Gestion des exceptions personnalisées

Une exception personnalisée en Java est une classe à part entière.
Elle hérite généralement de :

1. Exception
2. ou RuntimeException pour les exceptions non vérifiées,

elle peut contenir :

- **Des attributs:** Pour stocker des informations supplémentaires sur l'exception (par exemple, un code d'erreur, une date, un objet causant l'exception ou même des objets représentant l'état du système au moment où l'exception a été levée).
- **Des méthodes:** Pour fournir des fonctionnalités spécifiques liées à l'exception (par exemple, une méthode pour obtenir un message d'erreur détaillé, une méthode pour récupérer la cause racine de l'exception).

Pourquoi créer des exceptions personnalisées ?

- **Spécificité:** Elles permettent de représenter des erreurs spécifiques à votre application, rendant le code plus clair et plus facile à déboguer.
- **Informations contextuelles:** Vous pouvez y stocker des données supplémentaires qui aideront à comprendre la nature de l'erreur.
- **Gestion fine:** Vous pouvez définir des hiérarchies d'exceptions pour une gestion plus granulaire.

Les exceptions personnalisées en Java : bien plus que de simples constructeurs

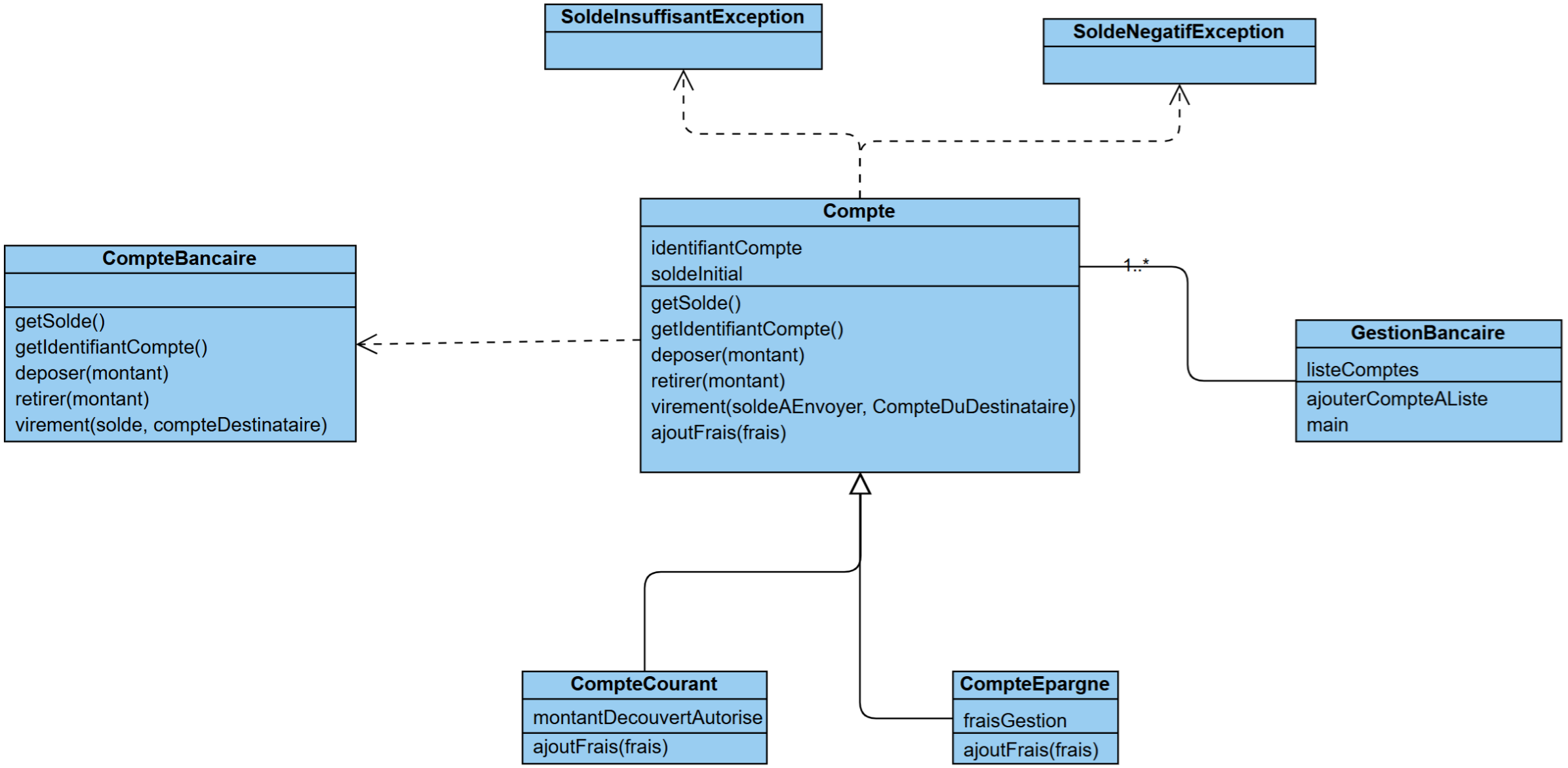
```
public class SoldeInsuffisantException extends Exception {
   private double montantDemande;
   private double soldeActuel;
   private LocalDateTime dateHeure;

   public SoldeInsuffisantException(double montantDemande, double
                                                        soldeActuel) {
      super("Solde insuffisant : vous avez demandé " + montantDemande +
            « DHS mais votre solde est de " + soldeActuel + « DHS");
      this.montantDemande = montantDemande;
      this.soldeActuel = soldeActuel;
      this.dateHeure = LocalDateTime.now();
   }
   public double getMontantDemande() {
      return montantDemande;
   }
   public double getSoldeActuel() {
      return soldeActuel;
   }
   public LocalDateTime getDateHeure() {
      return dateHeure;
   }
}
```

```
public void retirer(double montant) throws
SoldeInsuffisantException {
   if (montant > solde) {
      throw new
SoldeInsuffisantException(montant, solde);
   }
   solde -= montant;
}

…

try {
   // retirer un montant superieur au solde
du client
   compte.retirer(5000);
} catch (SoldeInsuffisantException e) {
System.err.println(e.getMessage());
System.err.println("Montant demandé : " +
e.getMontantDemande());
System.err.println("Solde actuel : " +
e.getSoldeActuel());
System.err.println("Date et heure de
l'opération : " + e.getDateHeure()); }
```

Créer votre propre classe d’exception !!

Pour le compte d’un de nos clients, notre équipe de développeurs a pour charge de créer une application permettant de gérer les retraits et les dépôts d’argents sur des comptes courant ou d’épargne.
Pour ce faire, nous allons construire :

- **Une interface CompteBancaire:** Définit les opérations de base d'un compte bancaire (obtenir le solde, déposer et retirer, faire un virement).
- **Classe Compte:** Implémente l'interface CompteBancaire et représente un compte d’un client.
- **Les classes CompteCourant et CompteEpargne:** qui héritent de la classe Compte et profite de ses méthodes pour gérer les comptes, ainsi qu’ils contiennent des attributs spécifiques et une méthode redéfinie.
- **Classe Banque:** Gère une liste de comptes bancaires et contient aussi la méthode principale main.
- **Exceptions personnalisées:** SoldeNegatifException et SoldeInsuffisantException sont utilisées pour signaler des erreurs spécifiques liées aux opérations sur les comptes.


SoldeInsuffisantException
SoldeNegatifException
Compte
identifiantCompte
soldeInitial
getSolde()
getIdentifiantCompte()
deposer(montant)
retirer(montant)
virement(soldeAEnvoyer, CompteDuDestinataire)
ajoutFrais(frais)
1..*
CompteBancaire
getSolde()
getIdentifiantCompte()
deposer(montant)
retirer(montant)
virement(solde, compteDestinataire)
GestionBancaire
listeComptes
ajouterCompteAListe
main
CompteCourant
montantDecouvertAutorise
ajoutFrais(frais)
CompteEpargne
fraisGestion
ajoutFrais(frais)

C’est quoi une interface?

- **Interface** = un contrat
- L’interface définit un ensemble de méthodes que les classes qui l'implémentent doivent obligatoirement posséder.

➔C'est un peu comme un plan : elle décrit ce qu'une classe doit être capable de faire, sans spécifier comment le faire.

Pourquoi utiliser des interfaces ?

- Abstraction: Elles permettent de se concentrer sur le "quoi" plutôt que le "comment", favorisant ainsi une meilleure compréhension du code.
- Polymorphisme: Grâce aux interfaces, on peut utiliser des objets de différentes classes de manière uniforme, si ces classes implémentent la même interface.
- Couplage faible: Les interfaces réduisent le couplage entre les différentes parties d'un programme, ce qui facilite la maintenance et les tests.
- Programmation par contrat: Elles définissent un contrat précis entre les différents éléments du système.

```
public interface Forme {
    double calculerAire();
    double calculerPerimetre();
}
```

Toute classe qui implémente cette interface devra fournir une implémentation pour ces deux méthodes.

```
public class Cercle implements Forme {
    private double rayon;
    // ...
    @Override
    public double calculerAire() {
        return Math.PI * rayon * rayon;
    }
    @Override
    public double calculerPerimetre() {
        return 2 * Math.PI * rayon;
    }
}
```

```
interface Forme {
    double calculerAire();
}
```

```
class Cercle implements Forme {
    // ...
    public double calculerAire() {
        // Calcul de l'aire d'un cercle
    }
}

class Carre implements Forme {
    // ...
    public double calculerAire() {
        // Calcul de l'aire d'un carré
    }
}
```

```
List<Forme> formes = new ArrayList<>();
formes.add(new Cercle(5));
formes.add(new Carre(4));
```

```
for (Forme forme : formes) {
    System.out.println("L'aire de la forme est
: " + forme.calculerAire());
}
```

- **Polymorphisme:** Grâce au polymorphisme, la variable forme dans la boucle for peut prendre la valeur de n'importe quel objet qui implémente l'interface Forme.

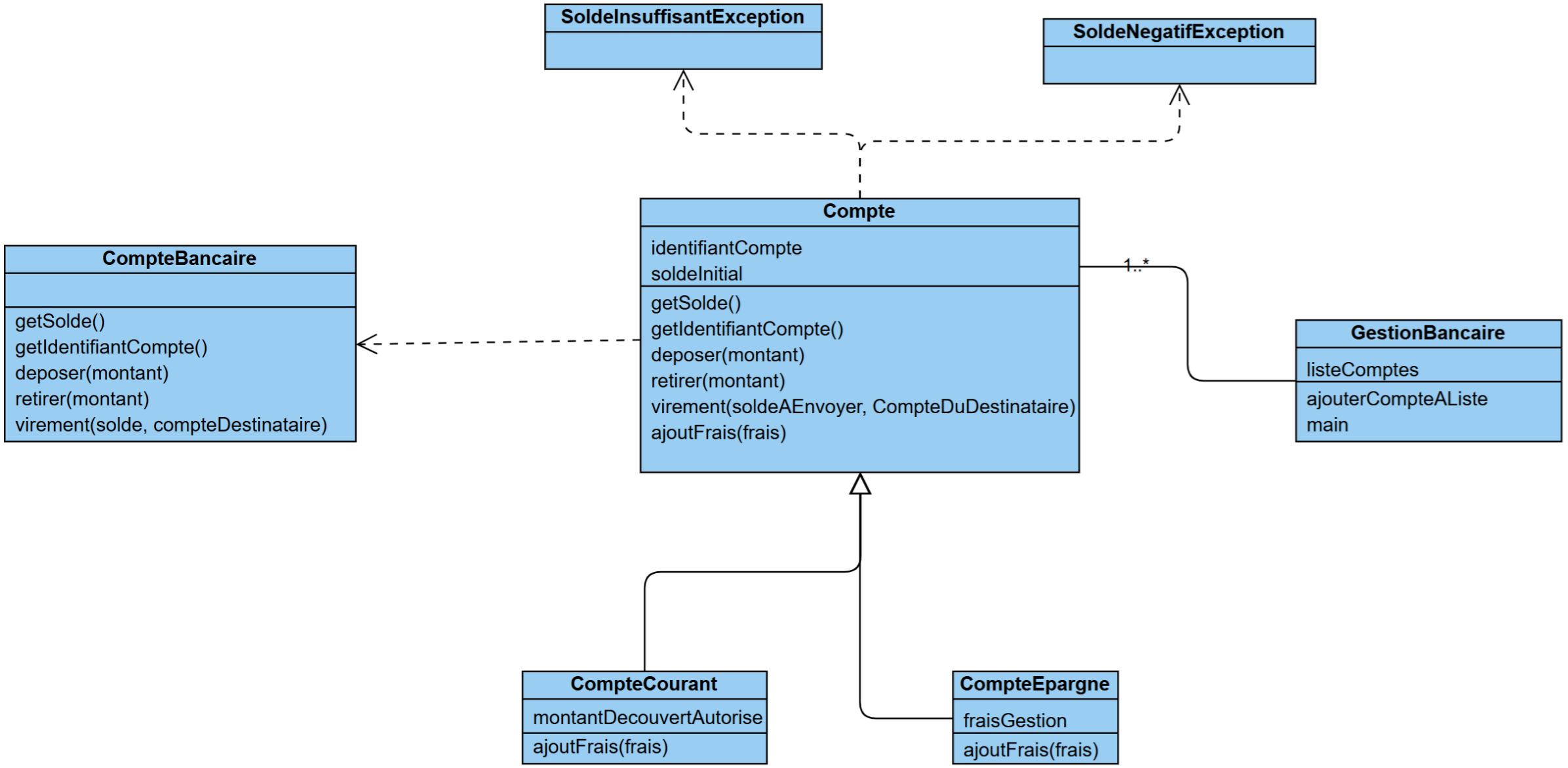
  Le compilateur vérifie que tous les objets ajoutés à la liste implémentent bien l'interface Forme, garantissant ainsi la sécurité du code.

Les interfaces

- **Toutes les méthodes d'une interface sont implicitement publiques et abstraites.** Cela signifie qu'une classe qui implémente une interface doit fournir une implémentation concrète pour chacune de ces méthodes.

- **Il n'existe pas de notion d'héritage multiple direct pour les classes en Java.** Une classe ne peut étendre qu'une seule classe parente. Cependant, une classe peut implémenter plusieurs interfaces.

- **Interfaces par défaut (Java 8 et plus):** À partir de Java 8, il est possible de fournir une implémentation par défaut pour certaines méthodes d'une interface. Cela permet d'ajouter de nouvelles fonctionnalités aux interfaces existantes sans obliger toutes les classes qui l'implémentent à redéfinir ces méthodes. Cependant, les classes qui ont besoin d'un comportement différent peuvent toujours redéfinir ces méthodes.


SoldeInsuffisantException
SoldeNegatifException
Compte
identifiantCompte
soldeInitial
getSolde()
getIdentifiantCompte()
deposer(montant)
retirer(montant)
virement(soldeAEnvoyer, CompteDuDestinataire)
ajoutFrais(frais)
1..*
CompteBancaire
getSolde()
getIdentifiantCompte()
deposer(montant)
retirer(montant)
virement(solde, compteDestinataire)
GestionBancaire
listeComptes
ajouterCompteAListe
main
CompteCourant
montantDecouvertAutorise
ajoutFrais(frais)
CompteEpargne
fraisGestion
ajoutFrais(frais)

```
package GestionBanque;

public interface CompteBancaire {
   double getSolde();

   String getIdentifiantCompte();
   void deposer(double montant) throws SoldeNegatifException;
   void retirer(double montant) throws SoldeInsuffisantException;
   void virement(double solde, CompteBancaire c) throws SoldeNegatifException, SoldeInsuffisantException;
}
```

```
package GestionBanque;

public abstract class Compte
implements CompteBancaire {

   private String identifiantCompte;
   private double solde;

   public Compte(String id, double
                        soldeInitial) {
      this.identifiantCompte = id;
      this.solde = soldeInitial;
   }
   @Override
   public double getSolde() {
      return solde;
   }
   @Override
   public String getIdentifiantCompte()
         {
         return identifiantCompte;
         }
   public abstract void
         AjoutFrais(double fraisAjoute);
```

```
@Override
   public void deposer(double
         montant) throws
         SoldeNegatifException {
      if (montant < 0) {
         throw new
         SoldeNegatifException ("Le
         montant à déposer doit être
         positif");
      }
      solde += montant;
   }
   @Override
   public void retirer(double
         montant) throws
         SoldeInsuffisantException {
      if (montant > solde) {
         throw new
         SoldeInsuffisantException
                  (montant, solde);
      }
      solde -= montant;
   }
```

```
@Override
   public void virement(double solde,
         CompteBancaire c) throws
         SoldeInsuffisantException,
         SoldeNegatifException {

// retirer le solde de l'emetteur
System.out.println("Le compte
"+identifiantCompte+" emet un solde égal
à : "+solde);
System.out.println("Le compte
"+c.getIdentifiantCompte()+" reçoit le
solde : "+solde);
this.retirer(solde);

// deposer le solde chez le destinataire
c.deposer(solde);
System.out.println("solde emetteur :
"+getSolde());
System.out.println("solde destinataire:
"+c.getSolde());
}

}
```

```
package GestionBanque;
class CompteCourant extends Compte {
   double montantDecouvert;
   public CompteCourant(String id, double solde,
                        double montantDecouvert) {
      super(id, solde);
      this.montantDecouvert = montantDecouvert;
   }
   @Override
   public void AjoutFrais(double a) {
      try {
         this.retirer(a);
         System.out.println("solde du compte courant
                                    est:"+this.getSolde());
      }
      catch (SoldeInsuffisantException e){
         System.out.println("ajout frais pour le compte
              courant pas possible!"+e.getMessage());
      }
   }
}
```

```
package GestionBanque;
public class CompteEpargne extends Compte{
   double fraisGestion;
   public CompteEpargne(String id, double solde, double
                                          fraisGestion) {
      super(id, solde);
      this.fraisGestion = fraisGestion;
   }
   @Override
   public void AjoutFrais(double a) {
      try {
         this.retirer(a);
         this.retirer(this.fraisGestion);
         System.out.println("solde du compte d'epargne
                                          est:"+this.getSolde());
      }
      catch (SoldeInsuffisantException e){
         System.out.println("ajout frais pour le compte Epargne
                                          pas possible!"+e.getMessage());
      }
   }
}
```

```
package GestionBanque;

public class SoldeNegatifException
extends Exception{
   public SoldeNegatifException(String
                           message) {
      super(message);
   }
}
```

```
package GestionBanque;
import java.time.LocalDateTime;
public class SoldeInsuffisantException extends Exception {
   private double montantDemande;
   private double soldeActuel;
   private LocalDateTime dateHeure;
   public SoldeInsuffisantException(double montantDemande, double
            soldeActuel) {
      super("Solde insuffisant : vous avez demandé " + montantDemande + "DHS
                        mais votre solde est de " + soldeActuel + "DHS");
      this.montantDemande = montantDemande;
      this.soldeActuel = soldeActuel;
      this.dateHeure = LocalDateTime.now();
   }
   public double getMontantDemande() {
      return montantDemande;
   }
   public double getSoldeActuel() {
      return soldeActuel;
   }
   public LocalDateTime getDateHeure() {
      return dateHeure;
   }
}
```

```
package GestionBanque;
import java.util.ArrayList;
public class GestionBancaire {
   private ArrayList<Compte> comptes;
   public GestionBancaire() {
      comptes = new ArrayList<>();
   }
   public void ajouterCompte(Compte
                              compte) {
      comptes.add(compte);
   }
   // ... autres méthodes pour gérer les comptes
public static void main(String[] args) {
      GestionBancaire banque = new
                                    GestionBancaire();
      Compte compte1 = new
            CompteCourant("compte1", 1000,
                                          1000);
      Compte compte2 = new
            CompteCourant("compte2", 1000,
                                          500);
      Compte compte3 = new
            CompteEpargne("compte3", 1000,
                                          20);
      banque.ajouterCompte(compte1);
      banque.ajouterCompte(compte2);
      banque.ajouterCompte(compte3);
      for (Compte c : banque.comptes) {
         c.AjoutFrais(20);                    // Le polymorphisme
      }
      try {
         compte1.deposer(5000);
         compte1.virement(2000, compte2);
         System.out.println("Le compte"+compte1.getIdentifiantCompte()+"
                                    a transfere 2000dh au
                                    compte"+compte2.getIdentifiantCompte());
         System.out.println("Le compte"+compte1.getIdentifiantCompte()+"a
                                    maintenant "+compte1.getSolde()+" dhs");
         System.out.println("Le compte"+compte2.getIdentifiantCompte()+"a
                                    maintenant "+compte2.getSolde()+" dhs");
      } catch (SoldeInsuffisantException e) {
         System.out.println("Solde insuffisant : " + e.getMessage());
      } catch (SoldeNegatifException e) {
         System.out.println("Solde negatif : " + e.getMessage());
      }
   }
}
```