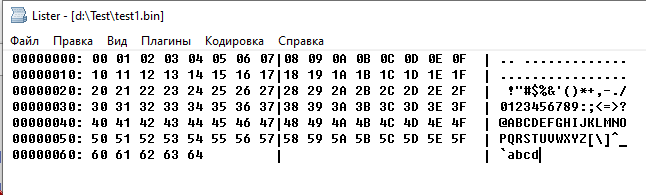
Создаем поток , используя файловую абстракцию, которая записывает данные в указанный файл, принимает путь или имя в текущей директории(если не находит его по имени-создает), обрабатываем исключения и после использования (записываем в файл значения счетчика) потока буфферизируем и закрываем его.

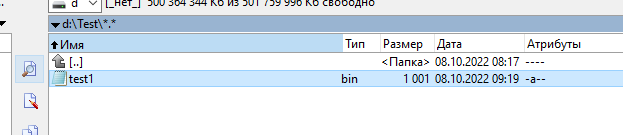
package itStep.yandr.javaLessons.lesson41;  
  
import java.io.\*;  
  
public class BinaryWorker {  
  
 public static void write(String FileName) {  
 OutputStream stream = null;  
 try {  
 stream = new FileOutputStream(FileName);  
 for (int i = 0; i < 101; i++) {  
 stream.write(i);  
 }  
 stream.flush();  
 } catch (IOException e) {  
 System.*err*.println(e);  
 } finally {  
 try {  
 if (stream != null) {  
 stream.close();  
 }  
 } catch (IOException e) {  
 System.*err*.println(e);  
 }  
  
 }  
 }  
}

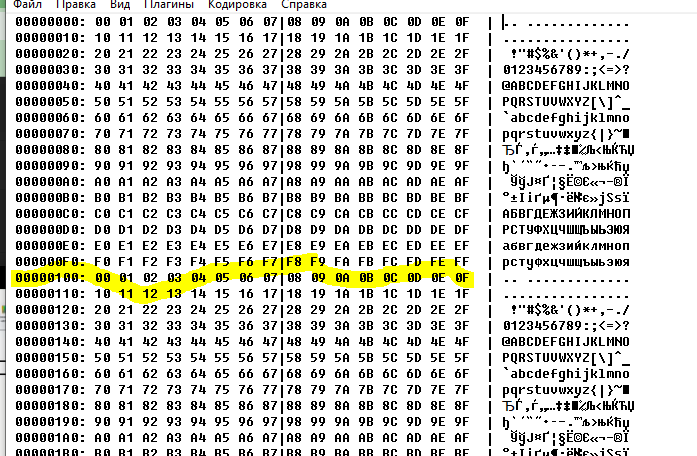
Так как метод статический, создаем переменную с указанием пути к файлу \* тк в строке один слэш =указание управляющего символа, то указываем 2 или один обратный, в файле можем указать любое расширение

package itStep.yandr.javaLessons.lesson41;  
  
public class Main {  
 public static void main(String[] args) {  
 String FileName = "D://Test//test1.bin";  
 BinaryWorker.*write*(FileName);  
  
 }  
}

 Создаем директорию Тест на диске Д и запускаем код и видим созданный файл размером ровно 101 байт



Попробуем записать значения до 1000, меняем вцикле 101 на 1001 и видим, что размер файла изменился,

но на значениях свыше 255, запись пошла заново, как и говорили ранее одна единица потока- 1 байт

Теперь попробуем считать этот файл, используя эту же абстракцию, создаем метод read по образцу write , только он будет возвращать прочитанное в виде строки и меняем иерархию потоков на InputStream, заглянем в нее:

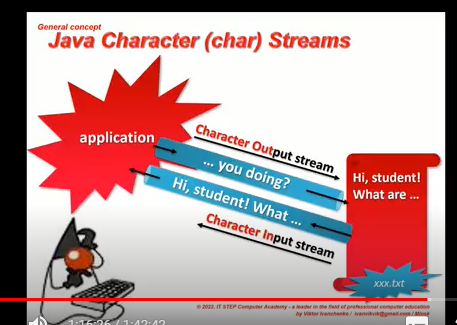
Имплементирует только closeable и видим абстрактный метод read(), возвращающий значение int, но как и при записи используется из него только последний байт, также перегруженные версии с массивом и егго частью, доходя до конца файла возвращает - 1. Метод skip() читать не с начала, а с пропуском, принимая пропущенное количество байт в виде лонг зн-я, вызывается перед read, available()-проверка есть ли данные для чтения, конечно унаследованный close(), синхронизированный mark – как возможность расставить метки и потом по ним читать.

Используем побайтовое чтение, сохраняя результат при помощи StringBuilder

public static String read(String FileName) {  
 StringBuilder builder = new StringBuilder();  
 InputStream stream = null;  
 try {  
 stream = new FileInputStream(FileName);  
 if (stream.available() != 0) {  
 int temp;  
 while ((temp = stream.read()) != -1) {  
 builder.append(temp).append(" ");  
 }  
 }  
 } catch (IOException e) {  
 System.*err*.println(e);  
 } finally {  
 try {  
 if (stream != null) {  
 stream.close();  
 }  
 } catch (IOException e) {  
 System.*err*.println(e);  
 }  
 }  
 return builder + "";  
}

public class Main {  
 public static void main(String[] args) {  
 String FileName = "D://Test//test1.bin";  
 System.*out*.println(BinaryWorker.*read*(FileName));  
 }



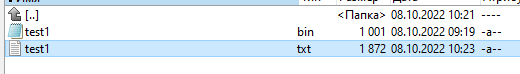
Перейдем к символьным потокам записи и чтения, единицей работы потока будет символ, но тк в разных кодировках размер символов отличается, то и размер файла будет разным. Методы для работы с сисмволами тоже основанны на Integer, но из этого значения используется столько сколько занимает символ. Абстракцией записи потока символьных данных является Writer, рассмотрим его подробней:

Класс также абстрактен

Дополняется имплементацией Appendable

Метод writ() помимо стандартных для потока перегрузок записи дополнен перегрузки принимающие строку

Зеализуем метод для записи 1000 символов

package itStep.yandr.javaLessons.lesson41;  
import java.io.\*;  
public class CharacterWorker {  
  
 public static void write(String FileName) {  
 Writer stream = null;  
 try {  
 stream = new FileWriter(FileName);  
 for (int i = 0; i < 1000; i++) {  
 stream.write(i);  
 }  
 stream.flush();  
 } catch (IOException e) {  
 System.*err*.println(e);  
 } finally {  
 try {  
 if (stream != null) {  
 stream.close();  
 }  
 } catch (IOException e) {  
 System.*err*.println(e);  
 }  
 }  
 }  
}

!!! первые 128 символов в любой кодировке заимают 1 байт, остальные 2 – поэтому размер файла после записи 1000 символов -1872 байта

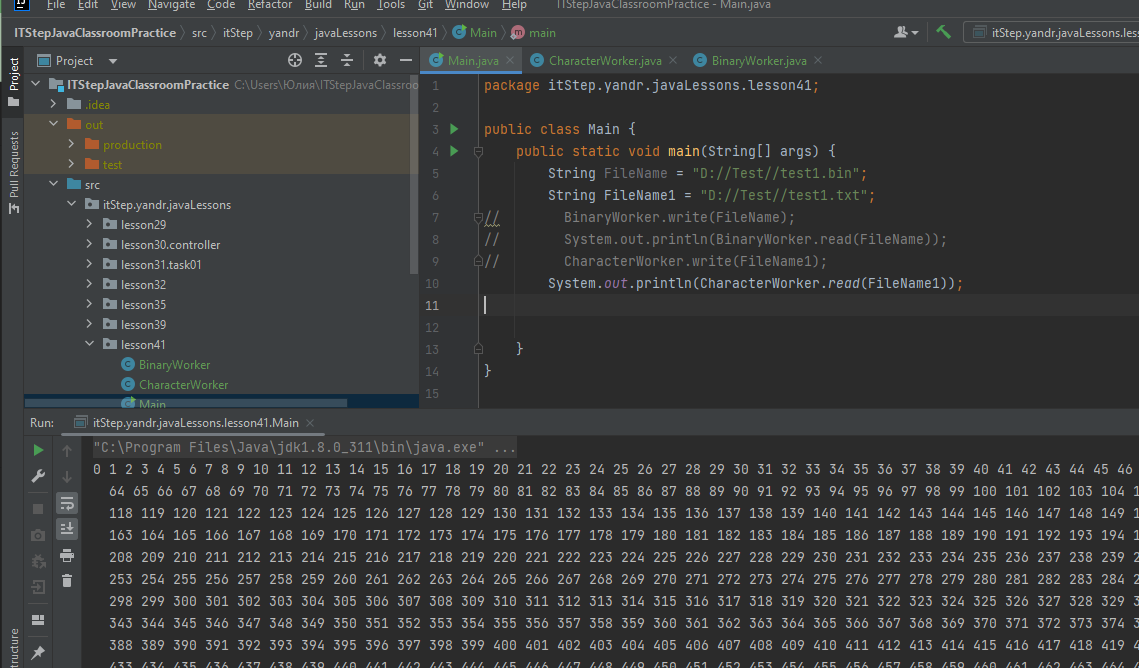
Перейдем к чтению данных, абстракция Reader:

Дополненная имплементацией Readable противоположность Appendable

Вбстрактная версия метода read читает массив со смещением, остальные ее вызывают

Основные методы skip и ready

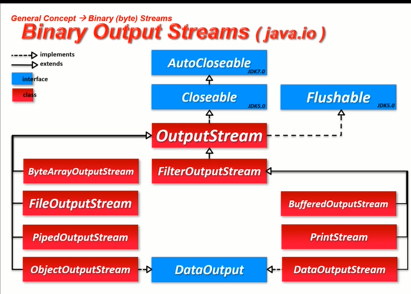
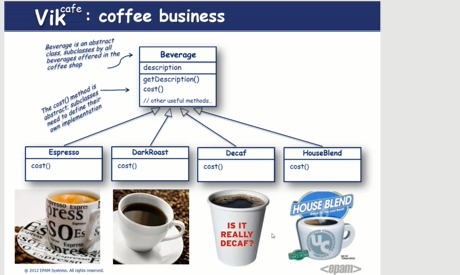
public static String read(String FileName) {  
 StringBuilder builder = new StringBuilder();  
 Reader stream = null;  
 try {  
 stream = new FileReader(FileName);  
 if (stream.ready()) {  
 int temp;  
 while ((temp = stream.read()) != -1) {  
 builder.append(temp).append(" ");  
 }  
 }  
  
 } catch (IOException e) {  
 System.*err*.println(e);  
 } finally {  
 try {  
 if (stream != null) {  
 stream.close();  
 }  
 } catch (IOException e) {  
 System.*err*.println(e);  
 }  
  
 }  
 return builder + "";  
}



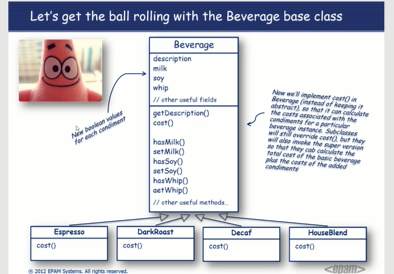
**А теперь рассмотрим упрощенное закрытие ресурсов при обработке исключений, просто переносим создание объекта в блок try (можно указать несколько через запятую), теперь будут закрываться автоматически выходя за пределы области видимости try**

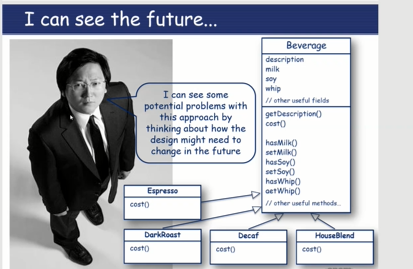
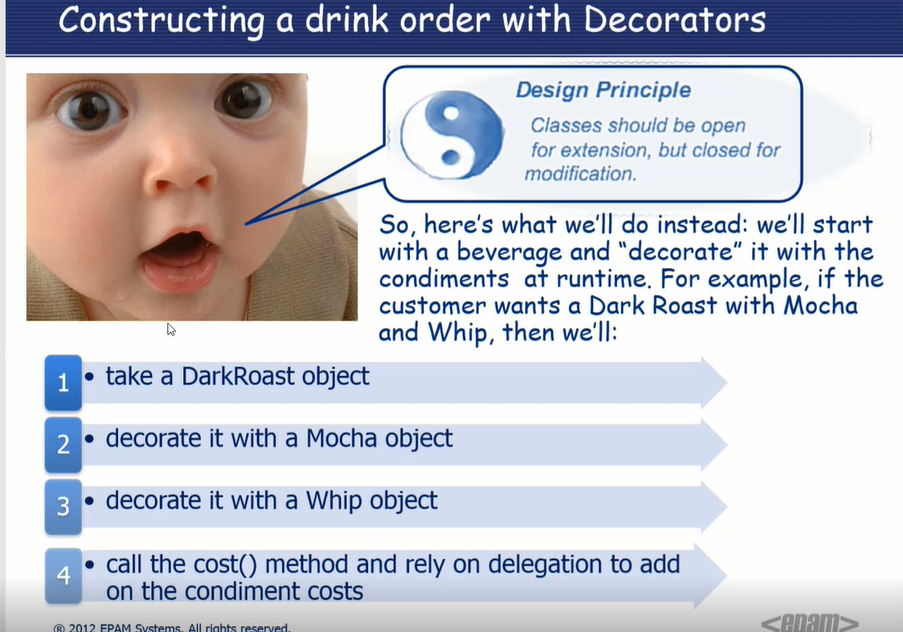
public static String read(String FileName) {  
 StringBuilder builder = new StringBuilder();  
  
 try (Reader stream = new FileReader(FileName)) {  
 if (stream.ready()) {  
 int temp;  
 while ((temp = stream.read()) != -1) {  
 builder.append(temp).append(" ");  
 }  
 }  
 } catch (IOException e) {  
 System.*err*.println(e);  
 }  
  
 return builder + "";  
}

**Поддерживается только с 7й версии Java, только для объектов имплементирующих AutoCloseable.**

Вернемся к рассмотрению базовой концепции потоков ввода-вывода. Чтобы понять эту структуру, необходимо рассмотреть шаблон Декоратор.Стоит помнить, что шаблоны стоит использовать только по необходимости, если есть вариант проще – рассматривать его. Декоратор рассмотрим на примере кофейни (из HeadFirst). Задача декоратора – динамически нарастить функционал, (Стратегии -заменить). Предположим мы открываем кафе, для начала мы будем предлагать посетителям только кофе разных сортов

Через какое-то время нам понадобилось расширить функционал, тк посетители хотят кофе с молоком, сахаром и тд🡪

Конечно можно внести эти характеристики в свойства объекта базового класса, но во- первых неудолетворение OCP каждое новое свойство потребует изменения существующего кода, во- вторых не учитывается возможность 2го-3го применения ингридиента, в третьих если мы захотим использовать эти добавки для чая… придется опять прописывать их в новом базовом классе.

Здесь и пригодится шаблон Декоратор. Предположим у нас есть объект с базовыми характеристиками и мы хотим их чем-то дополнить – декорировать.

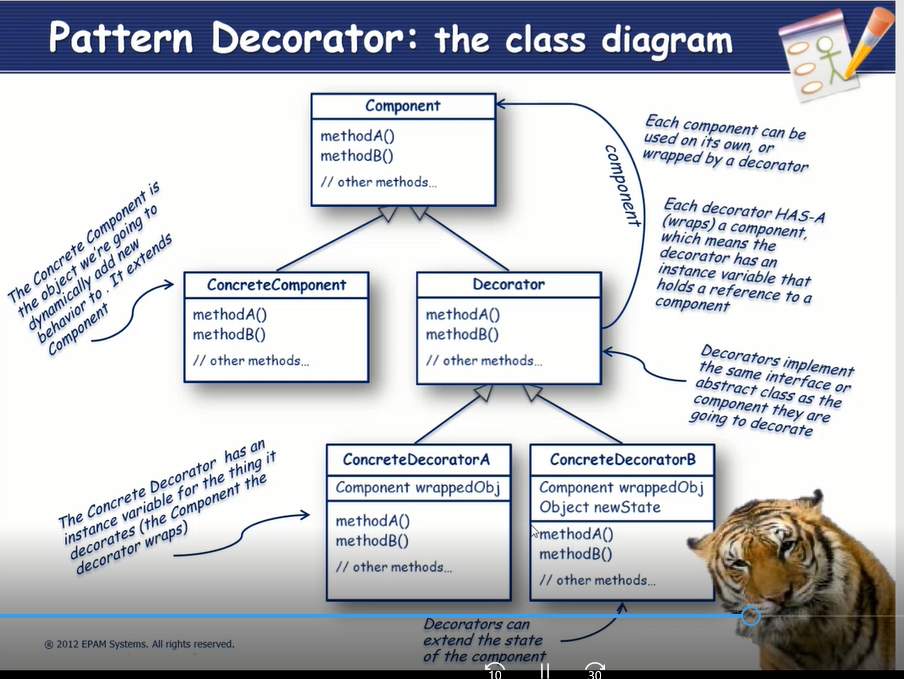
С точки зрения математической модели, это будет представлено

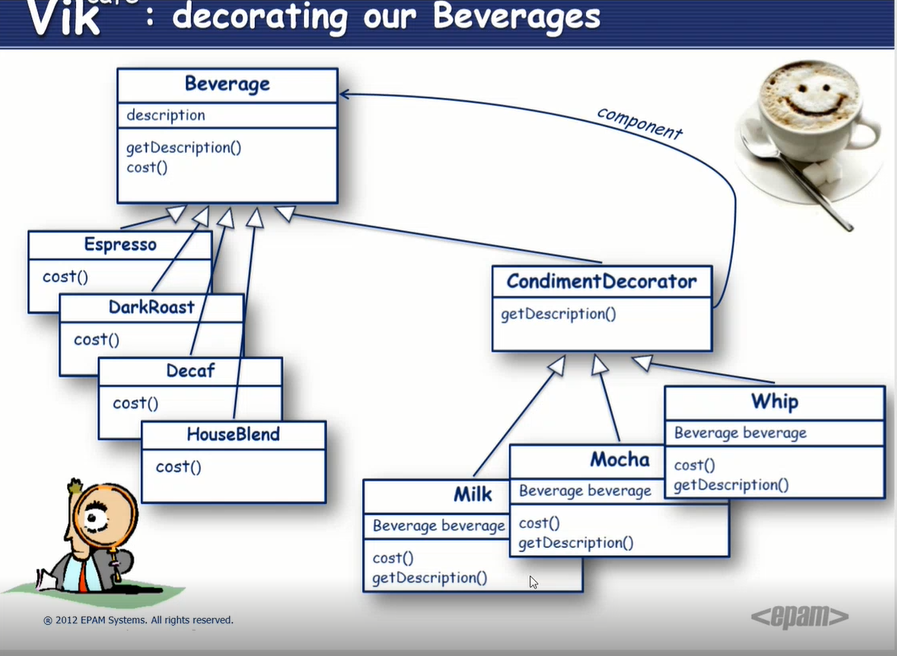
DarkCoffe

Объект Декоратор, динамически обрамляет наш объект Кофе, для пользователя объект остается прежним, с точки зрения связи это будет композиция (обертка).Декоратор не является самостоятельным объектом, только обрамляет основной🡪убираем дефолтный конструктор и оставляем только тот который принимает основной объект. Все декораторы дб того же типа, что и основной объект для сохранения абстракции. Промежуточный базовый класс Декоратор абстрактен, а его наследники уже сначала делегируют выполнение объекту ссылку на который приняли, а потом дополняют его.

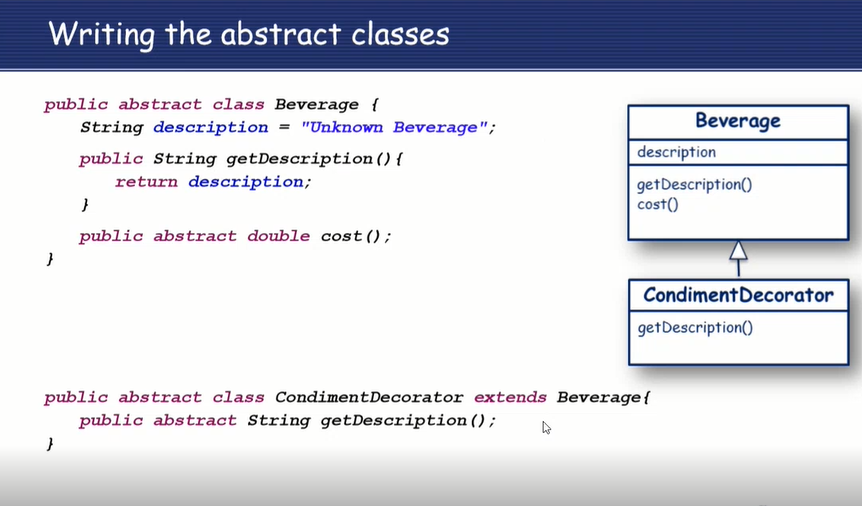
сost()->+$0.2

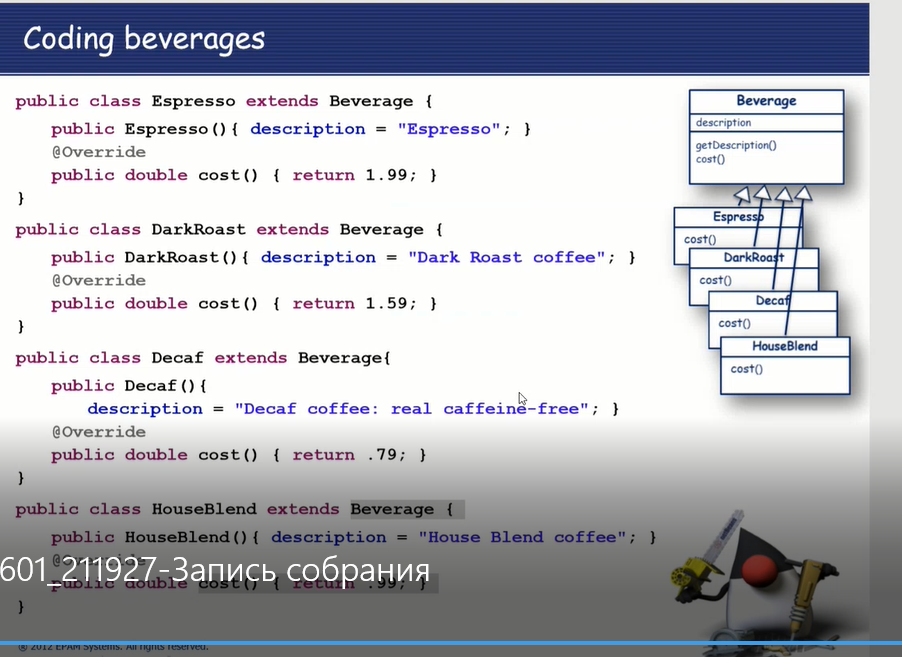
сost()->$1.2

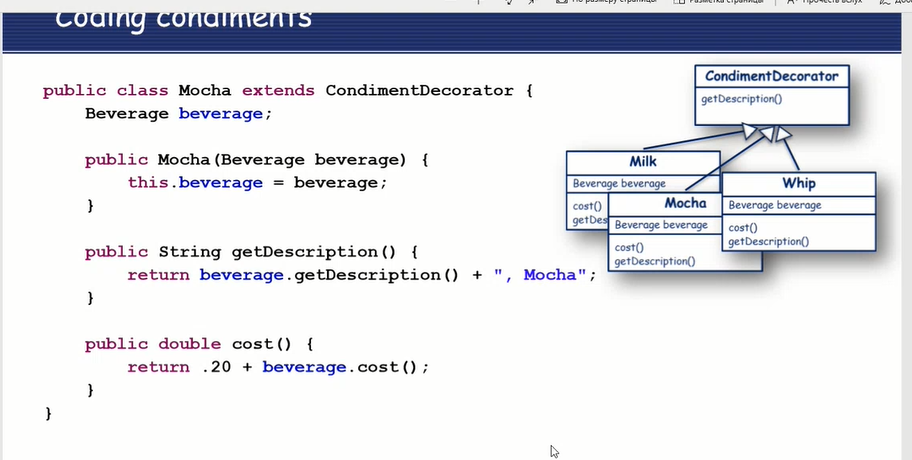




Перейдем к реализации, класс Декоратор наследуется от базового метод cost() просто передаем наследникам, а getDescription переопределяем как абстрактный

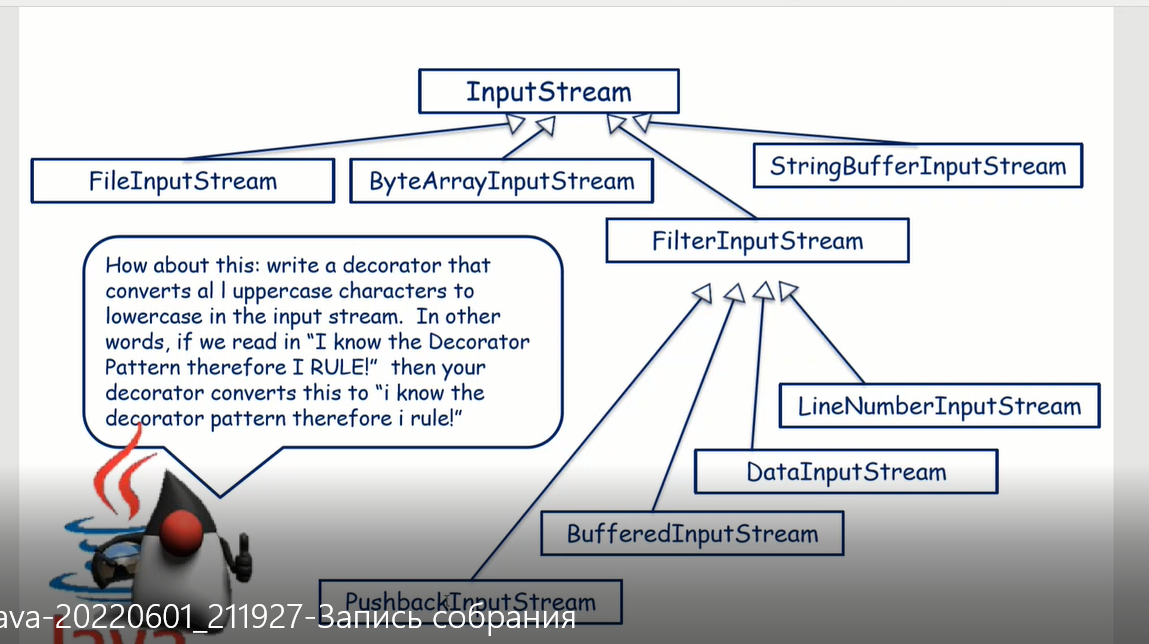


В прямых наследниках реализуем инициализацию описания в дефолтном конструкторе и реализуем метод cost()

В конкретном декораторе создаем поле хранящее ссылку на объект напитка, вместо дефолтного конструктора реализуем конструктор принимающий объект, которым при создании декоратора инициализируем поле-ссылку, а в остальных методах делегируем выполнение переданному нам объекту, дополняем результат.



Теперь, зная шаблон Декоратор мы сможем лучше понять иерархию потоков ввода вывода



Теперь посмотрим, как проще реализовать данную систему без использования декоратора на базе композиции, создаем иерархию декораторов, а сам напиток будет контейнером для добавок – декораторов, базового класса Добавок, реализуем метод add для добавления в контейнер компонентов