In cadrul pachetului **`investment.portofolio`** am creat :

- pachetele: **`algorithm`**, **`asset`**, **`comparators`**, **`item`**
- clasele *:* **AssetManager, Building, Jewel, Vehicle**

In pachetul **`item`** am creat clasa abstracta ***Item*** in care avem campurile **name** si **price**, de asemenea avem constructor, setter, getteri pentru aceste campuri, am suprascris toString() -- pt afisare frumoasa si equals() -- pt ca nu avem voie sa avem 2 item-uri de acelasi tip. Cand suprascriem equals() implicit tb suprascris si hashCode() – totul se face automat din IDE.

Clasele **Building**, **Jewel** si **Vehicle** sunt de tipul ***Item***, deci aceleasi proprietati ca si clasa din care mostenesc.

De exemplu, pentru Jewel care, asa cum scrie in cerinta, nu reprezinta si un asset, voi avea doar:

```
public Jewel(String name, int price) {

    super(name, price);
}

@Override
public String toString(){
    return "Jewel{" + super.toString() + "}";
}
```

Pentru obiecte de tipul Building si Vehicle se specifica faptul ca reprezinta si asset-uri, de aceea pe langa constructor si toString() mai avem 2 metode deoarece aceste clase implementeaza si interfata ***Asset***.

Astfel ni s-a cerut sa cream interfata ***Asset*** care are 2 metode:

- o metoda abstracta: `int computeProfit()` – calculeaza profitul obtinut
- o metoda default : `default double riskFactor()` – care ne spune care este factorul de risc financiar la care ne expunem (in mod implicit returneaza 0 – nici un risc)

In clasele **Building** si **Vehicle** am suprascris aceste metode deoarece aceste clase implementeaza interfata ***Asset***. Si cum in cerinta ni se spunea ca riscul financiar poate fi o valoare cuprinsa intre 0 (fara risc) si 1 (foarte riscant) am considerat ca la achizitionarea unui obiect de tipul **Buiding** sa avem un factor de risc de 0.3 (mai putin riscant) iar la achizitionarea unui obiect de tipul **Vehicle** avem un factor de risc de 0.7 (mai riscant – accidente ☺).

Pentru calcularea profitului am gandit urmatoarea formula:

```
Profit  = Castig – Castig * Risc =  Castig * (1 – Risc)
```

Pentru **Building** ni se spune ca castigul initial (initialProfit) = area/price, iar pentru **Vehicle** = performance/price.

Si cum trebuie sa tinem cont si de riscul pe care ni-l asumam (la punctual Optional), in final metodele arata asa (aici arat doar pentru **Building**):

```
@Override
public int computeProfit() {
    int initialProfit = area / getPrice();
    double risk = 1 - riskFactor();

    return (int) (initialProfit * risk);
}
@Override
public double riskFactor(){
    return 0.3;
}
```

Clasa **AssetManager** are urmatoarele campuri:

- un comparator de Item-uri – ni s-a cerut ca item-urile sa fie sortate dupa nume
- un comparator de Asset-uri – ni s-a cerut ca asset-urile sa fie sortate dupa profit
- un set de item-uri : Set<Item> items (am ales Set deoarece nu vrem sa avem doar elemente distincte)

si are implementate urmatoarele metode care ni se cer:

- constructor
- `getItems()` – intoarce o lista de item-uri (Building, Vehicle si Jewel)
- `getAssets()` – intoarce o lista de item-uri care sunt si asset-uri (doar obiecte de tipul Building si Vehicle)
- `createPortofolio (<algoritm><maxValue>)` – care ne intoarce un portofoliu de asset-uri selectate conform unui algoritm (avem unul greedy si unul random) pentru care valoarea asset-urilor nu trebuie sa depaseasca maxValue

```
public Portofolio createPortofolio(Algorithm algorithm, int maxValue) {
    List <Asset> assets = getAssets();
    algorithm.orderAssetsAccordingToStrategy(assets);

    Portofolio portofolio = new Portofolio();
    for (Asset asset : assets) {
        Item item = (Item) asset;
        if (item.getPrice() <= maxValue) {
            portofolio.addAsset(asset);
            maxValue = maxValue - item.getPrice();
        }
    }

    return portofolio;
}
```

In pachetul **`algorithm`** avem:

- interfata ***Algorithm*** – cu o metoda abstracta:

```
public interface Algorithm {
    void orderAssetsAccordingToStrategy(List <Asset> assets);
}
```

- clasa **GreedyAlgorithm** – care implementeaza ***Algorithm*** – deoarece este un algoritm greedy am gandit sa sortam lista de asset-uri conform profitului, dupa care am inversat-o deoarece vrem sa alegem intai profitul cel mai mare, apoi urmatorul cel mai mare …. – de aceea am avut nevoie si de un comparator de asset-uri pe baza de profit (***ASSET_COMPARATOR***);

```
@Override
public void orderAssetsAccordingToStrategy(List <Asset> assets) {
    assets.sort(ASSET_COMPARATOR);
    Collections.reverse(assets);
}
```

- clasa **RandomAlgorithm** – care implementeaza ***Algorithm*** – aici pur si simplu am permutat lista de asset-uri cu ajutorul Collections.shuffle(<list>)

```
@Override
public void orderAssetsAccordingToStrategy(List <Asset> assets) {
    Collections.shuffle(assets);
}
```

Din documentatia pentru shuffle : "*Randomly permutes the specified list using a default source of randomness. All permutations occur with approximately equal likelihood.*"

- Clasa **Portofolio** care reprezinta o solutie pentru problema noastra si care contine lista de asset-uri pe care ne-o returneaza metoda createPortofolio (<…><…>) explicata mai sus
  - Pentru afisare frumoasa am suprascris toString()
  - Mai avem 2 constructori: unul cu argumente si unul fara

In pachetul **`comparators`** avem:

- Clasa **AssetComparator** care implementeaza Comparator<Asset> – care compara 2 obiecte de tip Asset pe baza de profit

```
public class AssetComparator implements Comparator<Asset> {
    @Override
    public int compare(Asset o1, Asset o2) {
        return o1.computeProfit() - o2.computeProfit();
    }
}
```

- Clasa **ItemComparator** care implementeaza Comparator<Item> -- care compara 2 obiecte in functie de nume (in cerinta se cere : "Display all the items sorted by their names.")

```
public class ItemComparator implements Comparator <Item> {
    @Override
    public int compare(Item o1, Item o2) {
        return o1.getName().compareTo(o2.getName());
    }
}
```

Din documentatia pentru Comparator vedem ca acesta reprezinta o interfata functionala:

```
@param <T> the type of objects that may be compared by this comparator
@FunctionalInterface
public interface Comparator<T>
```