1. **Working with Abstraction**

**Абстракция** – действието на описването на само това, което ни трябва, както и дефинирането на малки методи, които извикват други

**element.ordinal()** – показва поредния номер на елемента в изброения списък (енъм)

**element.toString()** – принтира елемента от изброения списък

CardSuit[] cardSuit = CardSuit.values(); // - връща масив с име cardSuit, съдържащ елементите, които са в енъма.

**Refactoring**

* Ако искаме да променим дадена променлива, маркираме я, след това натискаме Shift+F6 и записваме новата стойност. По този начин на всички места стойността ще бъде актуализирана.

**Static Keyword in Java**

* Можем да имаме статични: вложени класове, променливи, методи и блокове код
* Членове са на класа, а не на обекта. Тоест не ни трябва инстанциран обект, за да ги извикаме, а направо чрез класа.
* **Статична променлива** – споделена е от всички обекти в даден клас. Например, името на фирмата, в която работят всички служители, е статична променлива. Най-често статичните променливи се използват за глобален counter.
* **Статичен метод** – метод, който може да се извика без инстанция от определен обект. Например – когато трябва да направим бърза сметка на нещо (например изчисляване на число на трета степен).
* **Статичен блок** – изпълнява се веднага със зареждането на класа

**Пакетите в Java**

* Използват се за подреждане на кода – един вид модули стоят на едно място
* Пакет, който ние създаваме: **User-defined Packages**
* Пакет, който Java е създала: **Built-in Packages** (Java.Util пакета)

Предимствата на енкапсулацията в Java включват:

Скриване на вътрешната имплементация на класа, което води до по-добра абстракция и модуларност на кода.

По-лесно поддържане и промяна на вътрешната имплементация на класа, без да се засяга външния код, който използва класа.

Възможност за контролиране на достъпа до променливите на класа, което подобрява сигурността и намалява вероятността от грешки.

1. **Encapsulation**

* Крием имплементация от външния свят; използваме access modify-ери; намаляваме сложността – не даваме на хората да ни чупят класовете; гарантираме, че ако ще има промени в класа, то те ще се случват в самия клас

**Енкапсулация** – криенето на информацията в един клас. Например private полетата. Ако се притесняваме, че някой може да въведе грешно име или т.н…Крием вътрешните детайли с private и избираме поетапно кое да разкриваме с public. Така ставаме по-гъвкави и можем да разширяваме кода.

* Лаптопът има различни характеристики, но това, с което ние имаме директен достъп, са клавиатурата и мишката.

Картина, която съдържа текст, екранна снимка, софтуер, Операционна система

Описанието е генерирано автоматично

* Полетата са private
* Методите са public

**Accessories** (getters), **Mutators** (setters) should be public!

**Keyword “this”**

* This сочи към конкретната инстанция на обекта
* Може да се използва и в конструктора
* Когато има припокриване на имената на променливи, задължително се използва **this**, за да се знае, че по този начин променяме самата променлива на обекта.
* Може да се използва за извикване на методи в класа от самия клас.
* Извикването на конструктор в друг конструктор става винаги на първия ред, защото така се създава новия обект, а със следващите промени – той се актуализира.
* Ако методът е статичен, няма как да използваме **this**, защото няма обект, върху който да го приложим, а има само клас.
* **Constructor** **chaining** – извикването на конструктор в конструктор

Картина, която съдържа текст, екранна снимка, софтуер, Мултимедиен софтуер

Описанието е генерирано автоматично

**Private Access Modifier**

* Достъпването на полето извън класа се случва с get и set методите.
* Възможно най-ограничено трябва да правим класовете. Тоест започваме от private, default, protected и чак тогава public.
* Със **super()** извикваме конструктори, полета и методи от родителския клас.

**Package private / Default**

* Променливата е private и се вижда само в пакета, в който е

Със **@Override** редактираме метод от родителския клас в дъщерния с цел да се използват нови функционалности.

**Validation**

* Валидацията трябва да се осъществява на две нива – UI level & Back-end level, за да сме сигурни, че няма да допуснем невалидни данни.
* Data validation happens in setters

*private void setSalary(double salary) {*

*if (salary < 460) {*

*throw new IllegalArgumentException("Message");*

*}*

*this.salary = salary;*

*}*

* Private setter – когато правим валидация за полето, която искаме да приложим в конструктора.

**Mutable and Immutable Objects**

* **Mutable** – обект, който може да се променя
* **Immutable** – обекти, които няма как да променяме информацията им. Пример за такива обекти е String. Всеки път, когато използваме метод, който си мислим, че променя такъв обект, то всъщност се създава нов в паметта на Heap-а, а не се променя.

Картина, която съдържа текст, екранна снимка, диаграма, линия

Описанието е генерирано автоматично

* Когато четем String от конзолата, променливата не влиза в String pool-a, а когато я инициализираме директно – влиза.
* For securing our collection we can return **Collections.unmodifiableList()**

**Final**

* С **final** правим константи – задаваме стойност само веднъж.

*public final static String NAME\_OF\_MY\_COUNTRY = “Bulgaria”;*

* Може да се използва пред методи и класове
* **Final class** не може да бъде наследен
* **Final method** не може да бъде пренаписан
* Можем да въведем final променлива чрез скенера **само** в конструктора.

1. **Наследяване / Inheritance**

Наследяване – способността на един клас да взима поведение и състояние от друг вече съществуващ клас.

Със **super()** използваме конструктор с променливи от базовия клас. В конструктора на наследяващия клас задължително първо трябва да използваме super(), за да извикаме конструктора от базовия клас.

* Конструкторите не се наследяват

**Композиция** – един клас съдържа обекти от други класове като част от своето състояние. Когато обектът-контейнер е унищожен, съдържащите се в него обекти също се унищожават.

**Делегация** – отношение, където един клас предава отговорностите си за определени операции на друг клас. Когато обектът-делегат е унищожен, делегираният обект не се засяга. Делегацията се постига чрез композиция на обект от друг клас и извикване на неговите методи при нужда.

В резюме, композицията е силно свързано отношение, където един клас съдържа други класове като част от себе си, докато делегацията е по-слабо свързано отношение, където един клас предава определени отговорности на друг клас, без да ги съдържа напълно.

За интервю

Картина, която съдържа текст, екранна снимка, софтуер, дисплей

Описанието е генерирано автоматично