Interfete

Ce este o interfata?

In mod foarte simplist, interfata este un tip, cu anumite restrictii. Aceste restrictii sunt legate de ceea ce poate sa contina.
Interfetele au rolul de a stabili un contract intre doua obiecte, anume, Obiectul A poate sa apeleze metoda X din obiectul B, daca ii paseaza anumite argumente si va primi ca raspuns un obiect de tip C. Deci putem observa ca interfata reprezinta promisiunea existentei unei metode cu o anumita semnatura.
Pentru a intelege mai bine interfetele voi porni de la ceea ce erau interfetele inainte de Java 8.
In primul rand, interfetele nu puteau contine decat metode publice. Acest lucru era foarte important pentru ca interfetele de fapt nu contineau implementare, deci erau implicit abstracte si atunci aceste metode trebuiau rescrise de subclase. Desigur, puteam avea functii statice care se puteau executa in orice conditii, dar nu si metode de instanta. De asemenea, interfetele nu pot avea decat campuri publice statice. De fapt, aceste reguli sunt atat de importante incat modificatorul implicit pentru campuri si metode este public si poate fi chiar omis!
Interfetele sunt modul Java de a se adapta la lipsa mostenirii multiple.
Sa luam urmatorul exemplu:
Avem clasa Animal, si subclasele Caine si Pisica. Ierarhia este evidenta. Totusi, din punctul de vedere al unui pet shop Caine si Pisica sunt de fapt “Produs” pentru ca ei vand acele animale. In acest caz, Caine si Pisica ar trebui sa mosteneasca atat Animal cat si Produs, ceea ce nu se poate.

Atunci, solutia este sa identificam functionalitatea ce trebuie mostenita. De exemplu, un Produs trebuie sa poata fi vandut, si sa poata sa ofere un pret (stabilit in functie de caracteristicile animalului sa spunem). Deci Produs nu contine campuri, in schimb Animal da.
In acest caz putem sa definim tipul Produs ca o interfata cu cele doua metode, vinde() si getPret().
Caine si Pisica pot acum sa mosteneasca atat metodele din tipul Produs cat si metodele/campurile din Animal.
Din moment ce atat Animal cat si Produs sunt tipuri, putem sa declaram variabile de acel tip, putem sa facem type cast etc. Din acest punct de vedere, nu exista diferente intre clasa si interfata.
Ce observam? Cand este vorba de mostenire, clasele sunt folosite pentru a mosteni 3 lucruri: contract (semnaturi de metode, care pot fi si abstracte), implementare (metode care au corp) si campuri (date). Interfetele (pana la java 8) nu puteau oferi decat contract (semnaturi de metode).
Sa ne intoarcem la cazul nostru de mai sus. Corpul metodei getPret difera in mod evident de la un animal la altul (in functie de rasa, eventual culoarea parului, lungimea botului, a cozii, cati soareci a prins sau ce viteza de alergare are, etc). Deci este de asteptat ca atat Caine cat si Pisica sa aiba propria implementare. Insa, vinde() ar putea o implementare comuna. Tot ce trebuie sa facem este sa modificam statistica pentru ziua respectiva (sa modificam incasarile). Deci daca am implementa metoda in Caine si in Pisica, am avea cod duplicat. Din aceste considerente, in Java 8 au fost adaugate implementarile “default”. Practic, interfetele pot avea acum si metode cu corp. Deci mostenirea, pentru interfete, poate contine si cod (implementare). Sa spunem acum ca o interfata are mai multe astfel de metode publice si cu implementare. Sa spunem ca toate aceste metode folosesc un fragment de cod comun, sa spunem o validare de date. Aceasta bucata de cod, ar trebui sa fie duplicata in fiecare din metodele publice cu implementare default. Evident, nu e prea placut. Atunci in Java 9 li s-a permis interfetelor sa aiba inclusiv metode private, tocmai in scopul de a refolosi codul.
Dupa cum vedeti, distanta intre clase si interfete s-a micsorat destul de mult odata cu Java 8 si Java 9, dar doua lucruri fundamental raman:

1. interfetele nu pot avea campuri ne-statice+final
2. interfetele sunt abstracte si nu pot fi instantiate.

Utilizari ale interfetelor

Pe langa exemplul (clasic) de mai sus, interfetele mai sunt utilizate si pentru a pasa functiilor “cod” ca parametru.
Ce inseamna cod? Atunci cand ne gandim la cod, la implementare, ne gandim intotdeauna la o metoda. Deci vrem sa trimitem ca parametru unei metode (A) o alta metoda (B) (atentie! nu rezultatul executiei unei alte metode!). In acest caz, definim o interfata care contine metoda respectiva cu semnatura pe care o dorim.
Apoi, putem sa trimitem ca parametru metodei A un obiect de tipul interfetei nou definite. Apoi apelam metoda cu un obiect de acest tip, care are deci o implementare corespunzatoare pentru metoda respectiva.

Astfel putem sa construim un obiect si sa ii definim reactii variabile la runtime pentru diverse situatii.

Enumerarile

Este foarte important sa ingelegem ca un enum este de fapt o clasa care extinde clasa Enum. Din aceasta afirmatie, deducem multe din regulile legate de un enum.
Un enum poate extinde o alta clasa?
Un enum poate implementa o interfata?

Valorile unui enum sunt de fapt instante ale acelui tip, construite prin apelarea constructorului corespunzator. Un enum NU poate fi instantiat!

Interfetele Comparable si Comparator

Putem vedea informatii complete despre ele la adresele:
https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html
https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html

Pe scurt, Comparator contine o metoda int compare(Object a, Object b), iar Comparable o metoda int compareTo(Object other).

Haideti sa construim o clasa Student, cu nume, prenume si nota (de tip float).

Am vrea sa putem sa sortam studentii. Pentru a putea sa ii sortam, fie definim o relatie de ordine intre stundeti (implementand Comparable) astfel incat orice student poate fi comparat cu orice alt student.
Haideti sa folosim Arrays.sort() ca sa sortam un vector de studenti. Implementam ordinea intre studentii in felul urmator: studentii sunt sortati dupa nume. Daca numele e identic, atunci dupa prenume. Daca si prenumele e identic, dupa nota.
Acum sa spunem ca vrem sa sortam stundetii dupa un alt criteriu, sa spunem dupa nota. Totusi nu vrem sa modificam relatia definita mai sus. Putem atunci sa furnizam lui Arrays.sort() pe langa vector si o implementare a lui Comparator, care sa defineasca practic relatia de ordine.

Atat Comparator (care compara a cu b) cat si Comparable (care compara this cu other) functioneaza pe aceeasi idee: compara cele doua obiecte, dupa logica definita in interiorul metodei compare/compareTo si returneaza un int care este <0 daca a<b, =0 daca a==b sau >0 daca a>b.

instanceof

In java putem sa verificam daca un obiect are un anumit tip (la runtime) folosind instanceof.
De exemplu
`"test" instanceof String`
este o expresie care se evalueaza la true.

Tema 1

Definiti interfetele pentru exercitiile din laboratoarele 4 si 5. DatabaseElement, MyMatcher, Splitter.

Tema 2

Pentru structura de date din laboratorul 4 (Persoana, Student, Profesor) definiti urmatoarea ordine intre obiecte:
Persoanele se compara dupa Nume, Prenume.
Daca doua persoane au acelasi Nume, Prenume, profesorul este mai mic decat studentul.
Daca doi profesori au acelasi nume, sortarea se face in ordine alfabetica dupa disciplina.
Daca doi studenti au acelasi nume, sortarea se face dupa an.

Tema 3

Sortati un vector de Persoana (de la tema 3) in ordinea inversa, folosind Arrays.sort().

Tema 4

Implementati structura de clase si interfete pentru a modela unitatile dintr-un joc.
Avem urmatoarele unitati:

1. Transportor blindat (viata, energie, spatiu de stocare). Poate merge pe pamant, si poate transporta doar infanterie sau spioni. Metode: move(int x, int y), load(???), unload(???), takeDamage(int damage).
2. Buldozer (viata, energie, spatiu de stocare). Metode move(x,y), buldoze(), takeDamage(), loadMaterials()
3. Builder(viata, energie, spatiu de stocare). move(), build(), takeDamage(), loadMaterials()
4. Reparator (viata, energie, spatiu de stocare) move(), repair(), takeDamage(), loadMaterials()
5. Radar mobil (viata, energie) move() takeDamage()
6. Tanc (viata, energie, munitie) move(),attack(), takeDamage(), canShoot(otherUnit) true/false in functie de tipul celeilalte unitati.
7. Lansator de rachete(viata, energie, munitie) move(), attack(), takeDamage(), canShoot(otherUnit)

8. Infanterist (viata, energie, munitie) move(), attack(), takeDamage()
9. Spion (viata, energie, munitie, experienta) move(), attack(), takeDamage(), hite(), sabotage().
10. Avion de transport (viata, energie, spatiu de stocare). Poate sa transporte orice unitate terestra (fara barci sau avioane). move(), land(), takeoff(), takeDamage(), load(), unload().
11. Avion de spionaj (viata, energie) move(), takeDamage()
12. Bombardier(viata, energie, munitie) move, attack, takeDamage, land, takeoff
13. Feribot (viata, energie, spatiu de stocare). Poate transporta orice untiate terestra (fara barci sau avioane). move, takeDamage, load, unload.
14. Submarin (viata, energie, munitie). move, takeDamage, attack. Poate ataca doar barci.

Trebuie sa construiti ierarhia in asa fel incat:

1. Sa nu aveti cod duplicat
2. Sa nu aveti obiecte in care exista campuri nefolosite.
3. Sa construiti implementari goale pentru diverse metode (functii cu corp vid).

Desigur, e doar un exercitiu de concepere a unei structuri de clase, nu ma astept sa implementati nimic. In loc de implementare vreau sa vad un simplu comentariu. De exemplu, pentru tanc

```
public void move(int x, int y)
{
// check if x,y on ground and move
}
```

Acolo unde trebuie verificat daca o unitate poate fi atacata sau incarcata in transportor vreau sa vad insa

```
public void load(Other otherUnit)
{
if (otherUnit instance of Infantry)
{
/// load unit.
}
else
{
/// error!
}
}
```

care este implementarea pentru metoda load a transportorului blindat.
Exemplul este foarte complex si va trebui sa folositi toate relatiile pe care le cunoasteti (mosternire, agregare, compunere).
Exemplu este extras (varianta simplificata) dintr-un joc care a fost lansat acum mai bine de 20 ani. Daca vreti o provocare puteti sa incercati sa implementati toate obiectele din joc (inclusiv cladiri, care au si ele mai multe subtipuri, ca si unitatile).
Folositi linkul acesta:

https://www.maxr.org/docs.php?id=31
Aveti acolo Vehicles si Buildings.
Siteul respectiv nu imi apartine, l-am gasit intamplator, si nu garantez siguranta eventualelor lucruri pe care le descarcati de acolo. Nu interactionati cu siteul decat ca sa vedeti lista de unitati de diferite tipuri. Nu introduceti date personale pe acel site!

Succes!