Projet à rendre - Compte bancaire v2.0

1 Projet à rendre

Ce projet est un projet à faire de manière **individuelle** et à rendre à votre encadrant via le module *Bases de la Prog* sur arche. Il donnera lieu à une évaluation qui comptera à hauteur de 15% dans la note finale du module.

Vous devrez rendre **IMPERATIVEMENT** pour **le mardi 15/11/2022** sur arche :

- des sources **commentées** qui **compilent** : tout problème dans la compilation ou l'absence de commentaire entrainera des points en moins ;
- une version déjà compilée de votre programme ;
- une classe `CarteBlocable` avec des commentaires associés (cf section 8) ;
- des classes de test **complétées** et intégralement **commentées** (cf fin de l'énoncé) ;
- un répertoire contenant la **javadoc** générée à partir de votre projet.

ATTENTION :

- Veuillez respecter **IMPERATIVEMENT** les noms des classes et méthodes que l'on vous donne (majuscule comprise, sans accent et sans faute d'orthographe - cf conventions en annexe du polycopié). Respectez aussi l'**ordre** des paramètres dans les fonctions ou méthodes.
- Tous les attributs doivent être déclarés en **private**, les méthodes sont déclarées en **public** sauf précision dans l'énoncé.
- Les méthodes ne doivent contenir aucun `System.out.println` sous peine de ne pas pouvoir être corrigées facilement. Des points pourront être retirés si cette consigne n'est pas respectée.
- Votre projet est à faire **en individuel uniquement**, tous les projets seront testés par une application capable de détecter automatiquement les tentatives de fraude et nous sévirons si besoin (comme cela a déjà été fait par le passé).

2 Présentation du projet

Le projet va avoir pour objectif de représenter (très simplement) des personnes faisant des achats avec des cartes bancaires. De manière plus précise, il s'agit de programmer plusieurs classes pour modéliser des personnes et leurs comptes :

- la classe `CarteB` a pour objectif de représenter une carte avec son solde, son découvert autorisé et son code secret ;
- la classe `Personne` a pour objectif de représenter une personne caractérisée par un nom et la carte bancaire qu'il possède.

3 Structuration du projet

Le fichier `.zip` fourni sur arche vous donne la structure de sous-répertoires à respecter pour votre projet :

- un répertoire principal `gr_Z_nom` dont le nom est à modifier
  - `Z` doit désigner votre numéro de groupe
  - `nom` doit désigner votre nom
- ce répertoire contient plusieurs sous-répertoires
  - un répertoire `src` qui doit contenir les fichiers `.java` ;
  - un répertoire `javadoc` qui contiendra le résultat de la javadoc.

4 La classe `CarteB`

Question 1

Écrire la classe `CarteB` avec les indications suivantes.

4.1 Attributs

Le classe `CarteB` a pour objectif de représenter une carte bancaire qu'une personne pourra utiliser pour effectuer un paiement. Une carte bancaire est caractérisée par plusieurs attributs privés :

- un attribut réel `solde` représentant le montant sur la carte ;
- un attribut réel positif `découvert` désignant le découvert autorisé (par exemple un découvert autorisé de 100 indique que la carte peut avoir un solde pouvant descendre jusque -100) ;
- un `String codeCarte` correspondant au code secret de la carte (on utilisera un simple mot en toutes lettres comme `"dromadaire"` ou ”`"pokemon"` pour ne pas avoir à utiliser de chiffres).

Bien entendu, c'est une vision très simpliste. L'objectif n'est pas de simuler des cartes bancaires réelles mais de vérifier que vous savez programmer des classes java simples.

4.2 Constructeurs

La classe `CarteB` possède deux constructeurs :

- un constructeur avec un paramètre `code` de type `String` construisant une carte bancaire par défaut (avec un solde de 0 et un découvert de 100) dont le code secret est passé en paramètre.
- un constructeur avec trois paramètres : un paramètre réel `montant` correspondant au solde initial du compte, un paramètre réel `decouv` correspondant au découvert autorisé et un paramètre de type `String` correspondant au code secret de la carte. Les paramètres réels doivent être positifs (sinon, on attribuera des valeurs par défaut de 0).

4.3 Méthodes Accesseurs

Écrire les méthodes `getSolde` et `getDecouvert`.

Attention ! Sauf indication contraire, ces accesseurs ne doivent servir que dans vos méthodes de test pour vérifier les bonnes valeurs des attributs.

4.4 Autres Méthodes

- `etreCodeCorrect` : Écrire la méthode `etreCodeCorrect` qui prend en paramètre une chaine de caractères correspondant au code tapé pour utiliser la carte et qui renvoie vrai si et seulement si ce code est bien le code secret de la carte. Cette méthode utilisera la méthode `equals` de la classe `String` qui permet de tester l'égalité entre deux chaines de caractères.

- `deposer` : Écrire la méthode `deposer` qui prend en paramètre un réel `montant` et ajoute ce `montant` au solde du compte. Lorsque le montant passé en paramètre est négatif, rien ne doit se passer.

- `depenser` : Écrire la méthode `depenser` qui prend en paramètre un double `prix` et un `String code` et qui retourne un `boolean`. Cette méthode doit retirer le prix au solde du compte. L'opération ne peut s'effectuer que si le code passé en paramètre est le bon code secret et si le solde après retrait du prix est bien supérieur au découvert autorisé. Si l'opération s'est correctement passée, cette méthode retourne vrai sinon elle doit retourner faux.

- `toString` : Écrire la méthode `toString` qui retourne la chaine `"carteB: "` suivie du solde du compte et du découvert autorisé `decouvert` sous la forme suivante (espaces compris) :
  `"carteB: `*`solde / -decouvert`*`"`
  où *solde* et *decouvert* représentent les valeurs de ces attributs.
  Ainsi, la chaine retournée pour une carte avec un solde de 100 et possédant un découvert autorisé de 500 doit être `"carteB: 100.0 / -500.0"`

4.5 Tests

Question 2

A l'aide des indications de la section 6, écrire la classe de test `TestCarteB` associée à la classe `CarteB`. N'oubliez pas de tester toutes les situations possibles (un test différent par situation).

5 La classe `Personne`

Question 3

Écrire la classe `Personne` avec les indications suivantes.

5.1 Attributs

Le classe `Personne` a pour objectif de représenter une personne pouvant posséder une carte bancaire. Une personne est caractérisée par deux attributs privés :

- un attribut `nom` de type `String` correspondant au nom de la personne ;
- un attribut `carte` de type `CarteB` correspondant à la carte bancaire possédée par la personne.

5.2 Constructeur

La classe `Personne` possède un constructeur qui prend en entrée un seul paramètre `pNom` de type String correspondant au nom de la personne. Ce constructeur construit une personne dont le nom correspond à `pNom` et qui ne possède pas de carte bancaire (attribut `carte` à `null`).

5.3 Méthodes accesseurs

Écrire les méthodes `getCarte` et `getNom`.

Attention ! Ces accesseurs ne doivent servir que dans vos méthodes de test pour vérifier les bonnes valeurs des attributs. **Ils ne doivent pas être utilisés dans les classes `Personne` et `CarteB`.**

5.4 Autres méthodes

- `prendreCarte` : Écrire la méthode `prendreCarte` qui prend un objet de type `carteB` en paramètre et met à jour la carte de la personne avec la carte passée en paramètre.

- `donnerCarte` : Écrire la méthode `donnerCarte` qui prend un objet `p` de type `Personne` en paramètre et retourne un `boolean`. Cette méthode donne à cette personne `p` la carte que possède la personne `this` sur laquelle la méthode est appellée. Ainsi, l'appel `p1.donnerCarte(p2)` consiste à donner à `p2` la carte que possède `p1`. L'opération ne peut s'exécuter que si la personne `p` passée en paramètre existe bien, et si elle ne possède pas déjà une carte. A l'issue de l'opération, la personne sur laquelle la méthode est appelée ne possède plus de carte (puisqu'elle a été donnée à `p`). La méthode retourne `true` si et seulement si la carte a effectivement été donnée.

- `payer` : Écrire la méthode `payer`. Cette méthode prend en paramètre un réel `prix` et un `String code` et retourne un `String` en fonction du déroulement de l'opération. L'objectif de la méthode est d'utiliser la carte possédée par la

personne pour payer le prix passé en paramètre. L'opération ne peut se faire que si (1) le code passé en paramètre correspond bien au code secret, (2) le retrait du prix passé en paramètre ne fait pas dépasser le découvert autorisé. On traitera les cas suivants avec un message de retour spécifique (une chaine de caractères) :
— si la personne ne possède pas de carte bancaire, rien ne se passe et la méthode retourne la chaine `"* pas de carte"` ;
— si le code passé en paramètre n'est pas le bon code secret, rien ne se passe et la méthode retourne la chaine `"* code incorrect"` ;
— si la carte bancaire ne permet pas le paiement, rien ne se passe et la méthode retourne la chaine `"* montant refuse"` ;
— si la carte bancaire autorise le paiement, le solde est débité et la méthode retourne la chaine `"* montant accepte"`.

- `toString` : Écrire la méthode `toString` qui permet d'afficher le statut d'une personne. Cette méthode retourne une chaine de la forme :
  "*nom*`(carteB: `*solde* `/ -`*decouvert*`)`"
  où
  — *nom* désigne le nom de la personne ;
  — *solde* désigne le solde de la carte ;
  — *decouvert* désigne le découvert autorisé.
  Si la personne ne possède pas de carte bancaire, l'affichage se limite à "*nom*`(pas de carte)`".

5.5 Tests

Question 4

Écrire la classe de test `TestPersonne` qui teste l'ensemble des méthodes de la classe `Personne` pour l'ensemble des situations possibles.

5.6 `Main` à écrire

En plus des tests, on souhaite écrire un programme principal qui vérifie que les méthodes fonctionnent bien. Le programme principal doit effectuer les opérations suivantes dans l'ordre :

1. dans une variable `carte`, créer une carte bancaire avec un solde de 100, un découvert autorisé de -1000 et un code secret égal à `"MonCode"` ;
2. dans une variable `albert`, créer une `Personne` nommée `"Albert"` ;
3. associer la carte bancaire `carte` à Albert ;
4. permettre à Albert de payer avec la carte bancaire un montant de 50 euros en utilisant le code correct ;
5. afficher à l'écran le résultat de l'opération ;
6. dans une variable `bertrand` créer une personne nommée `"Bertrand"` ;

7. faites donner la carte de Albert à Bertrand ;
8. permettre à Bertrand de payer avec la carte bancaire un montant de 500 euros en utilisant le code correct ;
9. afficher à l'écran le résultat de l'opération ;
10. demander à Albert de payer avec la carte bancaire un montant de 100 euros en utilisant le code correct ;
11. afficher à l'écran le résultat de l'opération (Albert n'a plus de carte et l'opération échoue) ;
12. demander à Bertrand de payer avec la carte bancaire un montant de 100 euros en utilisant un mauvais code ;
13. afficher à l'écran le résultat de l'opération (l'opération a échoué à cause du code incorrect).

Question 5

Écrire le `main` correspondant dans la classe `ProgCarte` et vérifier que les opérations se sont bien déroulées comme prévu (en affichant les personnes au fur et à mesure des instructions).

6 Tests

Avant de rendre votre projet, vous vérifierez que celui-ci fonctionne correctement en développant une classe de test par classe écrite (comme celles que vous avez utilisées pendant les TPs).

Pour cela, pour chaque classe (`CarteB` et `Personne`), vous allez

- ❑ réfléchir aux différents tests à écrire ;
- ❑ ajouter ces tests dans les classes de test fournies ;
- ❑ lancer vos tests et vérifier que votre classe fonctionne correctement par rapport à vos attentes.

6.1 Sélection des tests

Normalement, chaque méthode et chaque constructeur doit être testé pour tous les cas qui peuvent se présenter. Il faut donc dans un premier temps :

- ❑ déterminer toutes les méthodes et constructeurs à tester pour la classe qui vous intéresse ;
- ❑ déterminer les différents cas à tester pour chaque méthode ;
- ❑ pour chaque cas à tester, décider des valeurs de départ utilisées pour faire le test, les valeurs de retour attendues, faire le test et vérifier que les retours des méthodes correspondent bien à vos attentes.

Voici, par exemple, les tests à faire pour tester la méthode `deposer` qui prend en entrée un réel `montant`. Il est dit dans l'énoncé que si le paramètre réel est négatif, le

solde du compte ne doit pas évoluer. On peut donc envisager deux cas d'utilisation à tester de la méthode `deposer` :

- le cas où le paramètre `montant` est positif (par exemple 20) ;
- le cas où le paramètre `montant` est négatif (par exemple -20).

Cela doit conduire à deux tests distincts (cf suite).

6.2 Écriture de la classe de Test

Une fois l'ensemble des cas de test déterminés, il reste à vérifier que tous les tests sont validés lorsque votre programme s'exécute. Vous compléterez les classes de test correspondantes (`TestCarteB` et `TestPersonne`) en ajoutant une méthode de test pour chacun des cas identifiés.

6.2.1 Organisation des méthodes de test

Les méthodes de test doivent **IMPERATIVEMENT** suivre les conventions de nommage suivantes :

- chaque méthode de test se trouve dans la classe correspondant à la classe à tester (par exemple dans la classe `TestPersonne` pour les méthodes de `Personne`) ;
- le nom de chaque méthode de test
  - débute par le mot `"test"` ;
  - est suivie par le nom de la méthode testée précédé d'un underscore, par exemple `"_Deposer"` ;
  - puis d'un descriptif **sans accent** du cas testé, par exemple `"_DeposerOK"` ;

Ainsi, la méthode vérifiant que la méthode `deposer` de la classe `CarteB` fonctionne bien quand on recharge avec un montant positif s'appellera `"test_deposer_deposerOK()"`. Elle se trouvera dans la classe `TestCarteB`.

Pour finir, chaque méthode de test est précédée d'un commentaire javadoc qui explique le cas que teste la méthode, par exemple `"Test deposer montant avec un montant positif"`.

6.2.2 Ajout d'une méthode de test

Une méthode de test a pour objectif de vérifier que le test est effectivement valide.

- la méthode ne prend pas de paramètre et ne retourne rien ;
- les premières instructions de la méthode préparent les données (en appelant des constructeurs par exemple) ;
- les instructions suivantes exécutent le test à proprement parler (à savoir la méthode testée avec les bonnes données) ;
- les instructions de vérification appellent la méthode `assertEquals` à chaque fois qu'une condition attendue doit être vérifiée ;
- vous pouvez utiliser les accesseurs pour vérifier les valeurs des attributs mais les accesseurs n'ont pas besoin d'être testés (ils sont très simples).

Une fois qu'une méthode de test est écrite, elle doit rester dans votre classe de test. Ainsi, à l'issue du projet, vos classes de test posséderont toutes les méthodes de tests (une par cas à considérer) et permettront de tester l'ensemble de l'application.

6.2.3 Méthode `assertEquals`

Pour effectuer une vérification dans un test, il faut utiliser la méthode `assertEquals`. Cette méthode possède le profil suivant :

`void assertEquals(String erreur, Object attendu, Object obtenu)`

- `erreur` correspond au message d'erreur à afficher si la vérification n'est pas correcte ;
- `attendu` désigne la valeur attendue ;
- `obtenu` désigne la valeur obtenue lors du test.

Par exemple, si on veut tester que le '+' correspond bien à la concaténation, on ajouterait la chaine `"bon"` à la chaine `"jour"` et on vérifierait que le résultat serait bien la chaine `"bonjour"`.

```
String s1="bon";
String s2="jour";
String s3=s1+s2;
assertEquals("erreur: mauvaise concatenation","bonjour",s3);
```

6.2.4 Exemple

Pour le test de la méthode `deposer` de la classe `CarteB`, la classe `TestCarteB` s'écrirait de la manière suivante :

```
import static libtest.Lanceur.lancer;
import static libtest.OutilTest.assertEquals;
import libtest.*;

/******************
 * test classe CarteB
 ******************/
public class TestCarteB {

  /**
   * methode de lancement des tests
   */
  public static void main(String[] args) {
    lancer(new TestCarteB(), args);
  }

  /**
   * quand le depot est effectue correctement
   */
  public void test_deposer_OK() {
    // preparation des donnees
    CarteB carte = new CarteB(100,1000,"monCode");
```

```
23
24      // methode testee
25      carte.deposer(20);
26
27      // verifications
28      assertEquals("solde doit etre de 120", 120, carte.getSolde());
29      assertEquals("decouvert reste 1000", 1000, carte.getDecouvert());
30    }
31
32    /**
33     * quand le depot est effectue avec un montant negatif
34     */
35    public void test_deposer_negatif() {
36      // preparation des donnees
37      CarteB carte = new CarteB(100,1000,"monCode");
38
39      // methode testee
40      carte.deposer(-20);
41
42      // verifications
43      assertEquals("solde doit rester de 100", 100, carte.getSolde());
44      assertEquals("decouvert reste 1000", 1000, carte.getDecouvert());
45    }
46
47    //... autres tests de la classe CarteB
48  }
```

6.3 Classes de test fournies

On vous fournit sur l'ENT le package `libtest` ainsi qu'un début des classes de test `TestCarteB.java` et `TestPersonne.java`.

Les classes de test fournies possèdent déjà :

- une méthode `main` qui lance tous les tests que vous aurez écrits dans la classe ;
- un premier test qui vérifie que vos méthodes sont correctement écrites ;
- quelques autres tests qui vérifient vos constructeurs (pour être sûr que vous créez les bons objets).

Ces classes doivent compiler correctement si vos méthodes sont bien déclarées. Il ne faut pas changer les tests initiaux qui vérifient que vos méthodes sont conformes au sujet.

Une fois que vos tests seront ajoutés à cette classe, l'exécution de tous les tests ne doit conduire à aucune erreur (puisque tous vos tests doivent être valides). Faire de bons tests est un moyen de vous assurer de rendre un projet conforme aux attentes.

ATTENTION, **il ne vous est pas demandé de rendre un menu**, mais de rendre les classes de tests complétées par vos méthodes de test. Tout rendu autre que ce qui est attendu ne sera pas évalué.

6.4 Démarche

Pour faire correctement votre projet, nous vous conseillons de penser à tous les tests possibles dans un premier temps (cf partie sélection des test), de les programmer et ensuite seulement de commencer à écrire les différentes classes de votre programme. Cela vous permettra

— d'avoir pensé à tous les cas particuliers lorsque vous écrirez votre programme ;
— de disposer d'un programme plus sûr.

7 Génération de la javadoc

Pensez à écrire la `javadoc` dans vos fichiers sources et générez la javadoc dans le répertoire `javadoc` situé à la racine de votre projet.

Pour cela, le plus simple est de se placer dans le répertoire `gr_Z_nom` et d'exécuter la commande `javadoc`

- avec l'option `'-d dest'` pour spécifier le répertoire de destination (ici le répertoire de destination sera `javadoc`) ;
- avec comme argument les fichiers sources situés dans `src`, à savoir `'src/*.java'`.

```
1 javadoc -d javadoc src/*.java
```

La `javadoc` générée est à rendre avec votre projet.

Une fois la javadoc générée, il est possible de la consulter en ouvrant le fichier `javadoc/index.html`.

8 Question ouverte - `CarteBlocable`

Cette dernière question est une question plus ouverte pour laquelle la réponse n'est pas directement guidée par l'énoncé. Bien qu'ouverte, cette question est considérée comme faisant partie du sujet à faire et des points du barème y sont affectées.

On souhaite empêcher l'utilisation d'une carte lorsqu'on se trompe de code 3 fois d'affilée. Une fois qu'une personne s'est trompée de code trois fois de suite, la carte est bloquée et n'est plus utilisable.

Pour cela, on souhaite proposer une autre classe nommée `CarteBlocable` construite à partir de la classe `Carte`. On veut trouver une manière simple d'ajouter ce blocage après trois essais successifs infructueux. Ainsi, seuls les attributs et la méthode `depenser` doivent être réécrites, le reste des méthodes sera supposé fonctionner de manière identique.

Question 6

Copier coller la classe `CarteB` et renommer la en `CarteBlocable` et renommer les constructeurs [a]. Aprés cette question, vous devez avoir deux classes `CarteB` et

`CarteBlocable` qui fonctionnent de la même manière.

a. Attention, faire un copier-coller est toujours une mauvaise solution pour écrire du code, mais vous ne disposez pas encore des outils nécessaires pour éviter cela.

Question 7

En commentaires en début de la classe `CarteBlocable`, expliquer en français comment la classe `CarteBlocable` peut fonctionner. Quels doivent être les attributs supplémentaires et comment la méthode `depenser` doit marcher.

Question 8

Ajouter les attributs et modifier la méthode `depenser` de carte blocable en intégrant ce fonctionnement.

Vous pouvez compléter la classe de test fournie `TestCarteBlocable` pour vérifier que votre classe fonctionne correctement.

9 Dernier conseil

Le projet que vous allez rendre forme un tout (classes + tests + javadoc).

— Si vous faites les bons tests, vous pourrez facilement détecter les erreurs de votre projet et le corriger.

— Inversement si les tests ne sont pas complets, vous risquez d'avoir des classes qui fonctionnent mal et un projet qui ne fournit pas les bons résultats.

Ainsi, le meilleur moyen d'avoir une bonne note est de mettre l'accent sur les tests qui vous permettront d'écrire un code correct et valide (et bien entendu de corriger les classes lorsque les tests ne passent pas).

Pour information, le fait d'écrire les tests dans le corrigé a permis d'éviter la présence de plusieurs erreurs d'inattention dans le corrigé du projet. Ils vous assurent aussi qu'une fois que votre projet est fini, tous les tests restent valides (vous n'avez donc pas introduit une nouvelle erreur dans votre programme en faisant une modification).