+1/1/60+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `public`
- ☐ `protected`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

☐ `something` ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `something` ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `something` ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

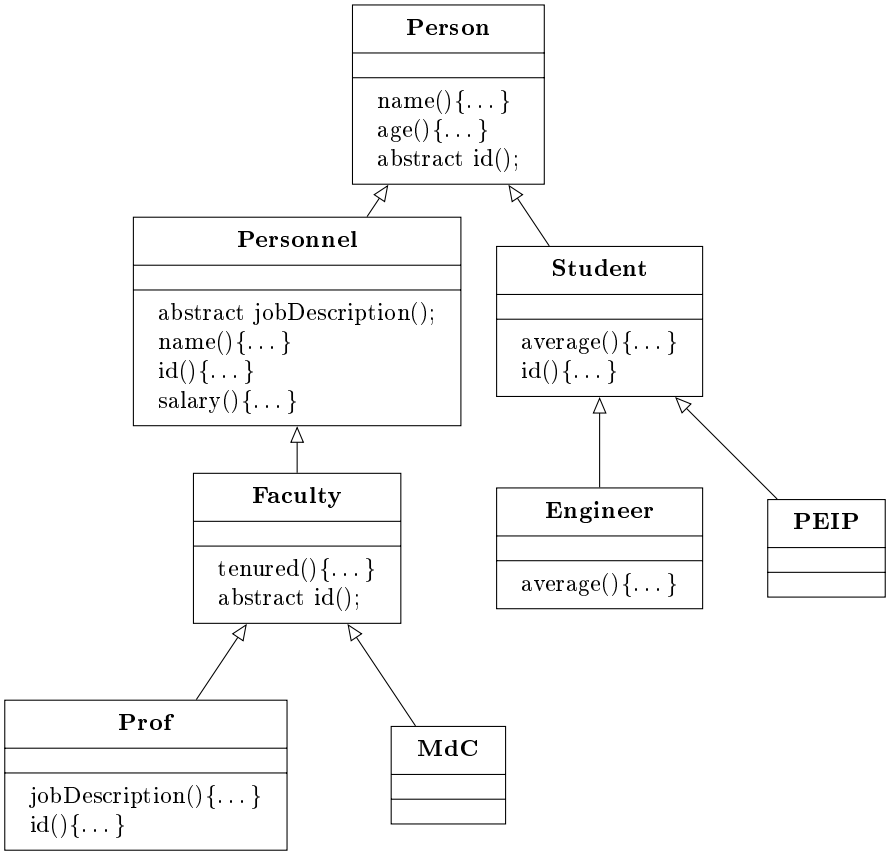
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Personnel`
- ☐ `MdC`
- ☐ `PEIP`
- ☐ `Student`
- ☐ `Faculty`
- ☐ `Person`
- ☐ `Engineer`
- ☐ `Prof`

+2/1/50+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche "Done"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ `private`
- ☐ package-private
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

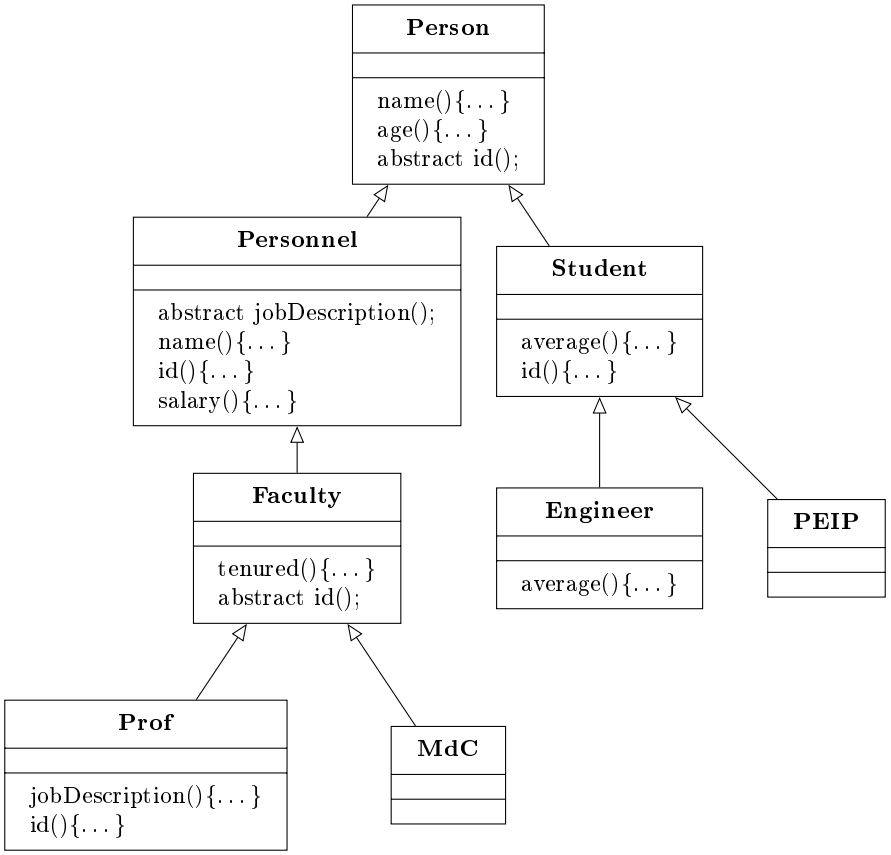
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

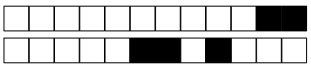
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Faculty`
- ☐ `Person`
- ☐ `PEIP`
- ☐ `Prof`
- ☐ `Engineer`
- ☐ `Student`
- ☐ `Personnel`
- ☐ `MdC`



+3/1/40+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Done"
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ package-private
- ☐ `public`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

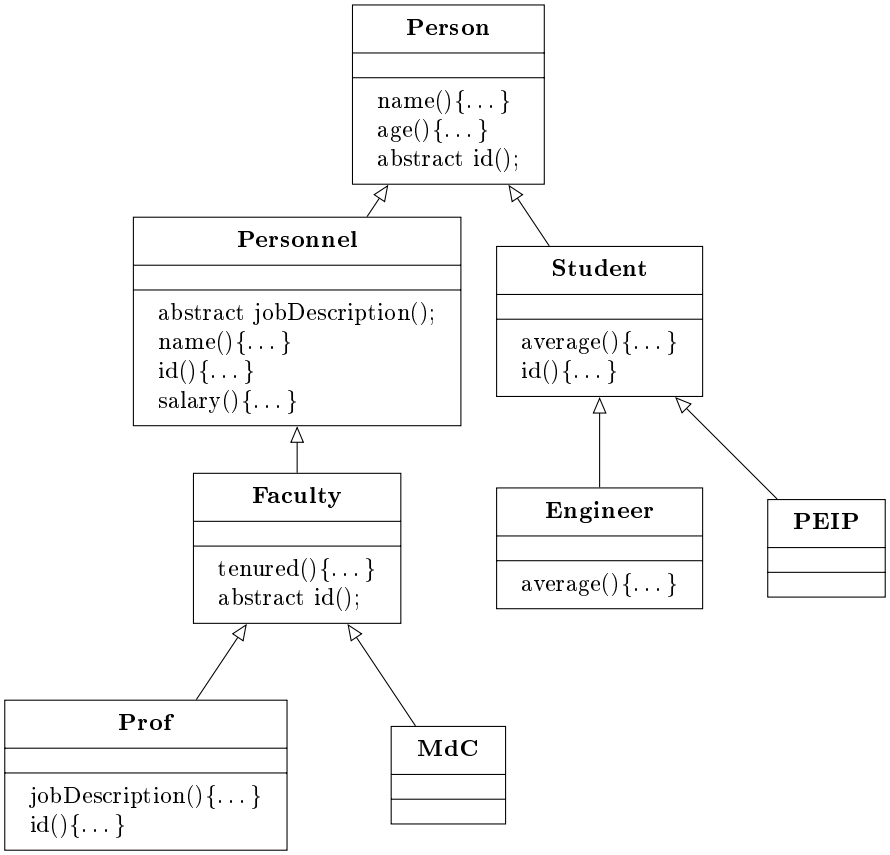
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

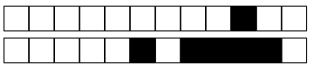
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `MdC`
- ☐ `PEIP`
- ☐ `Student`
- ☐ `Faculty`
- ☐ `Prof`
- ☐ `Personnel`
- ☐ `Engineer`
- ☐ `Person`



+4/1/30+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

☐ Erreur d'exécution

☐ Erreur de compilation

☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `private`
- ☐ `protected`
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

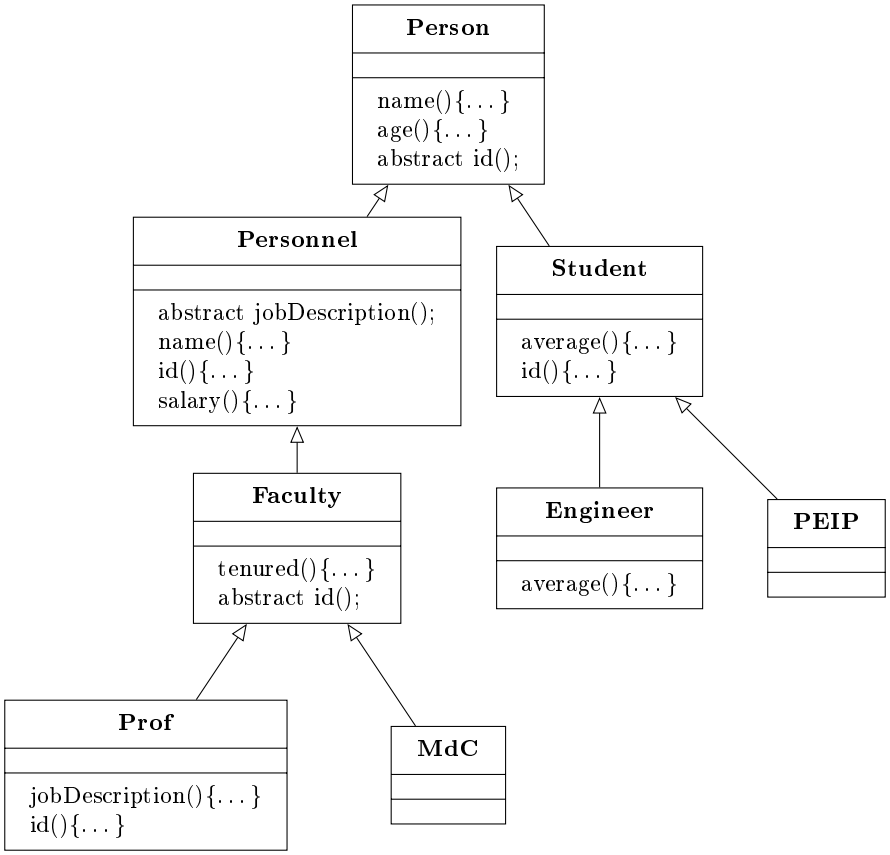
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

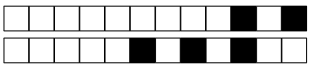
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `MdC`
- ☐ `Personnel`
- ☐ `Prof`
- ☐ `PEIP`
- ☐ `Faculty`
- ☐ `Engineer`
- ☐ `Person`
- ☐ `Student`



+5/1/20+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

☐ Erreur de compilation

☐ Affiche seulement "Caught exception"

☐ Affiche "Caught exception" et "Finallied exception"

☐ Affiche seulement "Finallied exception"

☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Done"
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ `public`
- ☐ `private`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

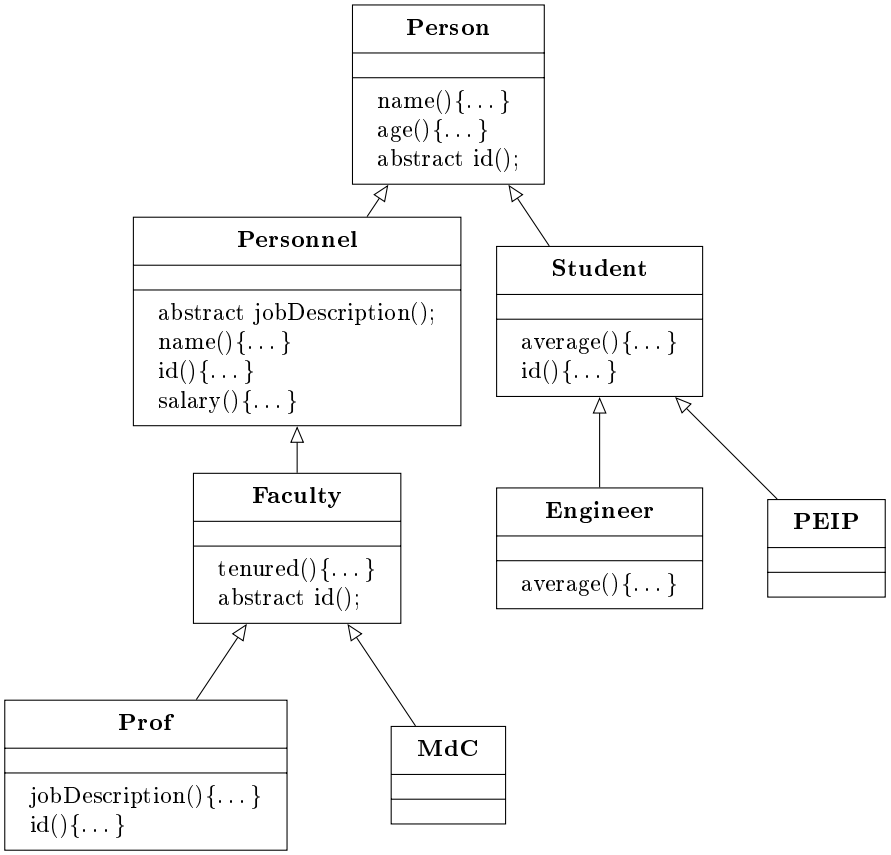
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

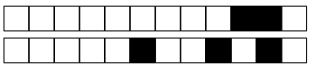
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `PEIP`
- ☐ `MdC`
- ☐ `Student`
- ☐ `Prof`
- ☐ `Engineer`
- ☐ `Personnel`
- ☐ `Faculty`
- ☐ `Person`



+6/1/10+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

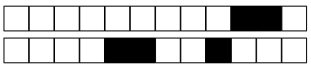
```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ `protected`
- ☐ `private`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`



Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `whatever` ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `whatever` ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre une classe abstraite

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

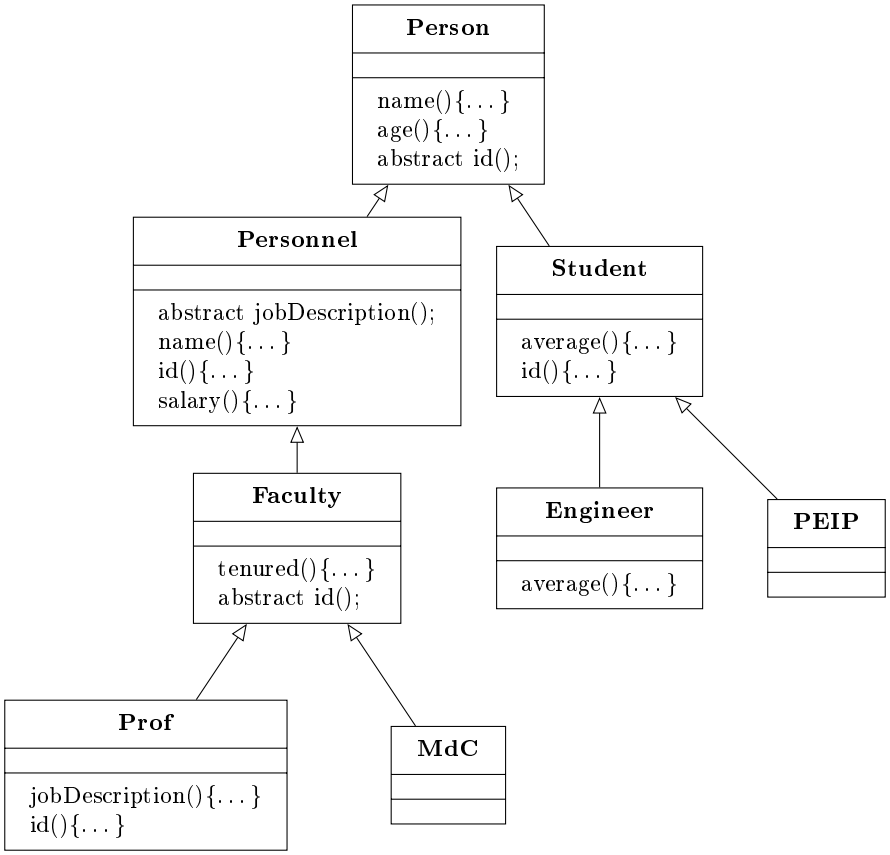
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Student`
- ☐ `Person`
- ☐ `MdC`
- ☐ `Faculty`
- ☐ `Engineer`
- ☐ `Prof`
- ☐ `Personnel`
- ☐ `PEIP`

+7/1/60+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

☐ Erreur de compilation

☐ Erreur d'exécution

☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ package-private
- ☐ `protected`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

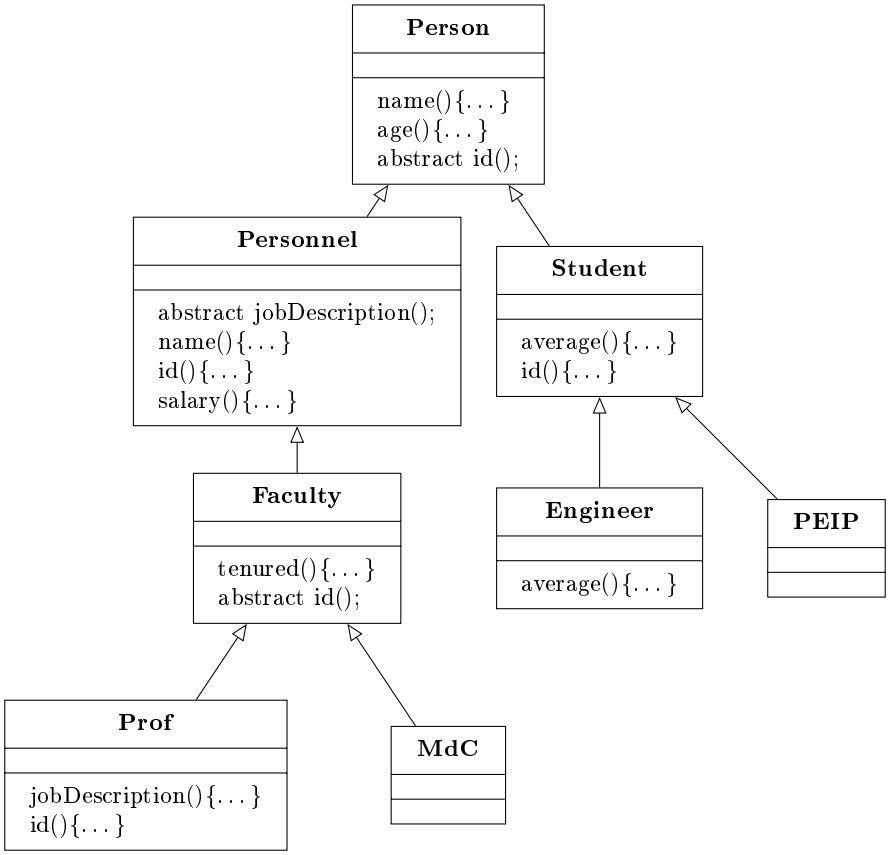
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Person`
- ☐ `Prof`
- ☐ `Personnel`
- ☐ `Student`
- ☐ `Faculty`
- ☐ `MdC`
- ☐ `PEIP`
- ☐ `Engineer`

+8/1/50+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

☐ Erreur de compilation

☐ Erreur d'exécution

☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `public`
- ☐ `protected`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

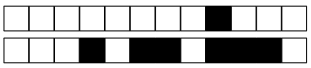
```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4



Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

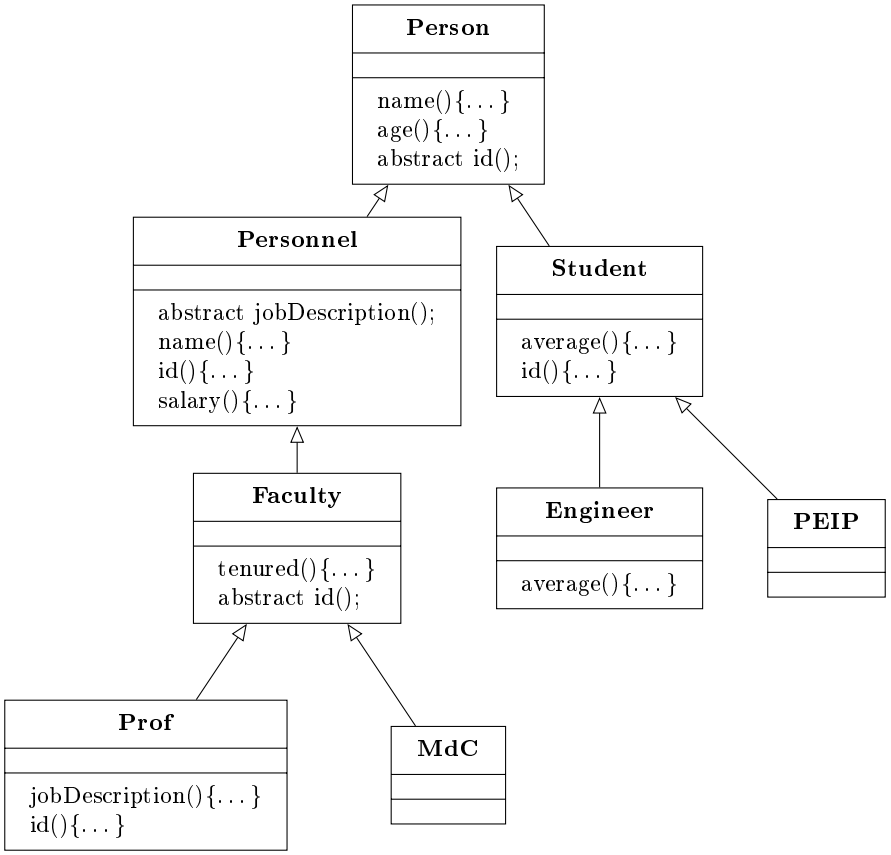
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

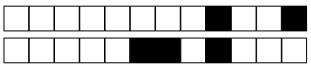
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `MdC`
- ☐ `PEIP`
- ☐ `Student`
- ☐ `Personnel`
- ☐ `Prof`
- ☐ `Person`
- ☐ `Engineer`
- ☐ `Faculty`



+9/1/40+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Done"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ `private`
- ☐ `protected`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

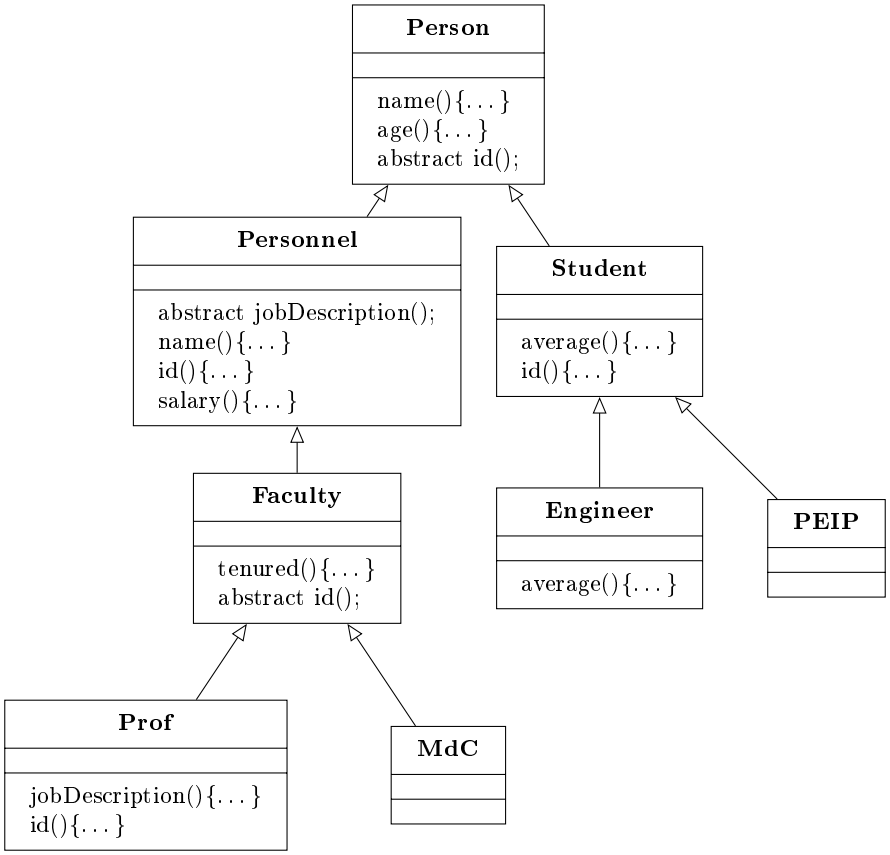
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

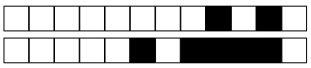
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Person`
- ☐ `Prof`
- ☐ `PEIP`
- ☐ `Engineer`
- ☐ `Personnel`
- ☐ `Student`
- ☐ `Faculty`
- ☐ `MdC`



+10/1/30+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

☐ Affiche "Done"

☐ Erreur de compilation

☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ package-private
- ☐ `public`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

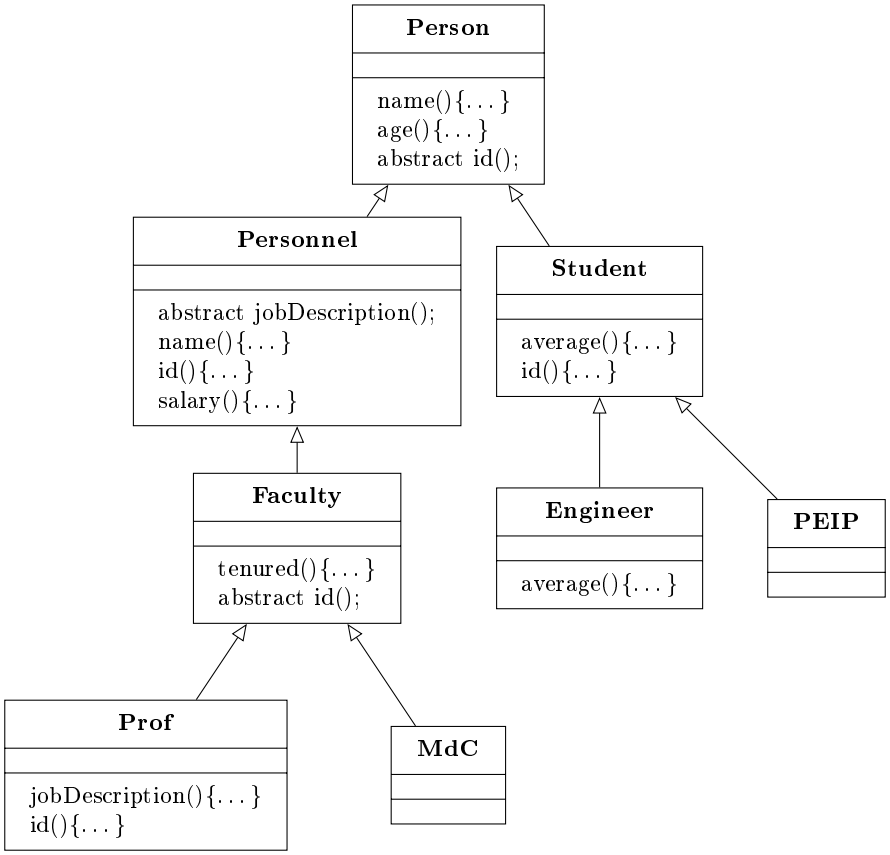
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Faculty`
- ☐ `Personnel`
- ☐ `Person`
- ☐ `Engineer`
- ☐ `PEIP`
- ☐ `Student`
- ☐ `MdC`
- ☐ `Prof`

+11/1/20+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

☐ Affiche seulement "Caught exception"

☐ Erreur d'exécution

☐ Affiche "Caught exception" et "Finallied exception"

☐ Affiche seulement "Finallied exception"

☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ `private`
- ☐ `protected`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `something` ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `something` ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut implémenter une interface

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

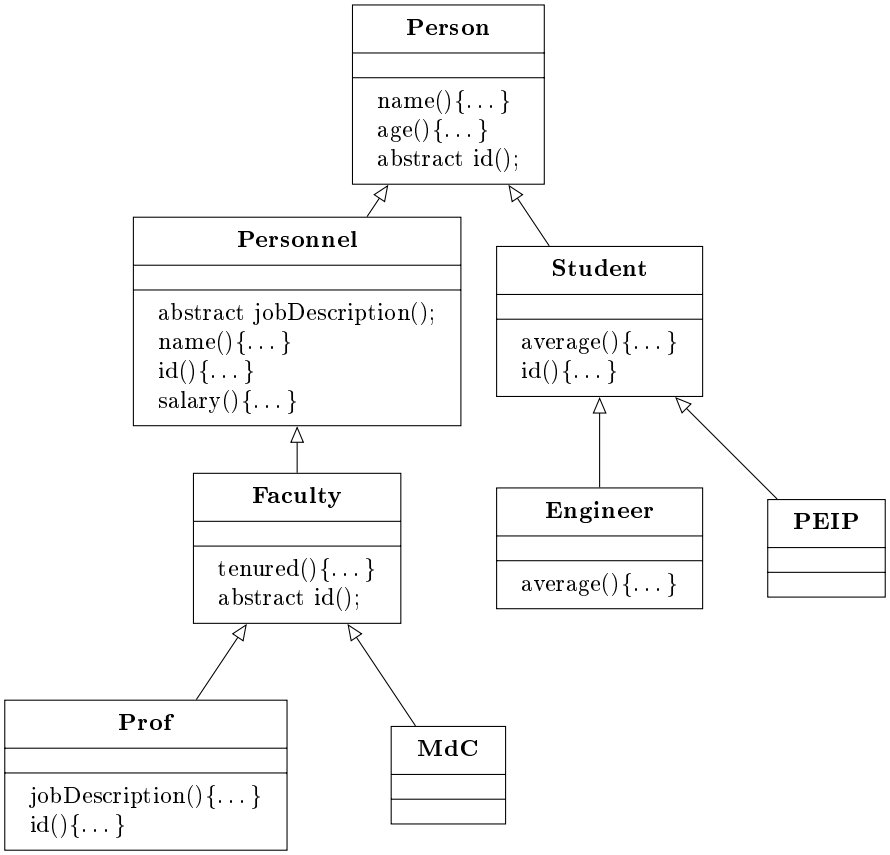
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

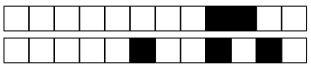
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `MdC`
- ☐ `Student`
- ☐ `Personnel`
- ☐ `PEIP`
- ☐ `Faculty`
- ☐ `Person`
- ☐ `Prof`
- ☐ `Engineer`



+12/1/10+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

☐ Erreur d'exécution

☐ Affiche "Done"

☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

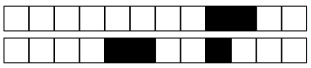
```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ package-private
- ☐ `public`
- ☐ `protected`



Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

☐ `something` ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `something` ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `something` ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

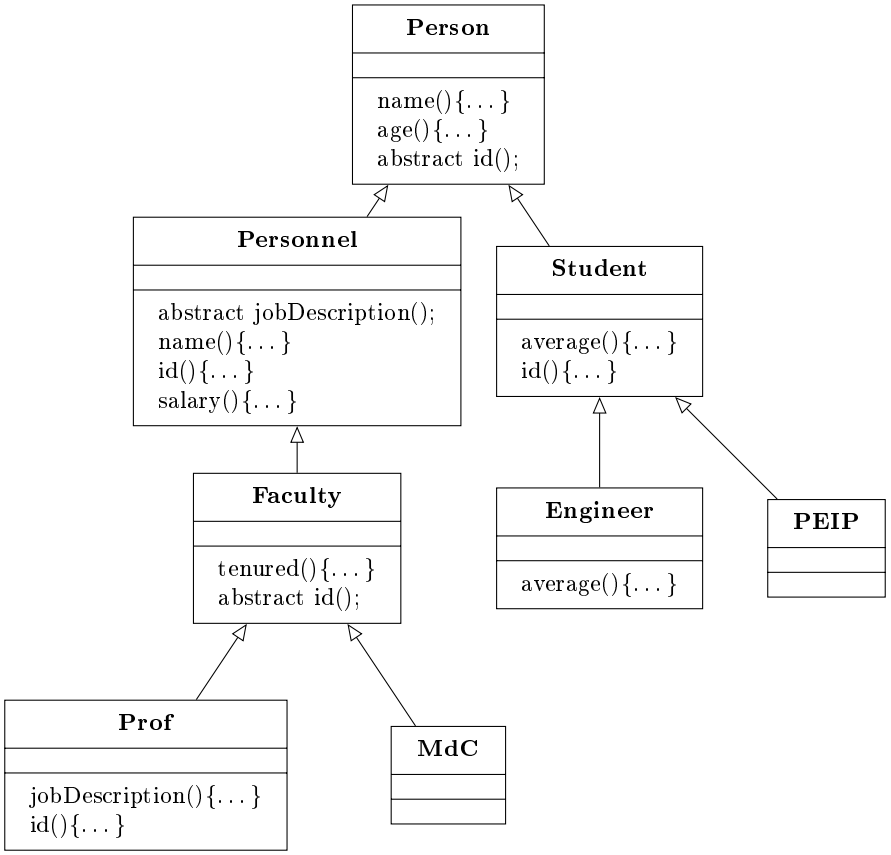
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Personnel`
- ☐ `Student`
- ☐ `Engineer`
- ☐ `PEIP`
- ☐ `Faculty`
- ☐ `Person`
- ☐ `Prof`
- ☐ `MdC`

+13/1/60+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `public`
- ☐ `private`
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `whatever` ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `something` ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

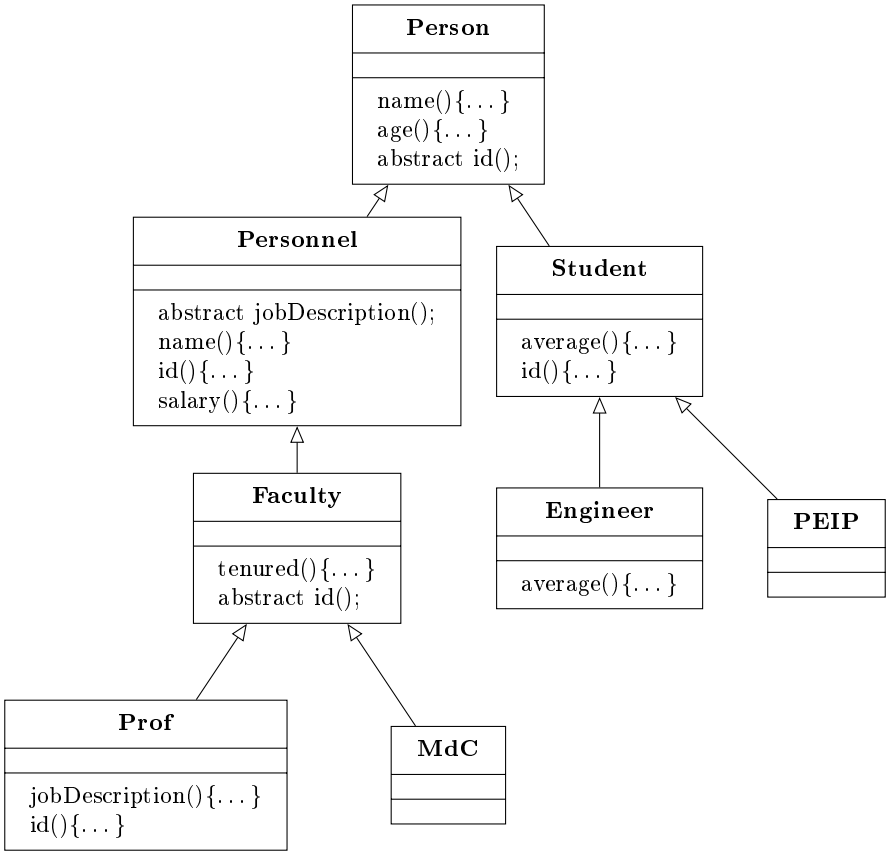
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `MdC`
- ☐ `Faculty`
- ☐ `PEIP`
- ☐ `Student`
- ☐ `Personnel`
- ☐ `Prof`
- ☐ `Person`
- ☐ `Engineer`

+14/1/50+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Done"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ `private`
- ☐ package-private
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `something` ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `something` ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre une classe abstraite

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

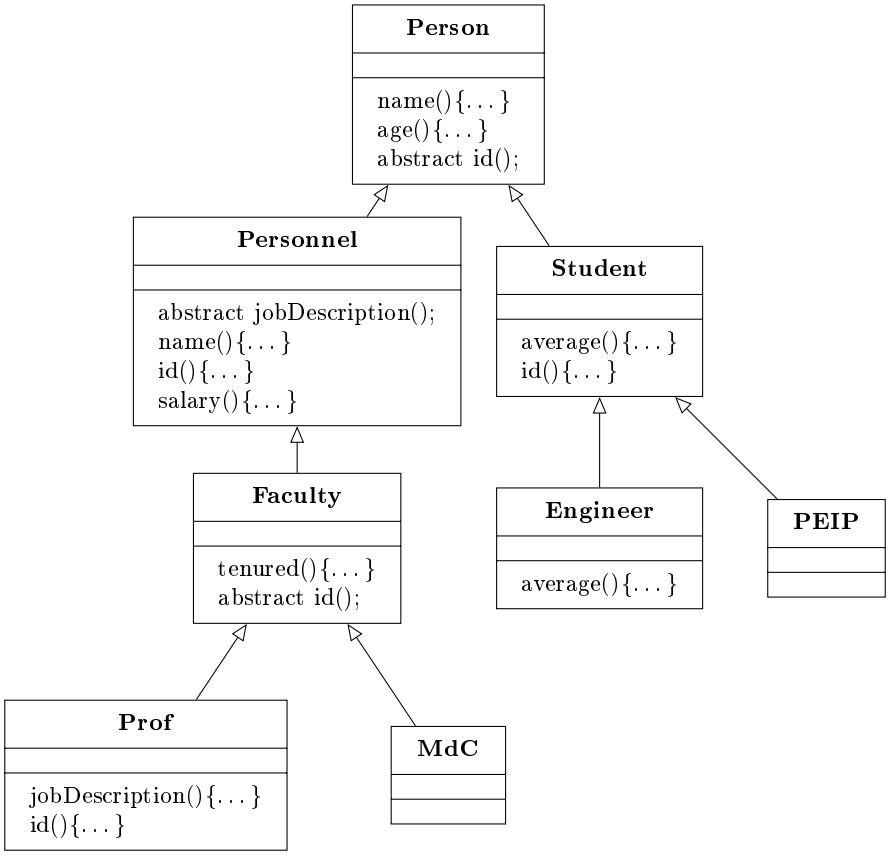
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

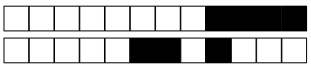
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `MdC`
- ☐ `Prof`
- ☐ `Engineer`
- ☐ `Faculty`
- ☐ `Person`
- ☐ `Student`
- ☐ `PEIP`
- ☐ `Personnel`



+15/1/40+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

☐ Erreur de compilation

☐ Affiche "Done"

☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ `public`
- ☐ `private`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

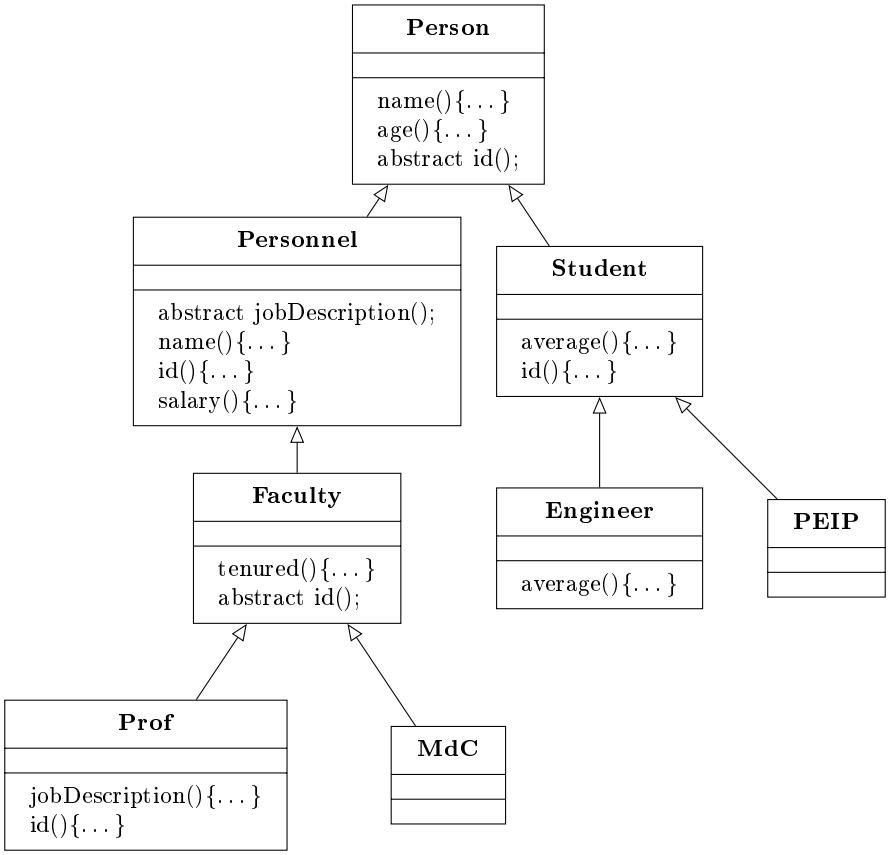
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

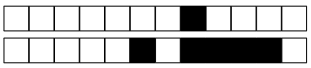
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Prof`
- ☐ `Student`
- ☐ `PEIP`
- ☐ `Faculty`
- ☐ `Personnel`
- ☐ `Person`
- ☐ `Engineer`
- ☐ `MdC`



+16/1/30+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ `protected`
- ☐ `private`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut implémenter une interface

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

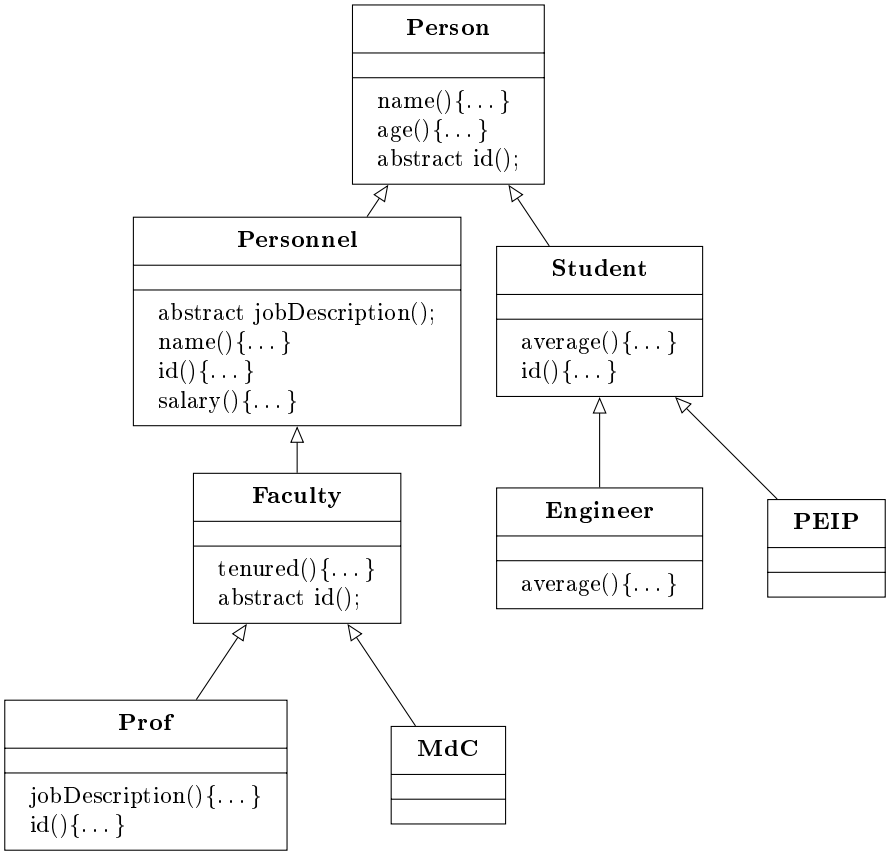
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

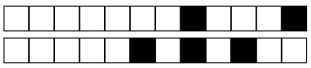
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Faculty`
- ☐ `Engineer`
- ☐ `Student`
- ☐ `Personnel`
- ☐ `Prof`
- ☐ `PEIP`
- ☐ `MdC`
- ☐ `Person`



+17/1/20+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

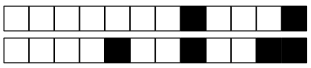
☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`



Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Done"
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ `protected`
- ☐ package-private
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

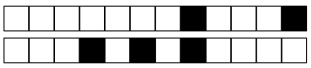
```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4



Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe abstraite

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

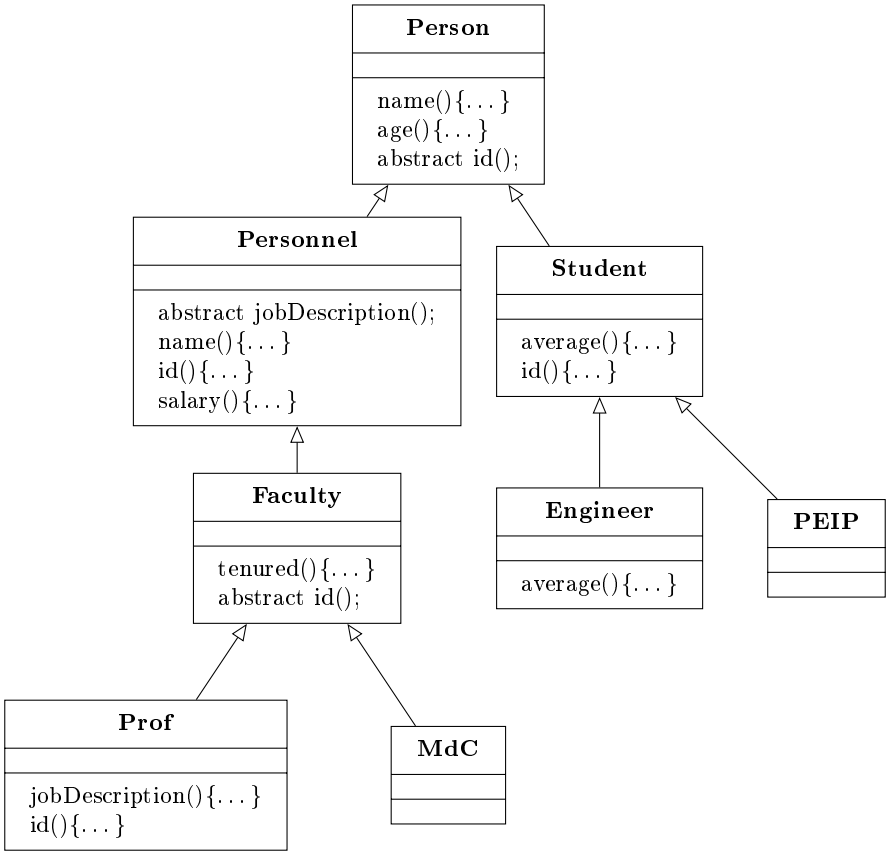
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

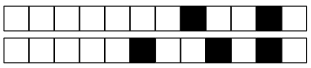
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Student`
- ☐ `Faculty`
- ☐ `Engineer`
- ☐ `Prof`
- ☐ `MdC`
- ☐ `Person`
- ☐ `Personnel`
- ☐ `PEIP`



+18/1/10+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

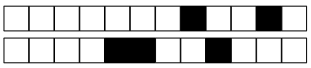
```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ package-private
- ☐ `private`
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`



Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

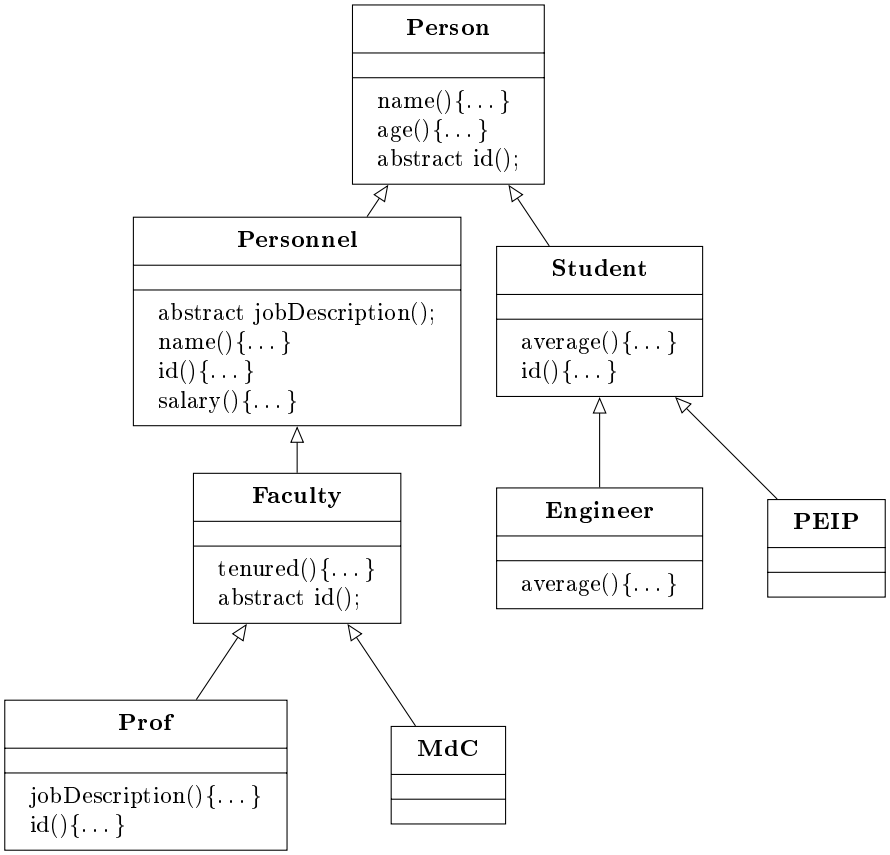
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Engineer`
- ☐ `MdC`
- ☐ `Prof`
- ☐ `PEIP`
- ☐ `Student`
- ☐ `Personnel`
- ☐ `Person`
- ☐ `Faculty`

+19/1/60+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

☐ Erreur d'exécution

☐ Erreur de compilation

☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `public`
- ☐ `protected`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

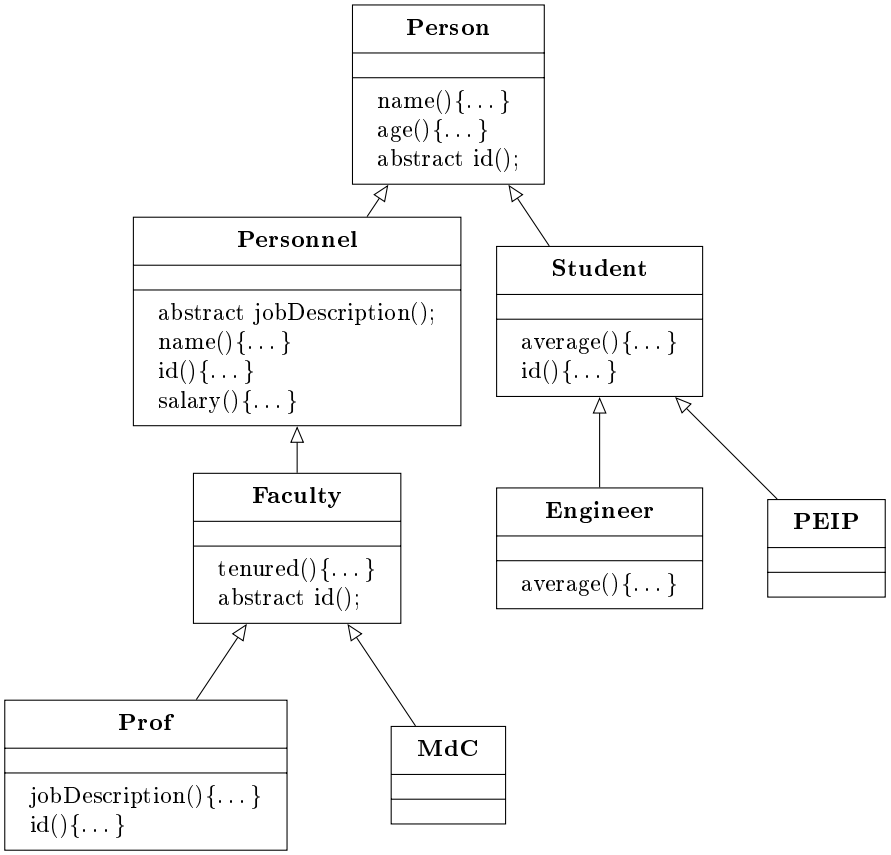
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Engineer`
- ☐ `PEIP`
- ☐ `MdC`
- ☐ `Prof`
- ☐ `Personnel`
- ☐ `Person`
- ☐ `Student`
- ☐ `Faculty`

+20/1/50+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche "Done"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ `private`
- ☐ `protected`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `whatever` ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `whatever` ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

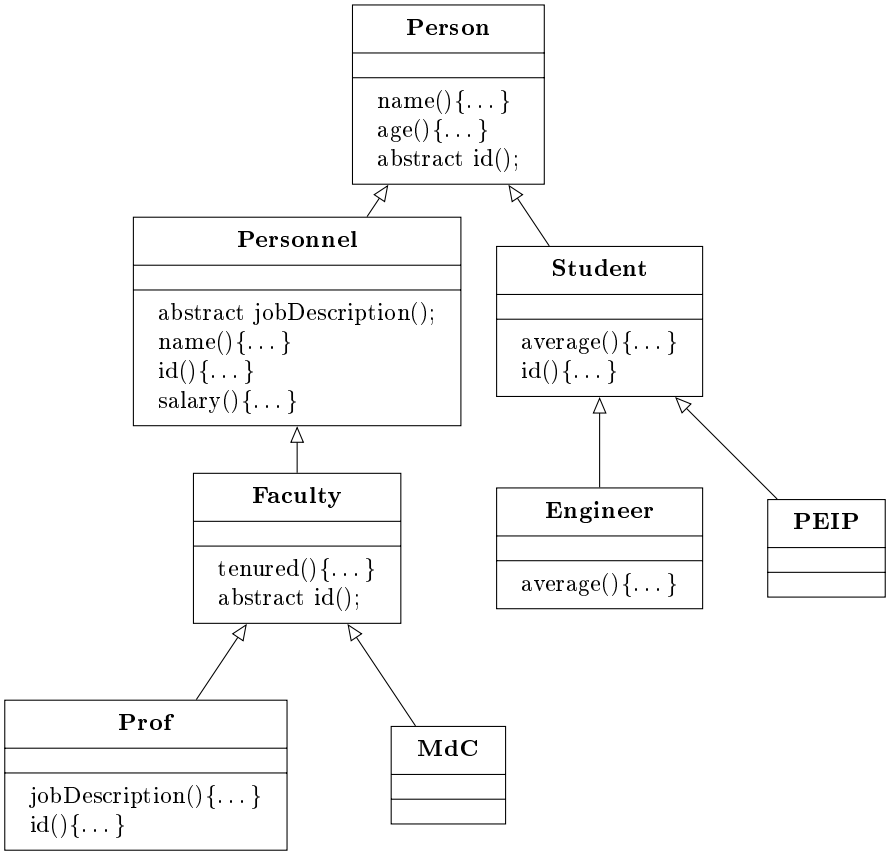
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

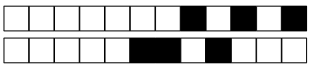
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `MdC`
- ☐ `Engineer`
- ☐ `Personnel`
- ☐ `Faculty`
- ☐ `Person`
- ☐ `Prof`
- ☐ `Student`
- ☐ `PEIP`



+21/1/40+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

☐ Affiche "Done"

☐ Erreur d'exécution

☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ package-private
- ☐ `protected`
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

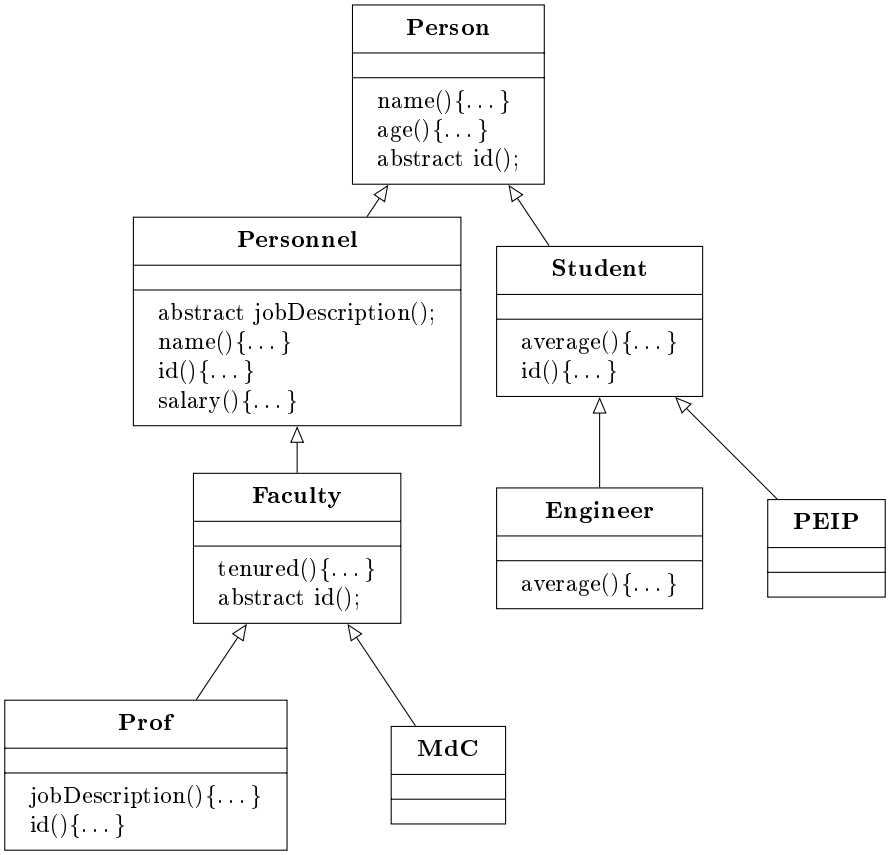
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

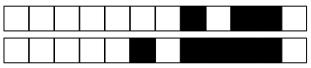
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Personnel`
- ☐ `PEIP`
- ☐ `Faculty`
- ☐ `Person`
- ☐ `Prof`
- ☐ `Engineer`
- ☐ `Student`
- ☐ `MdC`



+22/1/30+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Done"
- ☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Done"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `protected`
- ☐ `public`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

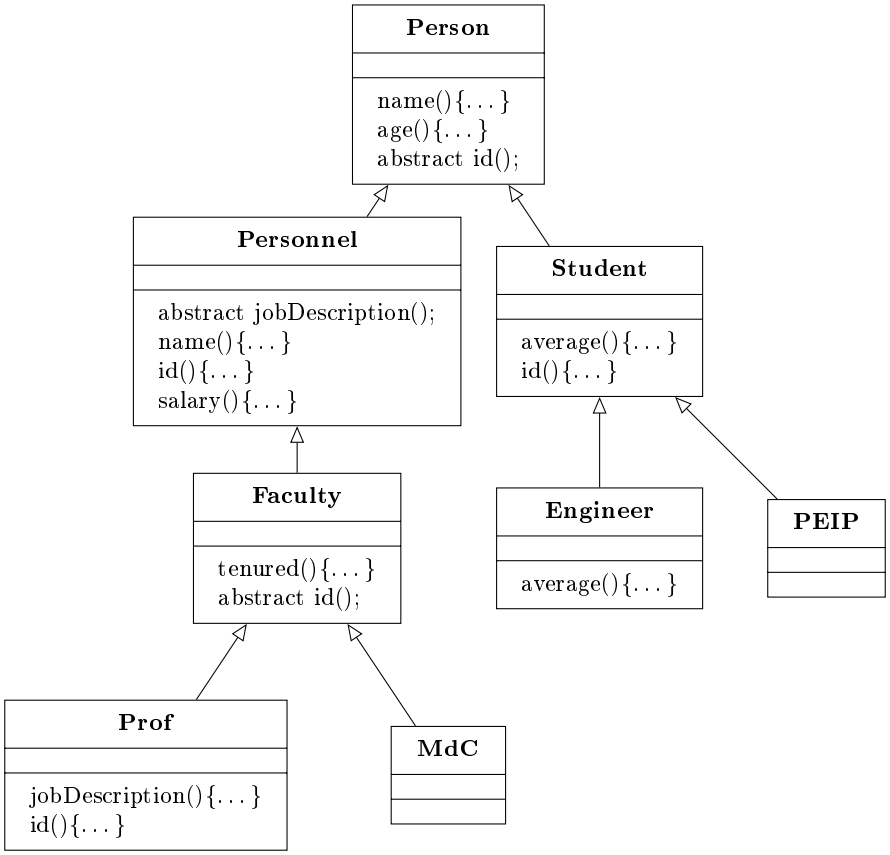
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

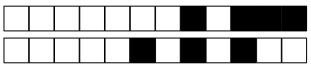
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Engineer`
- ☐ `Person`
- ☐ `Personnel`
- ☐ `Prof`
- ☐ `MdC`
- ☐ `PEIP`
- ☐ `Faculty`
- ☐ `Student`



+23/1/20+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Done"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ `protected`
- ☐ package-private
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe abstraite

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

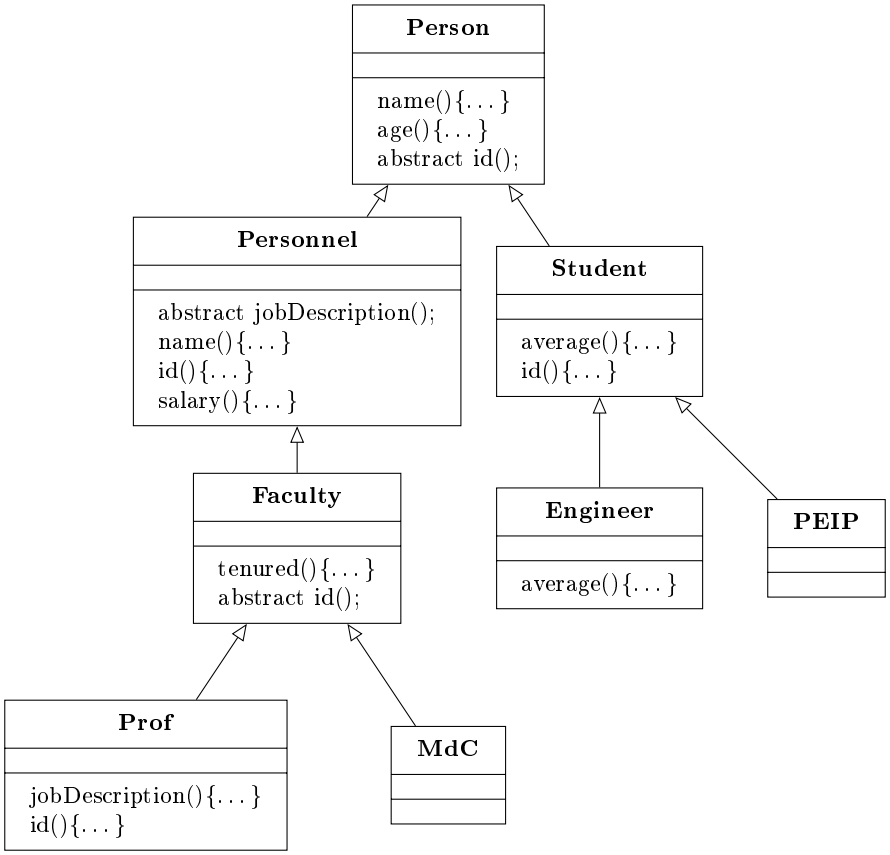
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

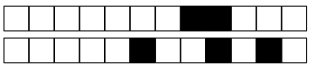
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `MdC`
- ☐ `Personnel`
- ☐ `Person`
- ☐ `Prof`
- ☐ `Student`
- ☐ `Engineer`
- ☐ `Faculty`
- ☐ `PEIP`



+24/1/10+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Done"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ package-private
- ☐ `public`
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `whatever` ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `whatever` ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

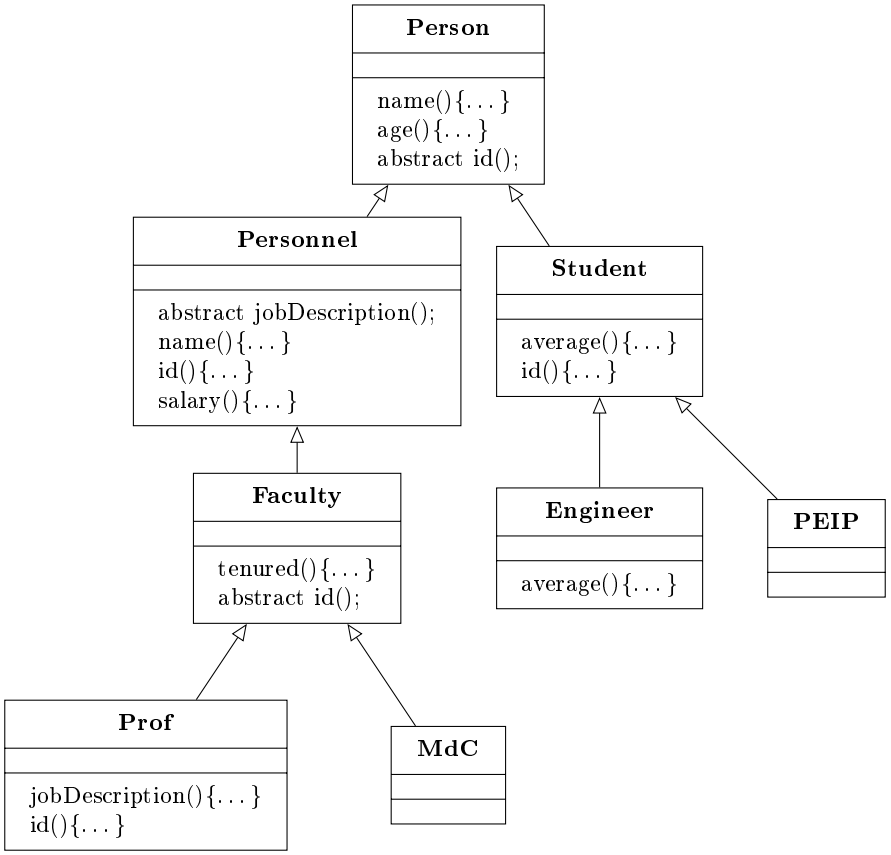
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `MdC`
- ☐ `Personnel`
- ☐ `Student`
- ☐ `Engineer`
- ☐ `PEIP`
- ☐ `Faculty`
- ☐ `Person`
- ☐ `Prof`

+25/1/60+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `private`
- ☐ `public`
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des méthodes privées

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

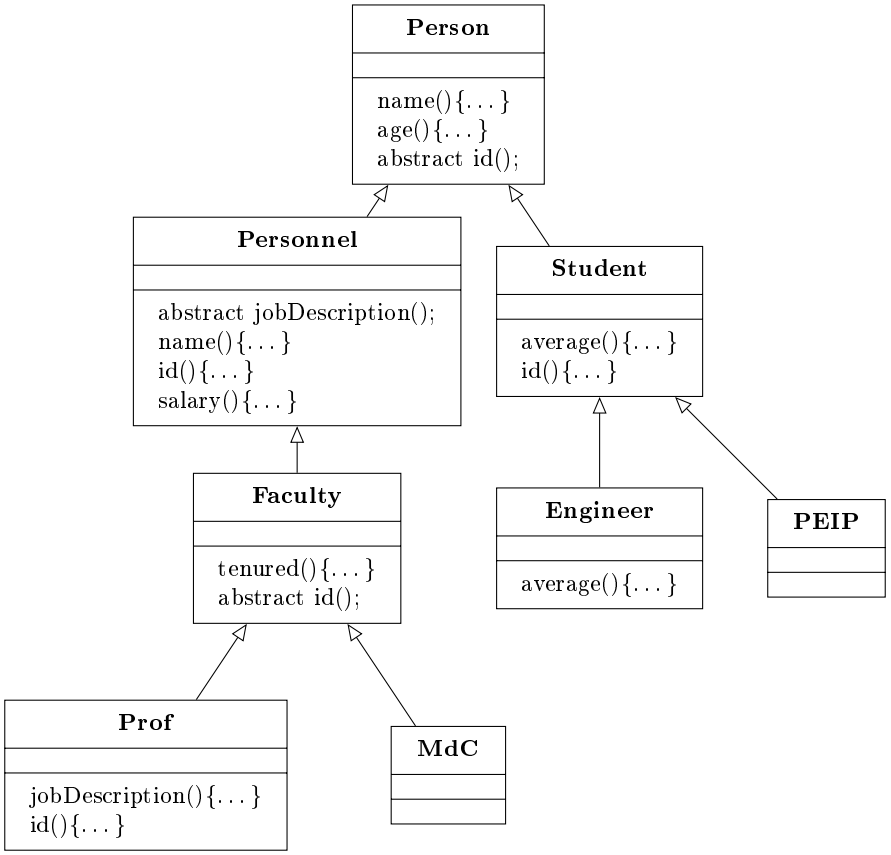
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `MdC`
- ☐ `Faculty`
- ☐ `Engineer`
- ☐ `Person`
- ☐ `Student`
- ☐ `PEIP`
- ☐ `Prof`
- ☐ `Personnel`

+26/1/50+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

☐ Erreur de compilation

☐ Affiche "Done"

☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ `protected`
- ☐ package-private
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `whatever` ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `whatever` ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

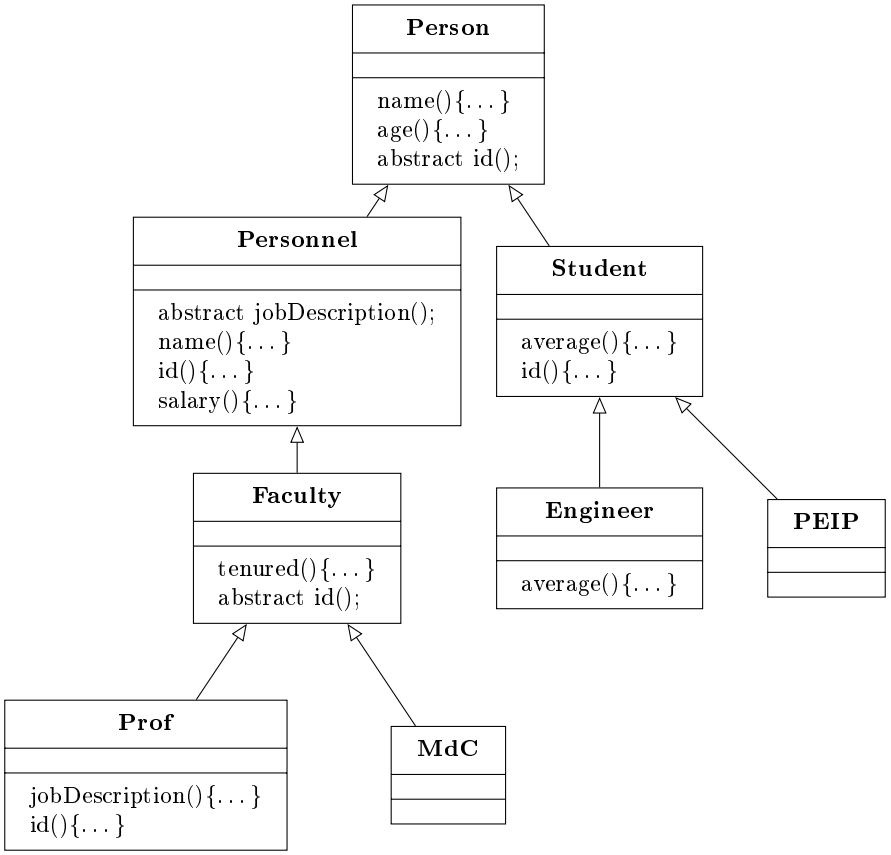
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

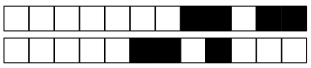
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `PEIP`
- ☐ `MdC`
- ☐ `Engineer`
- ☐ `Student`
- ☐ `Person`
- ☐ `Personnel`
- ☐ `Faculty`
- ☐ `Prof`



+27/1/40+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Done"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ `private`
- ☐ package-private
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

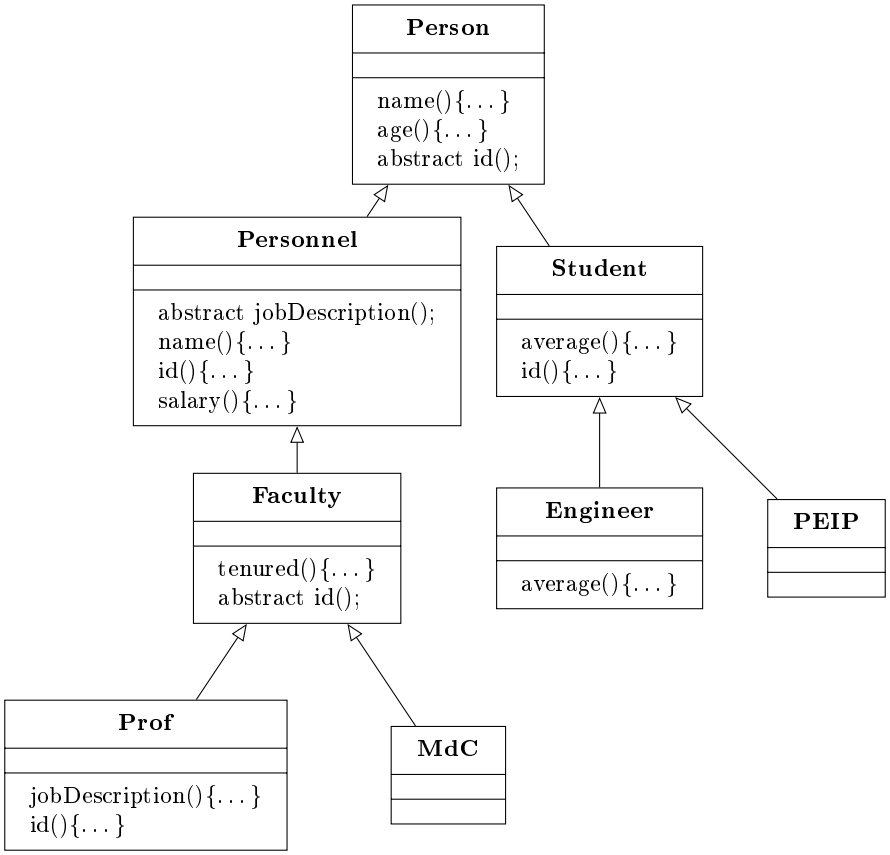
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

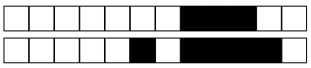
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Personnel`
- ☐ `Faculty`
- ☐ `Engineer`
- ☐ `Student`
- ☐ `Prof`
- ☐ `MdC`
- ☐ `PEIP`
- ☐ `Person`



+28/1/30+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

☐ Erreur de compilation

☐ Erreur d'exécution

☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ `public`
- ☐ `private`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

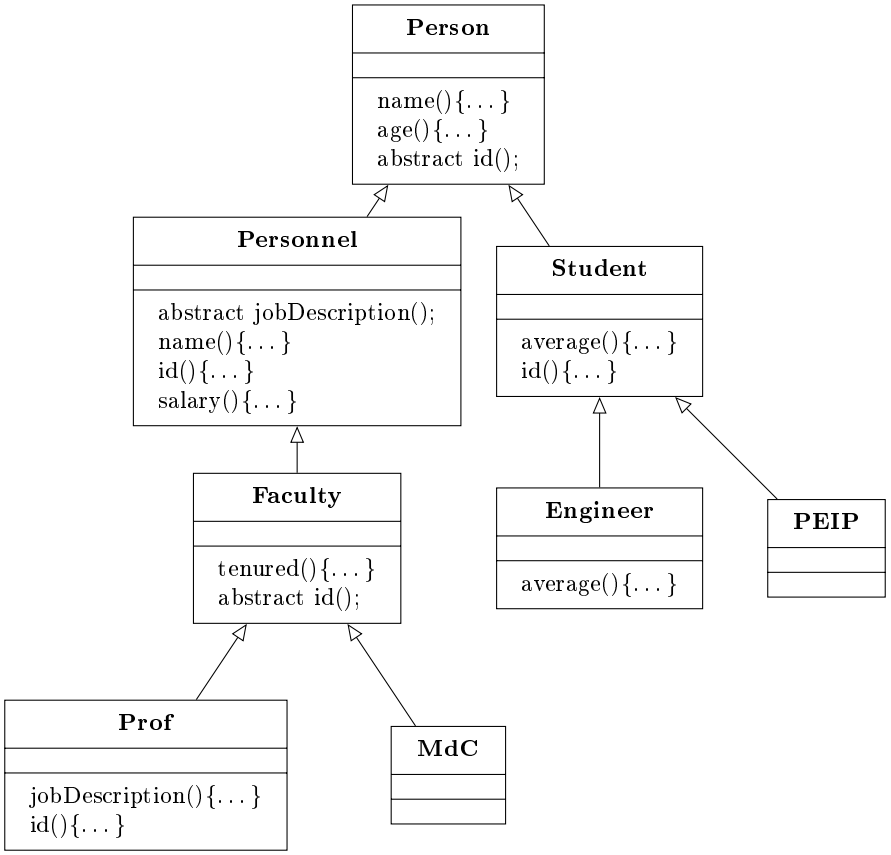
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

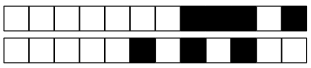
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Student`
- ☐ `MdC`
- ☐ `Engineer`
- ☐ `Faculty`
- ☐ `PEIP`
- ☐ `Prof`
- ☐ `Person`
- ☐ `Personnel`



+29/1/20+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `private`
- ☐ `public`
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

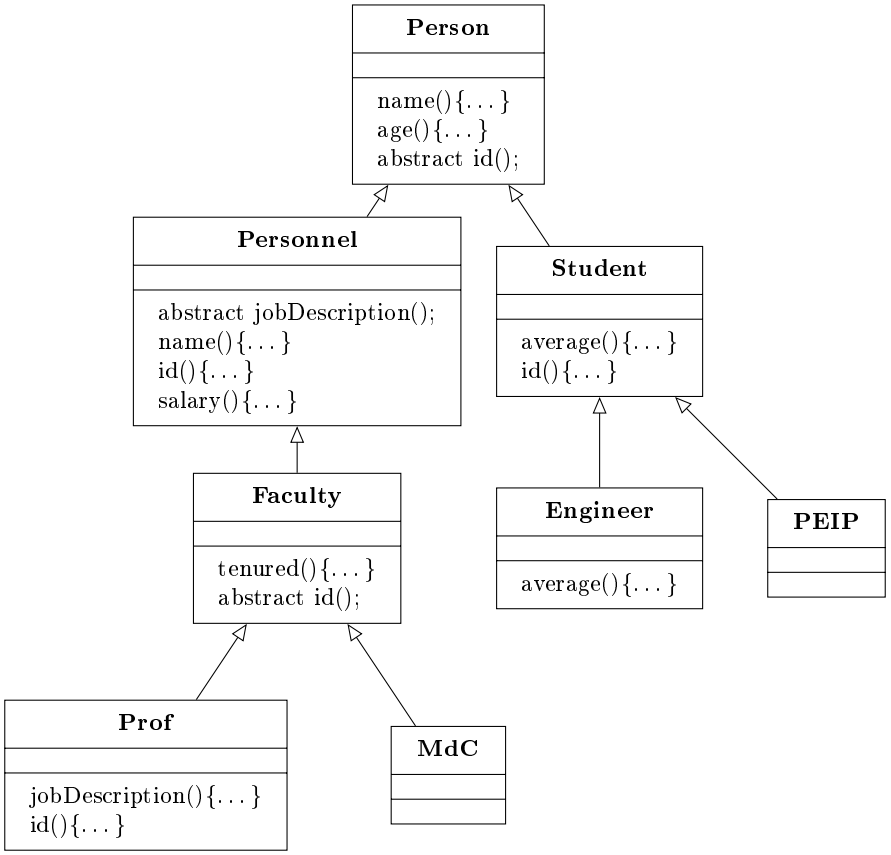
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

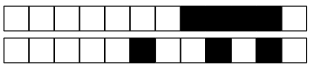
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Prof`
- ☐ `Faculty`
- ☐ `Person`
- ☐ `MdC`
- ☐ `Student`
- ☐ `Engineer`
- ☐ `PEIP`
- ☐ `Personnel`



+30/1/10+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

☐ Erreur d'exécution

☐ Erreur de compilation

☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Done"
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

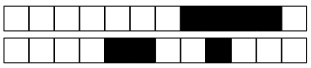
```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ package-private
- ☐ `public`
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`



Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `whatever` ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `whatever` ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

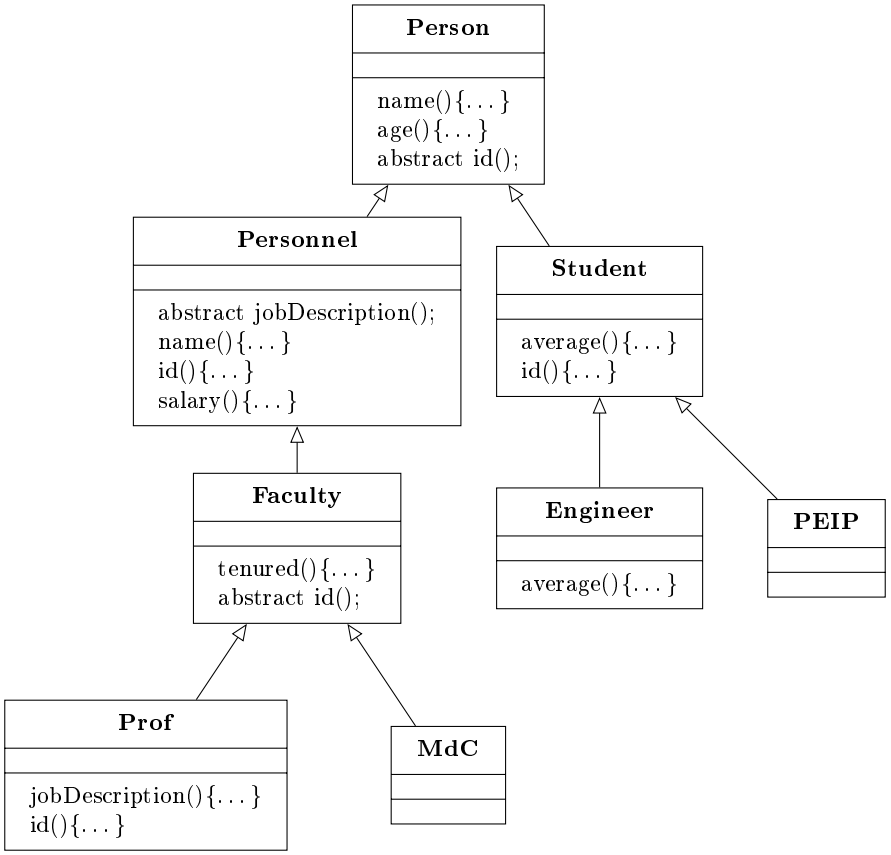
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Prof`
- ☐ `MdC`
- ☐ `Personnel`
- ☐ `Engineer`
- ☐ `Person`
- ☐ `Student`
- ☐ `PEIP`
- ☐ `Faculty`

+31/1/60+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ package-private
- ☐ `protected`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

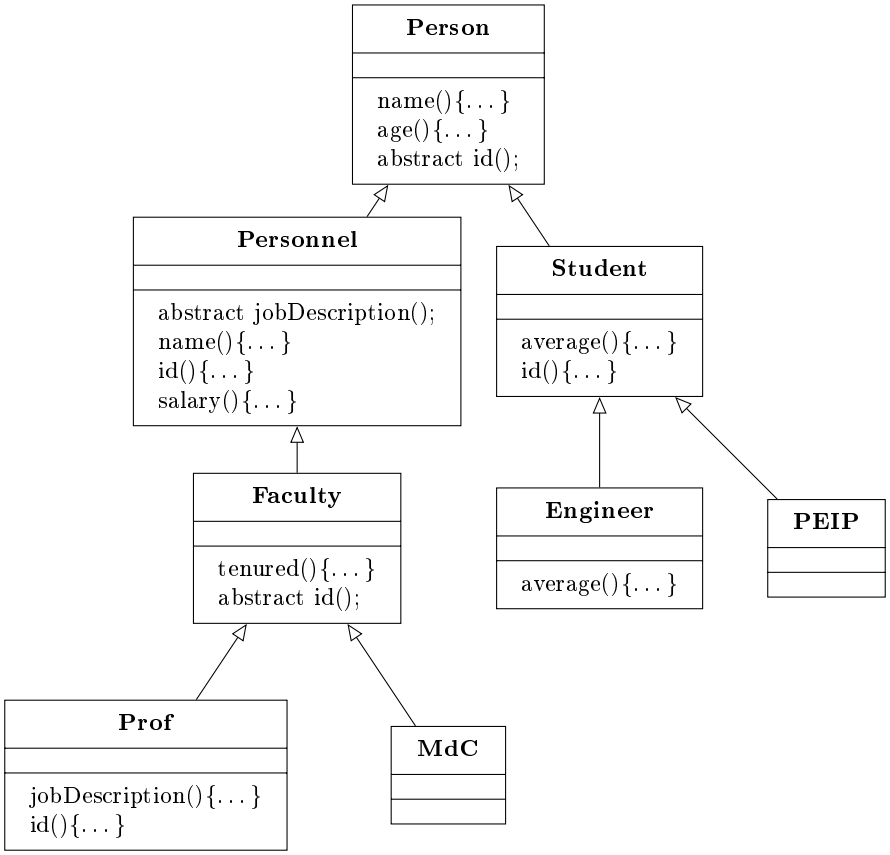
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Engineer`
- ☐ `Personnel`
- ☐ `MdC`
- ☐ `Student`
- ☐ `Person`
- ☐ `PEIP`
- ☐ `Faculty`
- ☐ `Prof`

+32/1/50+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Done"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ `protected`
- ☐ `private`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

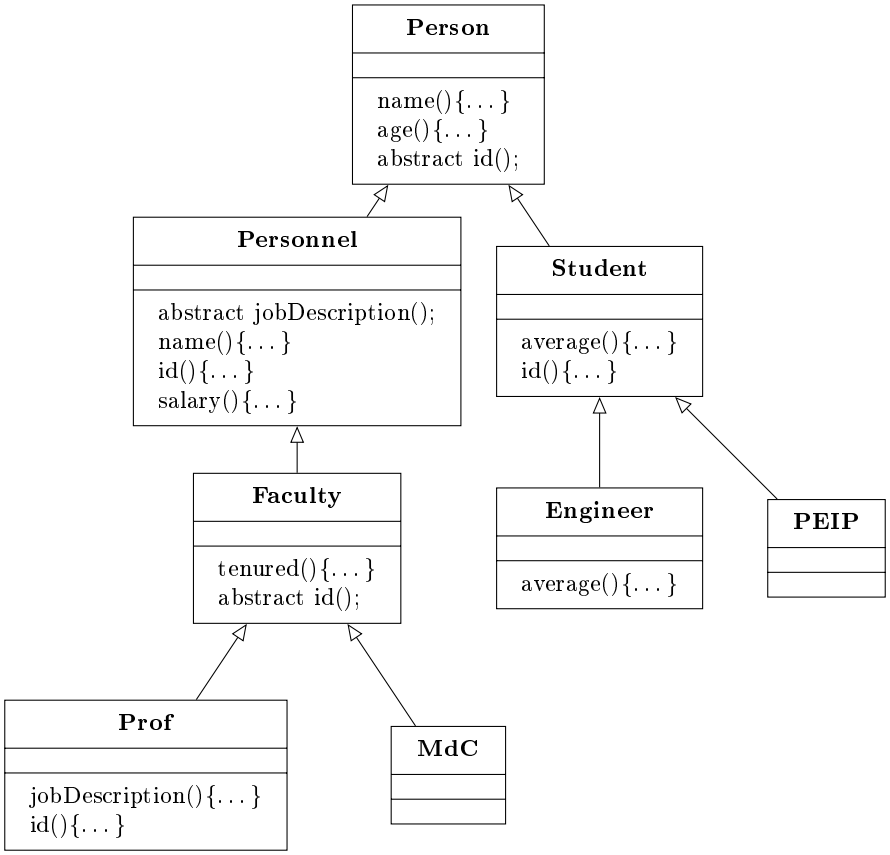
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `PEIP`
- ☐ `Engineer`
- ☐ `Personnel`
- ☐ `Faculty`
- ☐ `Person`
- ☐ `MdC`
- ☐ `Student`
- ☐ `Prof`

+33/1/40+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `private`
- ☐ `public`
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

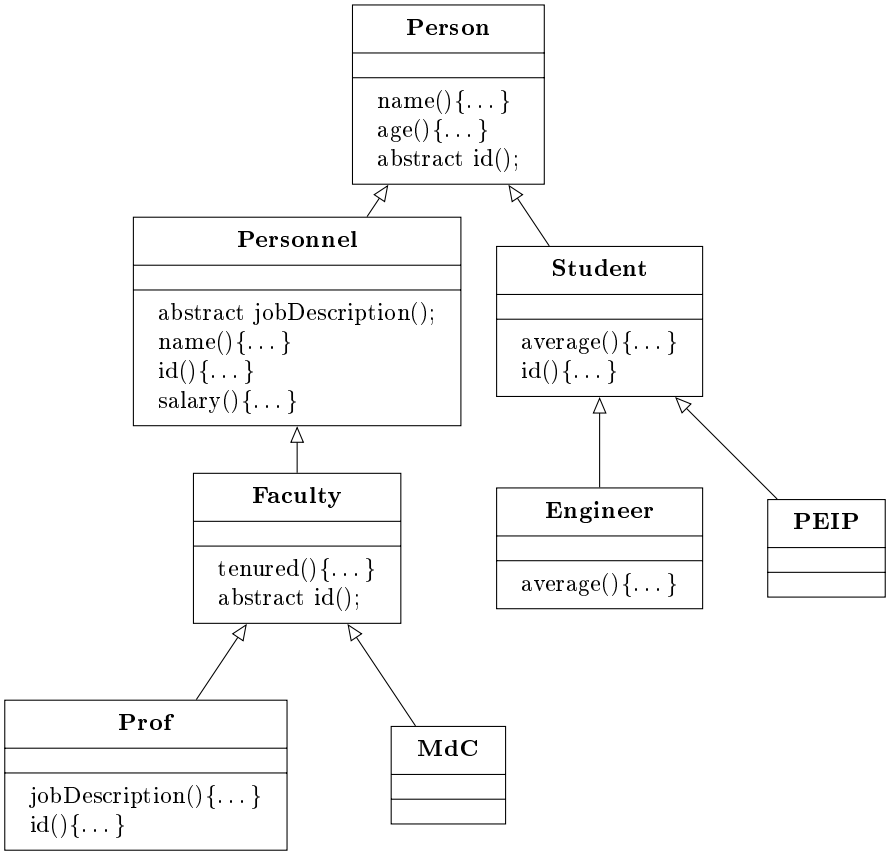
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `MdC`
- ☐ `Engineer`
- ☐ `Prof`
- ☐ `Student`
- ☐ `PEIP`
- ☐ `Person`
- ☐ `Personnel`
- ☐ `Faculty`

+34/1/30+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

☐ Erreur de compilation

☐ Affiche "Done"

☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `private`
- ☐ `protected`
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

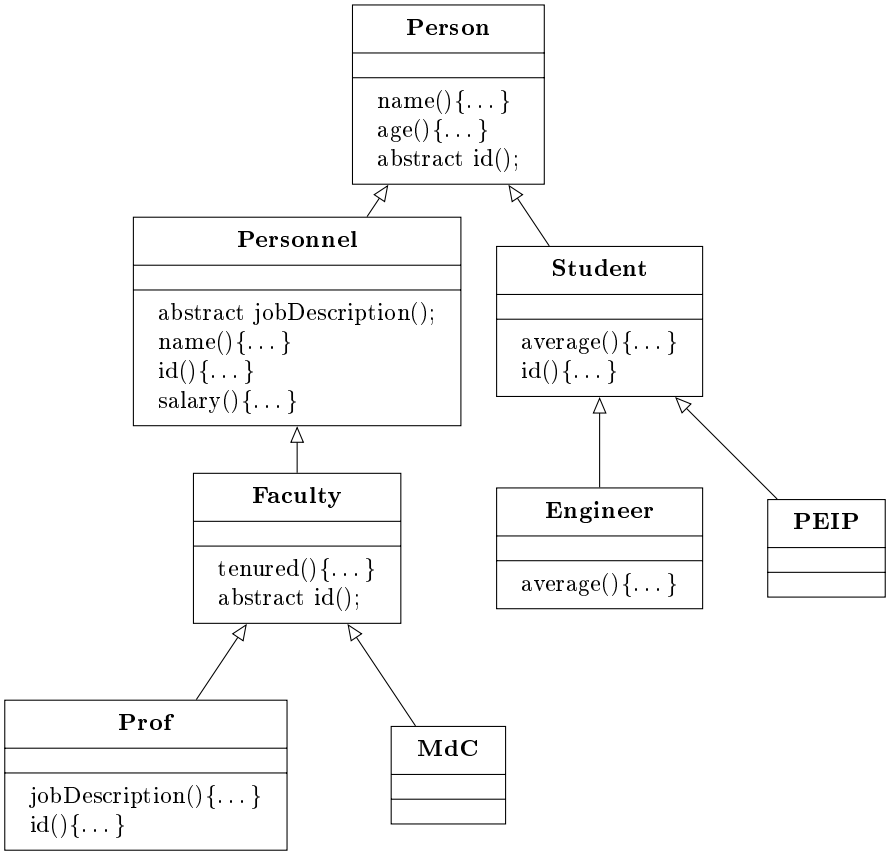
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `PEIP`
- ☐ `Person`
- ☐ `Prof`
- ☐ `Engineer`
- ☐ `Personnel`
- ☐ `Faculty`
- ☐ `MdC`
- ☐ `Student`

+35/1/20+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `public`
- ☐ `private`
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

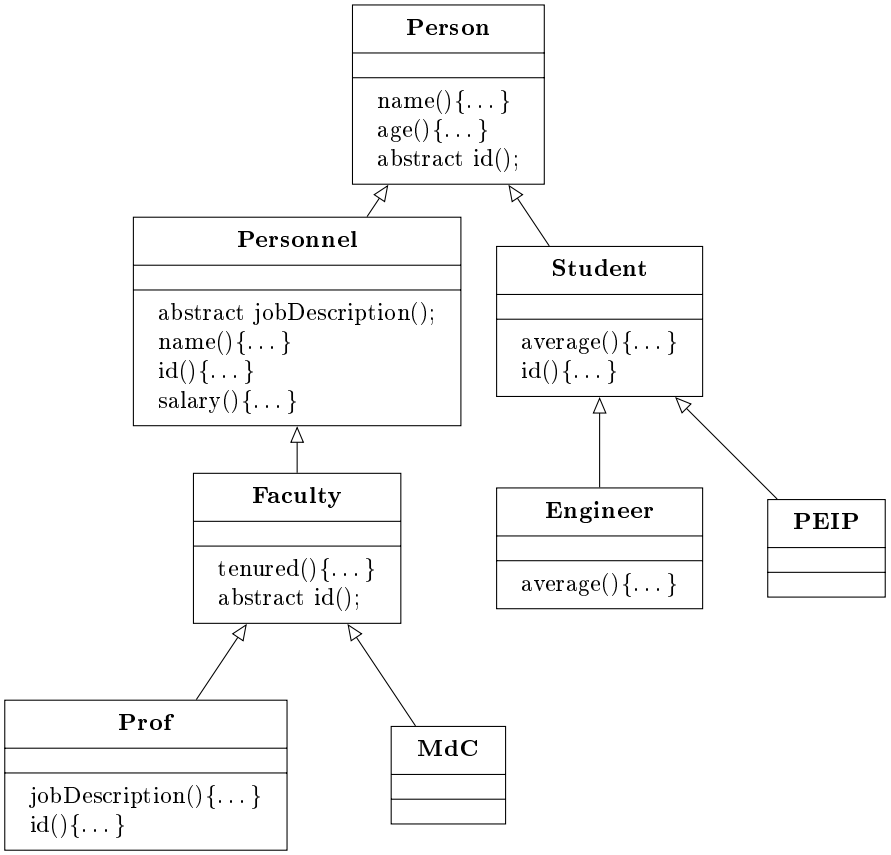
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

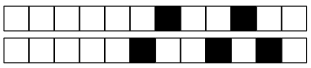
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Person`
- ☐ `Personnel`
- ☐ `PEIP`
- ☐ `Prof`
- ☐ `Engineer`
- ☐ `Faculty`
- ☐ `MdC`
- ☐ `Student`



+36/1/10+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

☐ Erreur d'exécution

☐ Erreur de compilation

☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ `protected`
- ☐ package-private
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

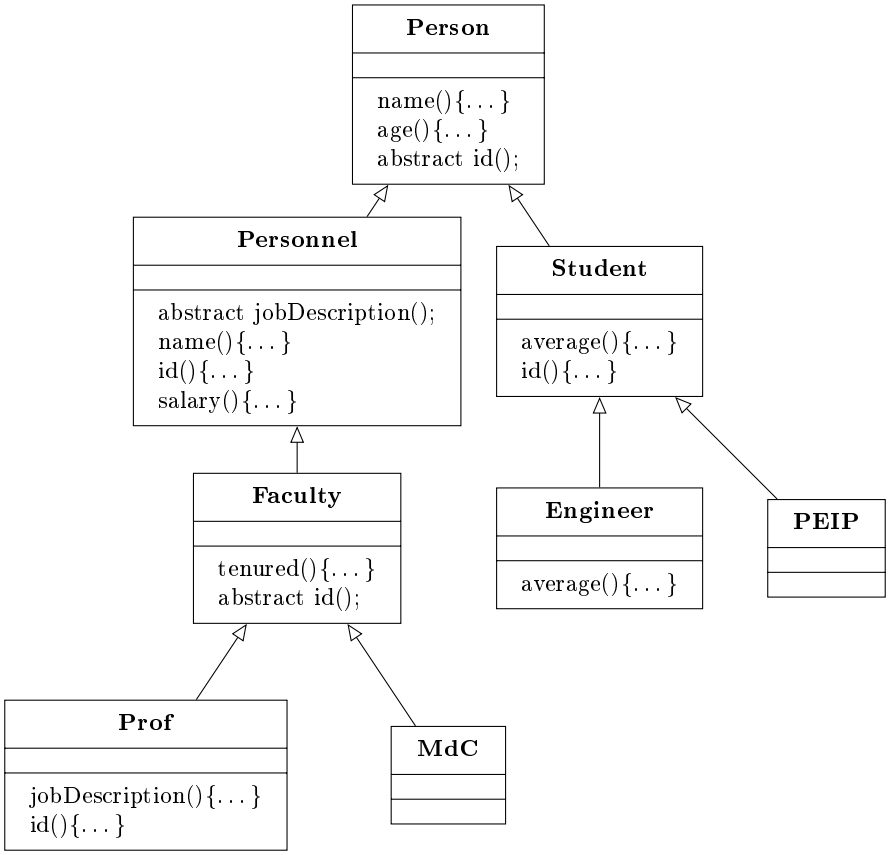
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Personnel`
- ☐ `Person`
- ☐ `Faculty`
- ☐ `Prof`
- ☐ `Student`
- ☐ `MdC`
- ☐ `PEIP`
- ☐ `Engineer`

+37/1/60+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Done"
- ☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ package-private
- ☐ `public`
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

☐ `whatever` ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `whatever` ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `whatever` ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

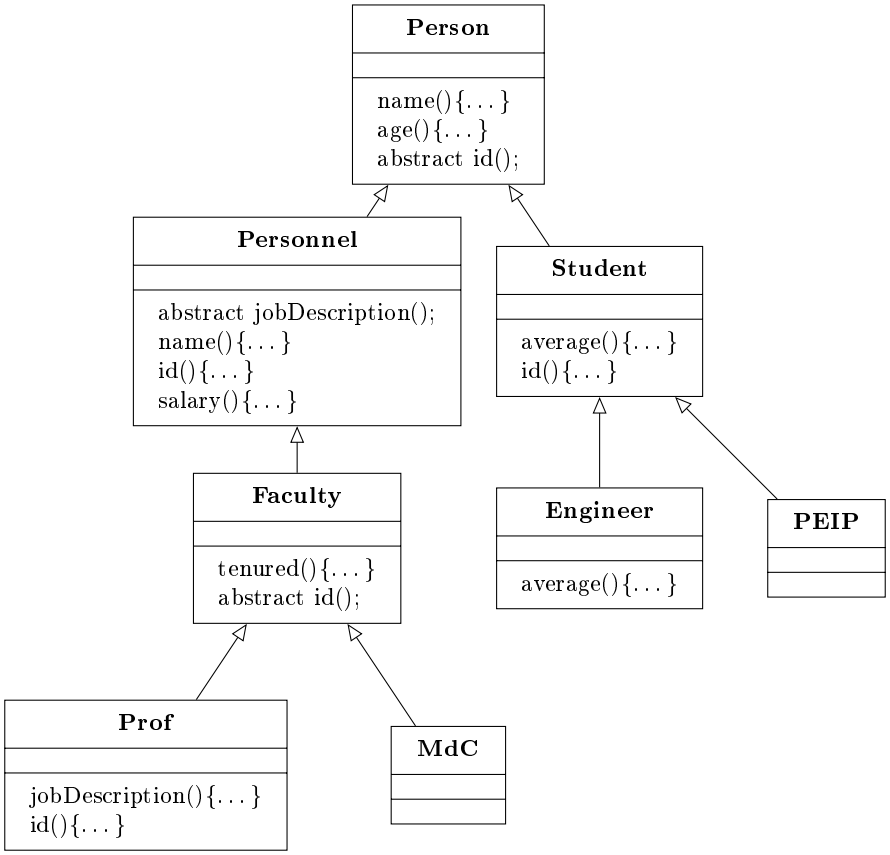
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Faculty`
- ☐ `MdC`
- ☐ `Engineer`
- ☐ `PEIP`
- ☐ `Person`
- ☐ `Prof`
- ☐ `Personnel`
- ☐ `Student`

+38/1/50+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche "Done"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Done"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ `protected`
- ☐ `private`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

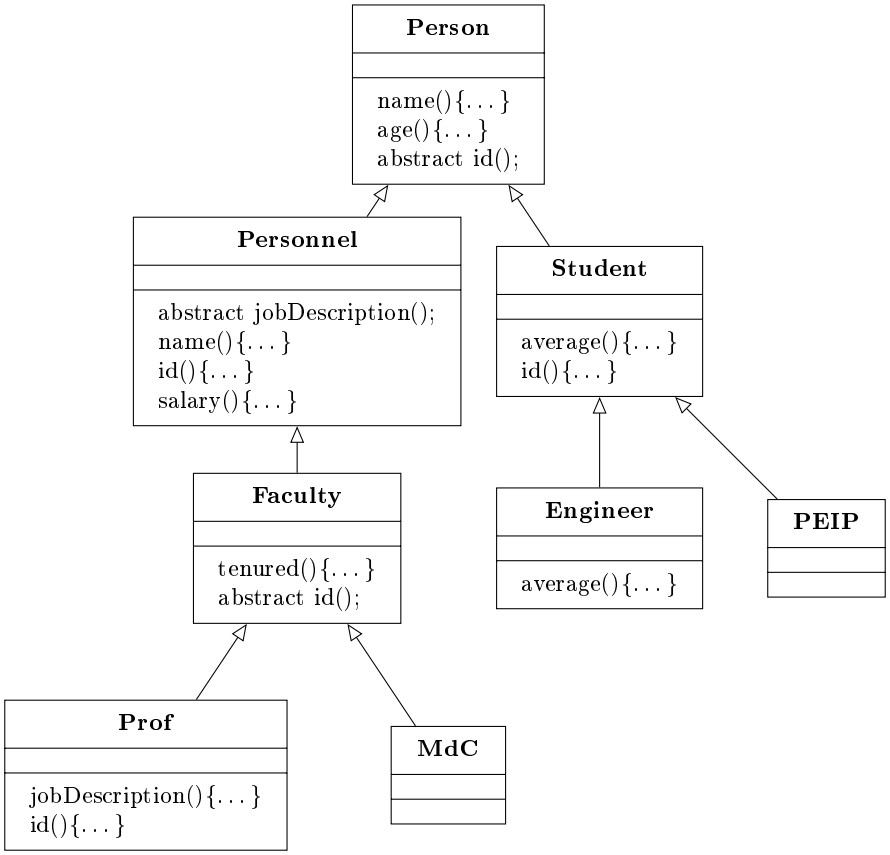
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Student`
- ☐ `Personnel`
- ☐ `MdC`
- ☐ `Engineer`
- ☐ `Faculty`
- ☐ `Prof`
- ☐ `PEIP`
- ☐ `Person`

+39/1/40+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Done"
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ `protected`
- ☐ package-private
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

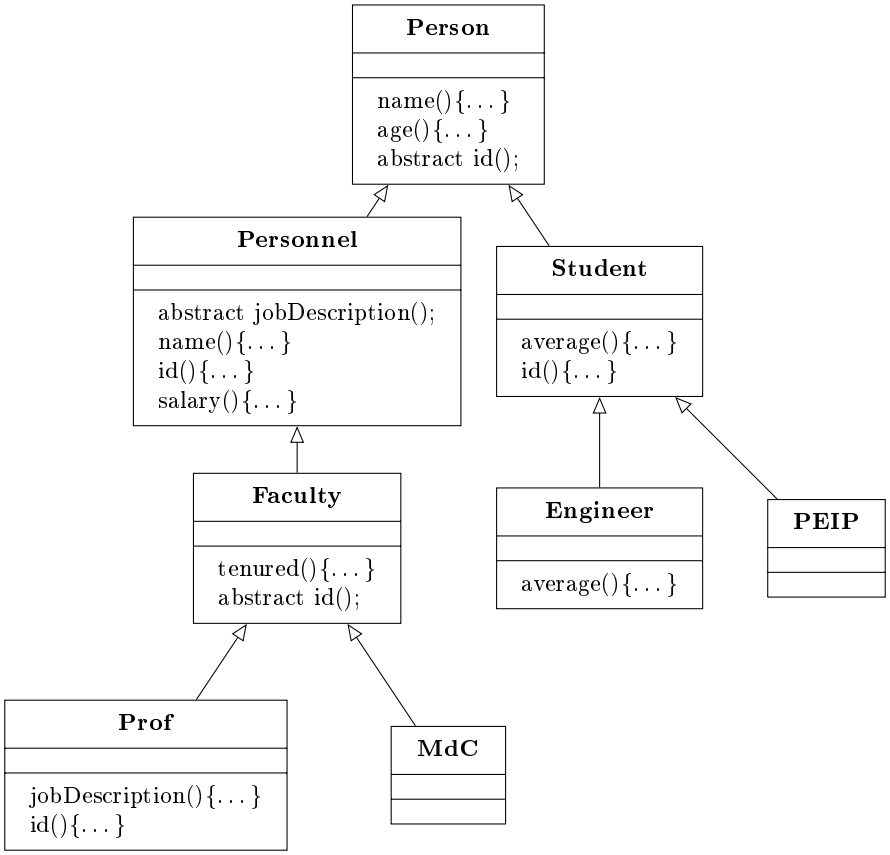
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Faculty`
- ☐ `Prof`
- ☐ `Personnel`
- ☐ `PEIP`
- ☐ `Engineer`
- ☐ `MdC`
- ☐ `Student`
- ☐ `Person`

+40/1/30+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

☐ Erreur de compilation

☐ Affiche "Done"

☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `private`
- ☐ `protected`
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

☐ `something` ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `something` ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `whatever` ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des méthodes privées

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

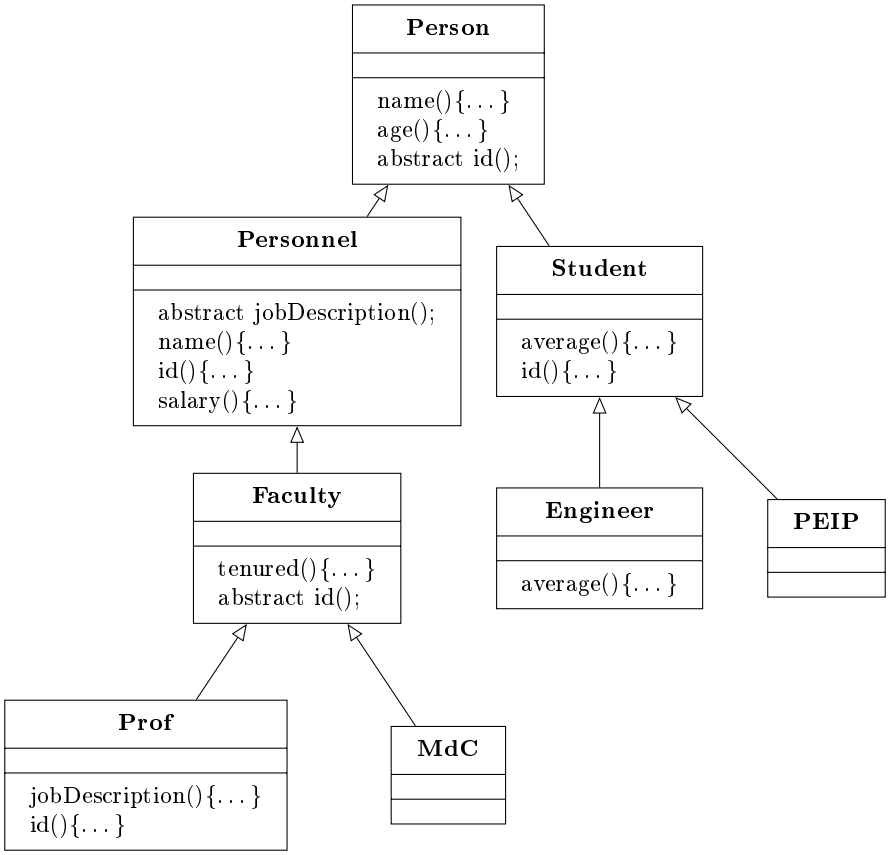
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Person`
- ☐ `Personnel`
- ☐ `Prof`
- ☐ `Engineer`
- ☐ `Student`
- ☐ `PEIP`
- ☐ `Faculty`
- ☐ `MdC`

+41/1/20+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Done"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ package-private
- ☐ `public`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des méthodes privées

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

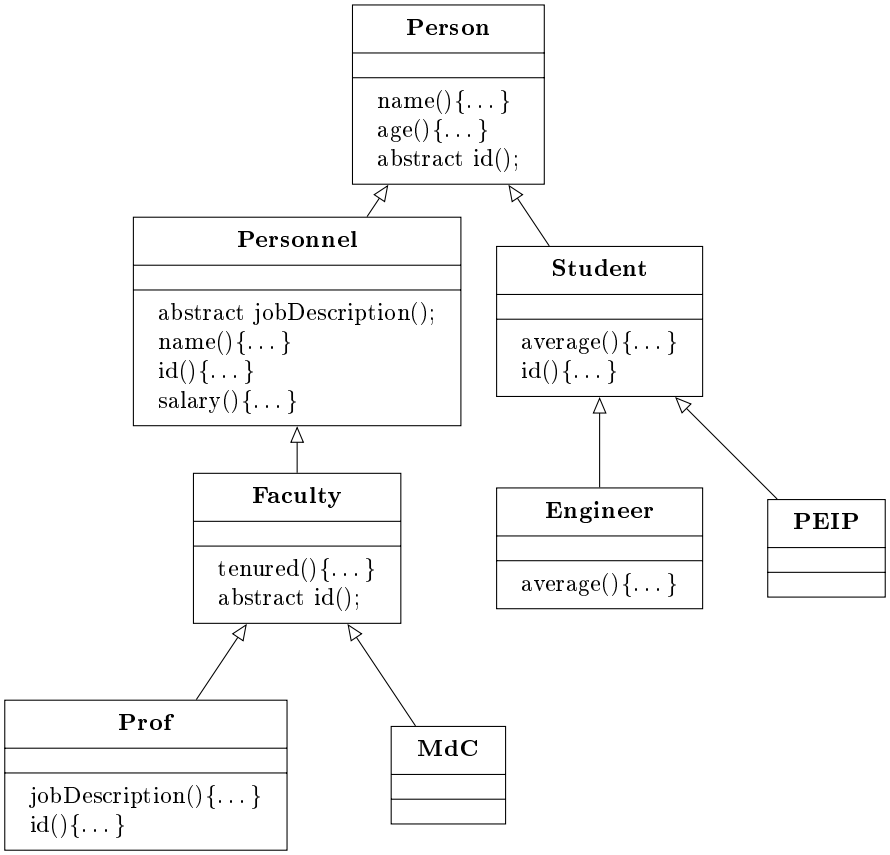
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

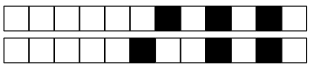
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `MdC`
- ☐ `PEIP`
- ☐ `Engineer`
- ☐ `Personnel`
- ☐ `Faculty`
- ☐ `Student`
- ☐ `Person`
- ☐ `Prof`



+42/1/10+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

☐ Erreur d'exécution

☐ Affiche "Done"

☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ `public`
- ☐ package-private
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

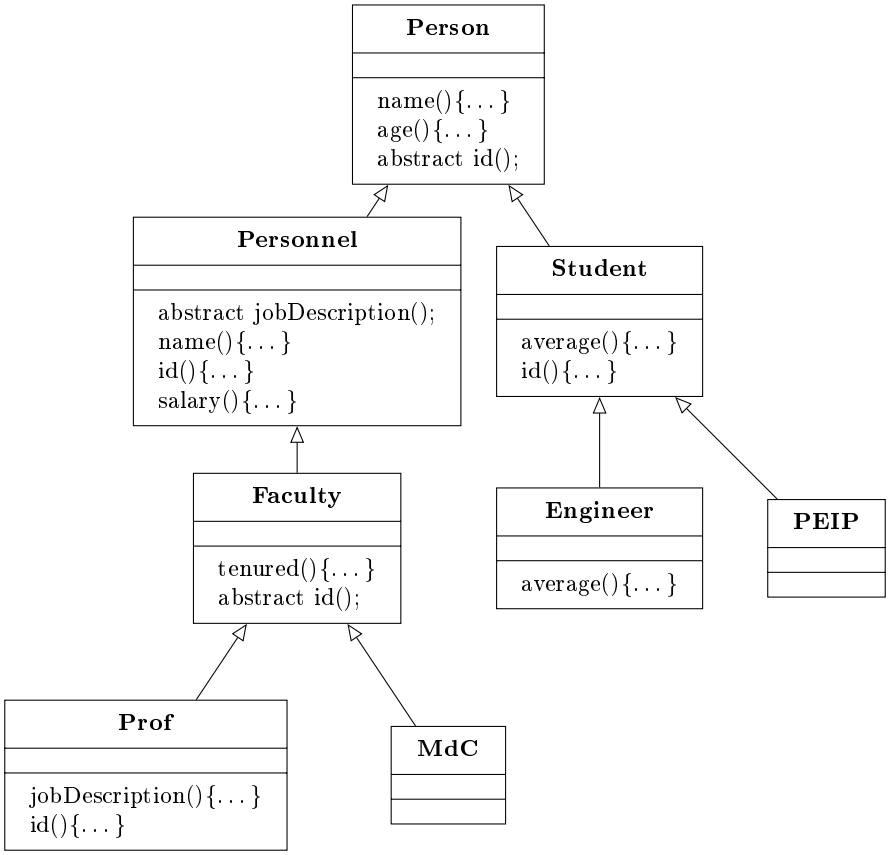
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `MdC`
- ☐ `Person`
- ☐ `Personnel`
- ☐ `Engineer`
- ☐ `Prof`
- ☐ `Student`
- ☐ `Faculty`
- ☐ `PEIP`

+43/1/60+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

☐ Affiche "Done"

☐ Erreur de compilation

☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Done"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `public`
- ☐ `protected`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `something` ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `something` ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

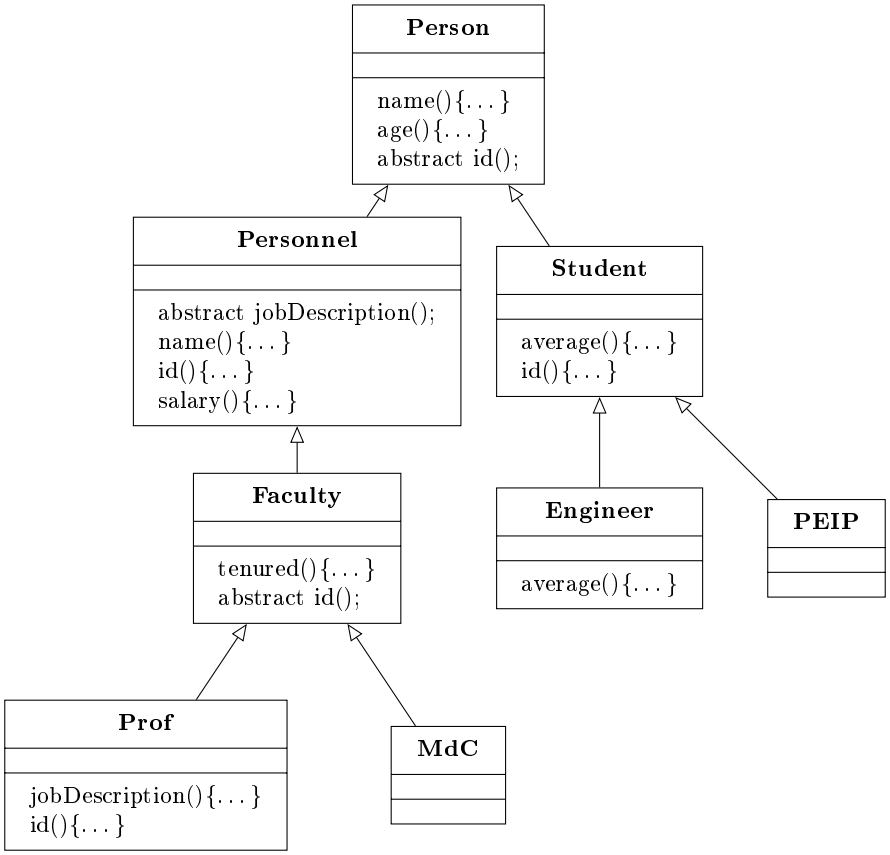
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Faculty`
- ☐ `Person`
- ☐ `Prof`
- ☐ `Student`
- ☐ `Engineer`
- ☐ `Personnel`
- ☐ `MdC`
- ☐ `PEIP`

+44/1/50+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche "Done"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ `public`
- ☐ package-private
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `whatever` ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `whatever` ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut implémenter une interface

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

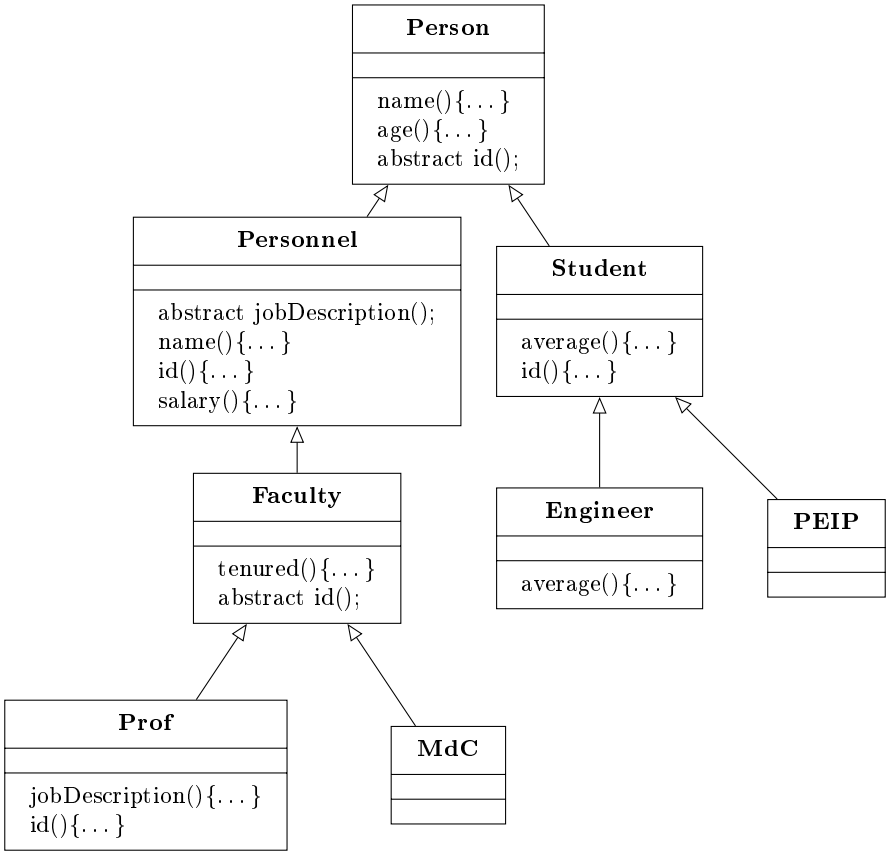
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Prof`
- ☐ `Engineer`
- ☐ `PEIP`
- ☐ `Person`
- ☐ `MdC`
- ☐ `Faculty`
- ☐ `Personnel`
- ☐ `Student`

+45/1/40+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

☐ Affiche "Done"

☐ Erreur de compilation

☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `private`
- ☐ `public`
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

☐ `something` ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `something` ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `something` ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe abstraite peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

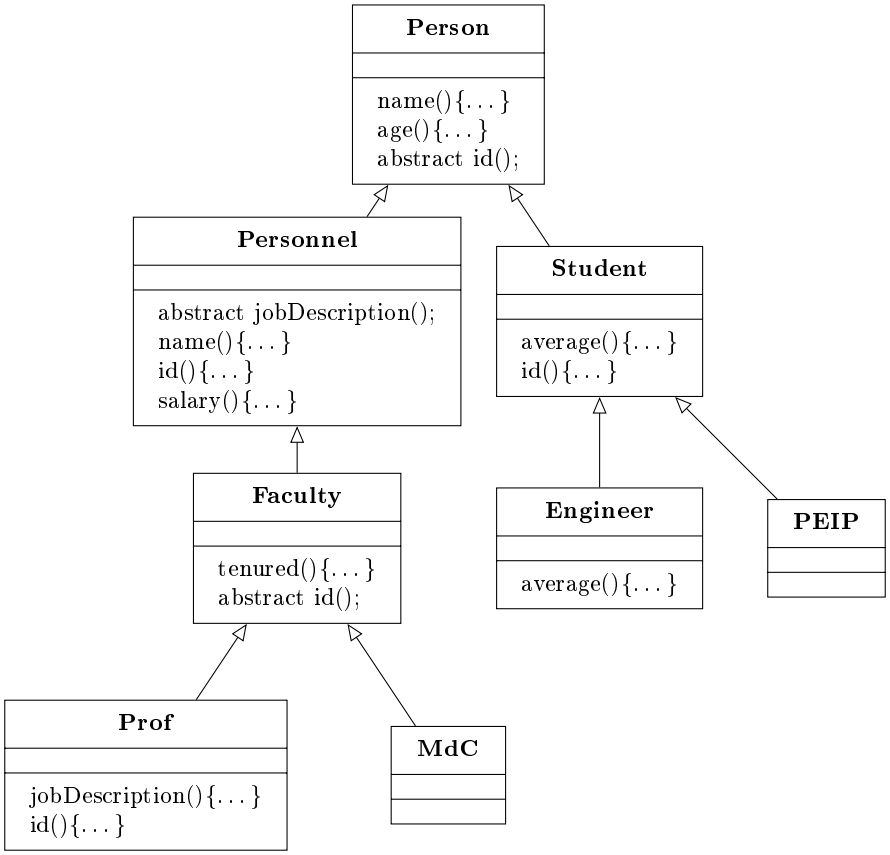
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Prof`
- ☐ `Person`
- ☐ `Student`
- ☐ `Engineer`
- ☐ `Personnel`
- ☐ `MdC`
- ☐ `Faculty`
- ☐ `PEIP`

+46/1/30+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Done"
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ `private`
- ☐ package-private
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut implémenter une interface

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

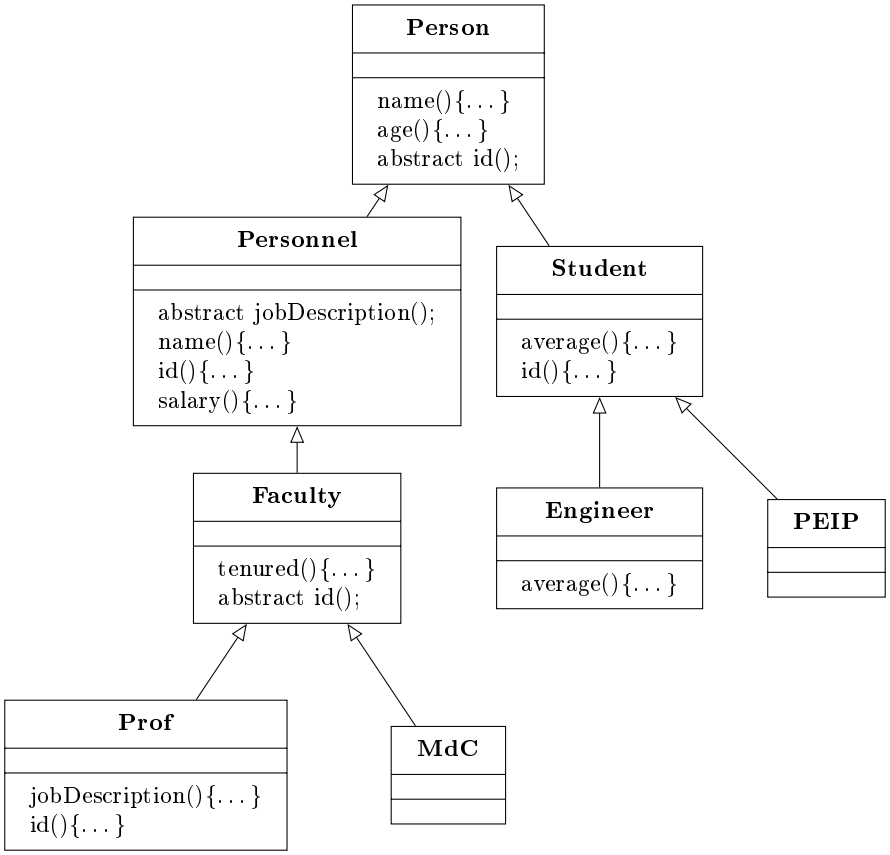
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Engineer`
- ☐ `Person`
- ☐ `Student`
- ☐ `PEIP`
- ☐ `Faculty`
- ☐ `MdC`
- ☐ `Prof`
- ☐ `Personnel`

+47/1/20+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Done"
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Done"
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ package-private
- ☐ `public`
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des méthodes privées

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

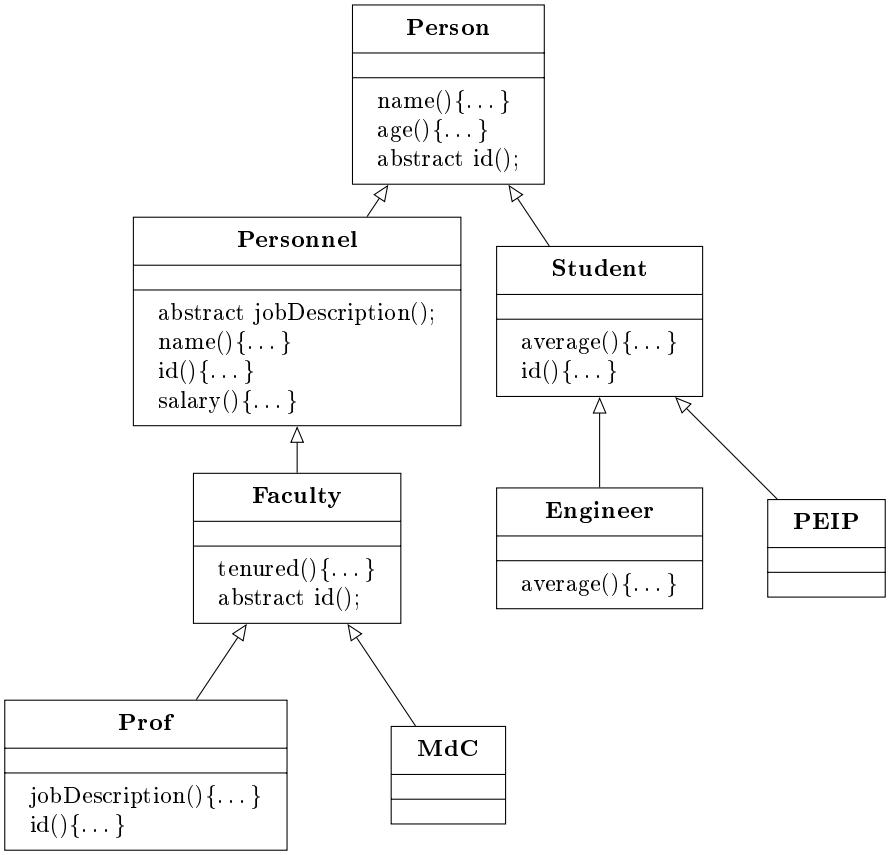
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

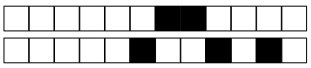
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Person`
- ☐ `Student`
- ☐ `Engineer`
- ☐ `Personnel`
- ☐ `PEIP`
- ☐ `Prof`
- ☐ `Faculty`
- ☐ `MdC`



+48/1/10+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `private`
- ☐ `public`
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

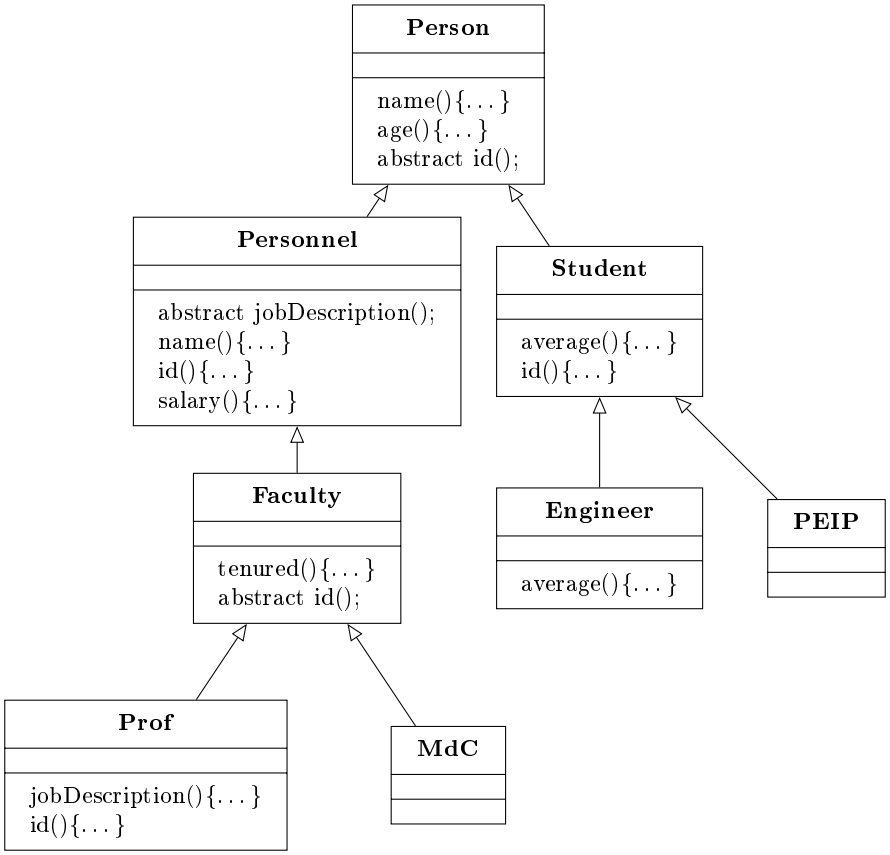
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `PEIP`
- ☐ `Personnel`
- ☐ `Prof`
- ☐ `MdC`
- ☐ `Engineer`
- ☐ `Person`
- ☐ `Student`
- ☐ `Faculty`

+49/1/60+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

☐ Erreur d'exécution

☐ Erreur de compilation

☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ package-private
- ☐ `public`
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre une classe abstraite

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

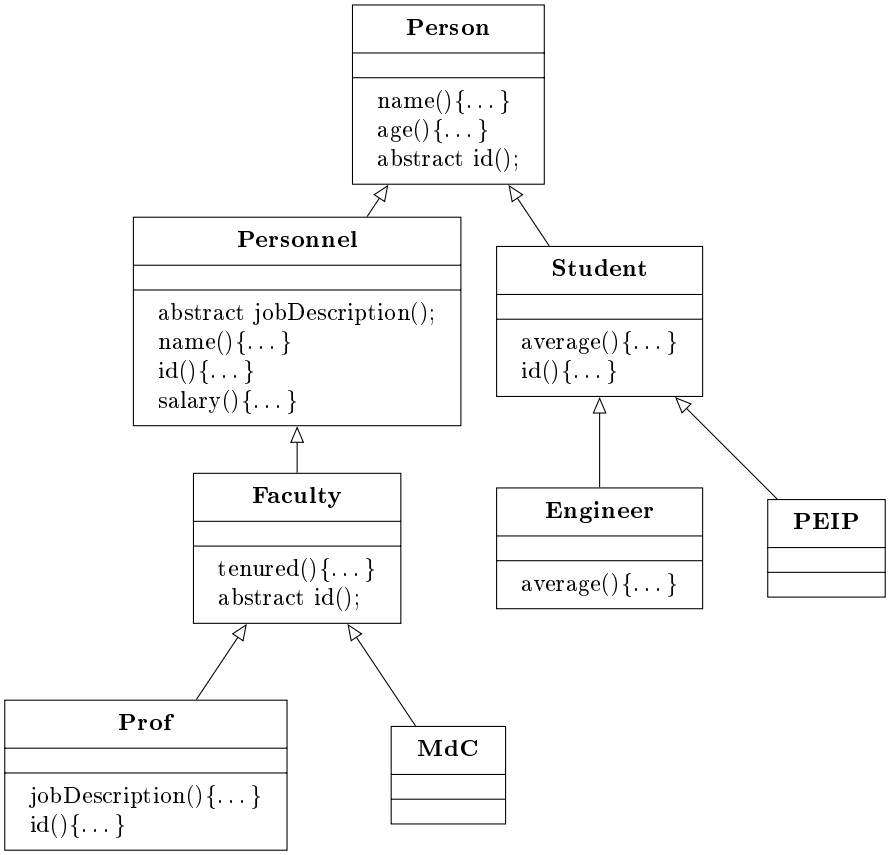
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `PEIP`
- ☐ `Engineer`
- ☐ `Person`
- ☐ `Prof`
- ☐ `MdC`
- ☐ `Student`
- ☐ `Personnel`
- ☐ `Faculty`

+50/1/50+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ package-private
- ☐ `public`
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `whatever` ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `whatever` ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

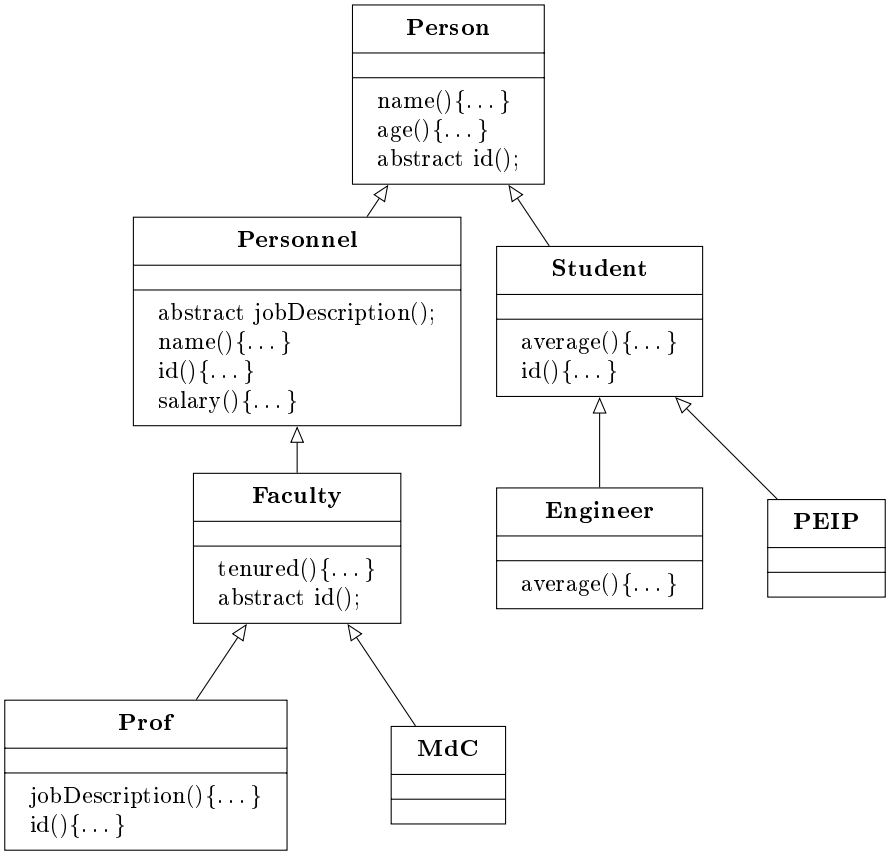
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Personnel`
- ☐ `Faculty`
- ☐ `Student`
- ☐ `Person`
- ☐ `MdC`
- ☐ `Engineer`
- ☐ `Prof`
- ☐ `PEIP`

+51/1/40+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `protected`
- ☐ `private`
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

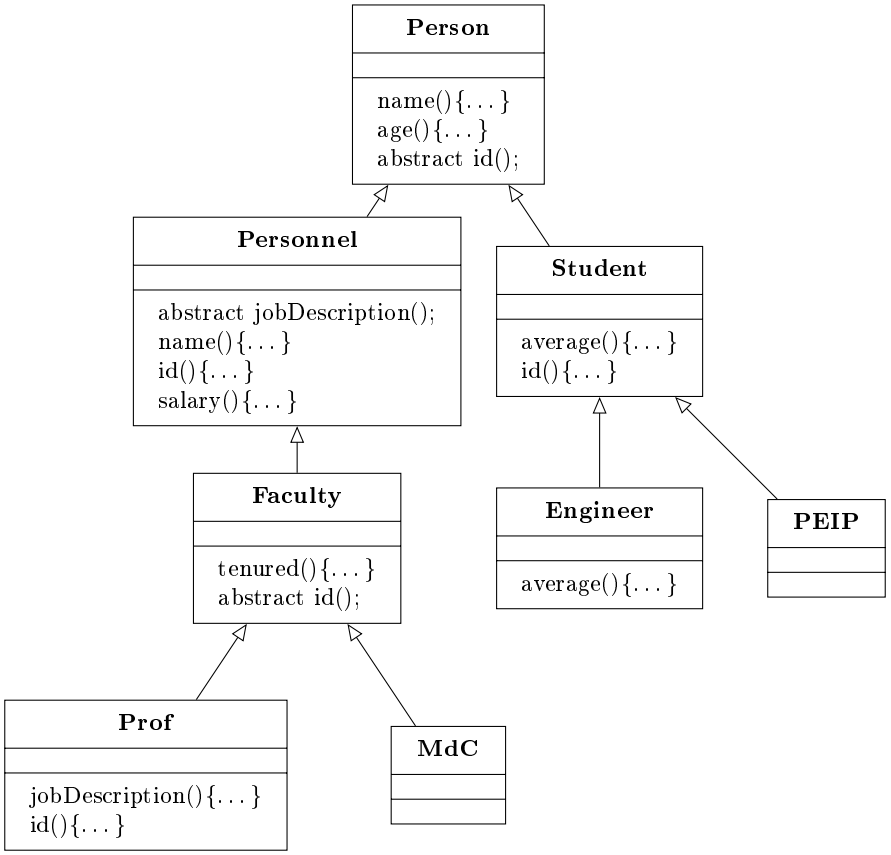
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

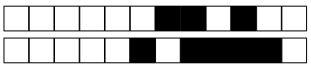
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Faculty`
- ☐ `Prof`
- ☐ `Engineer`
- ☐ `Student`
- ☐ `Personnel`
- ☐ `PEIP`
- ☐ `MdC`
- ☐ `Person`



+52/1/30+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ *indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.*
Dans tout le code, les `package` *et les* `import` *sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.*

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

☐ Erreur d'exécution

☐ Erreur de compilation

☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `private`
- ☐ `protected`
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

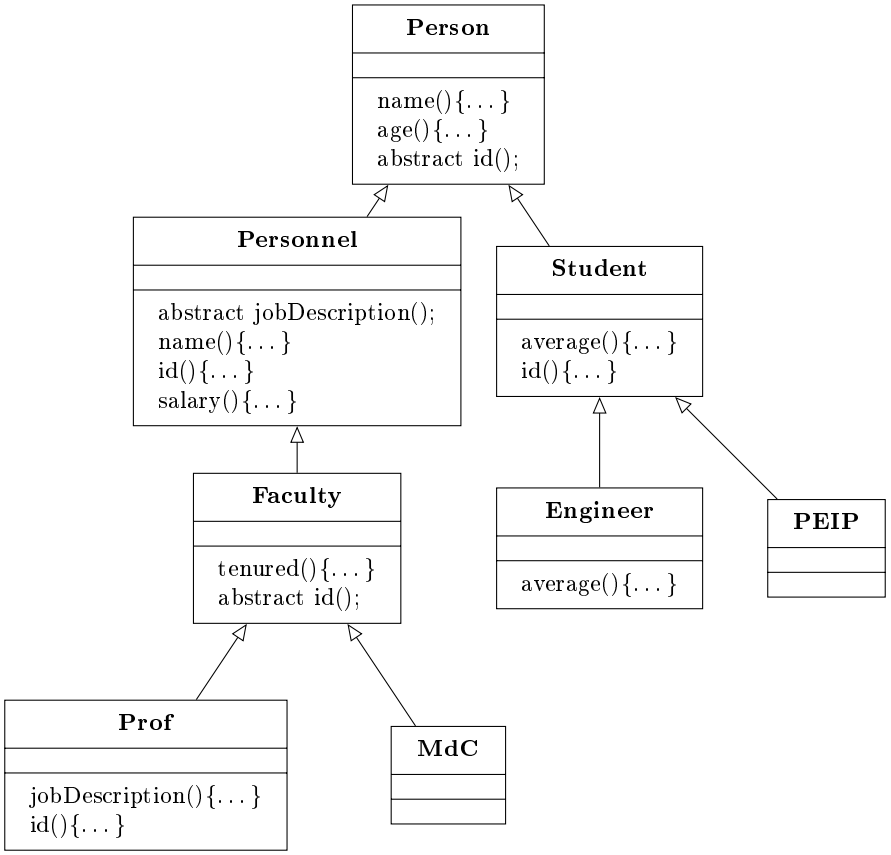
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

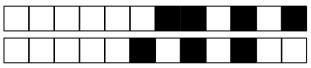
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `PEIP`
- ☐ `Engineer`
- ☐ `Prof`
- ☐ `Person`
- ☐ `MdC`
- ☐ `Personnel`
- ☐ `Student`
- ☐ `Faculty`



+53/1/20+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ *indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.*
Dans tout le code, les `package` *et les* `import` *sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.*

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Done"
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ package-private
- ☐ `protected`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

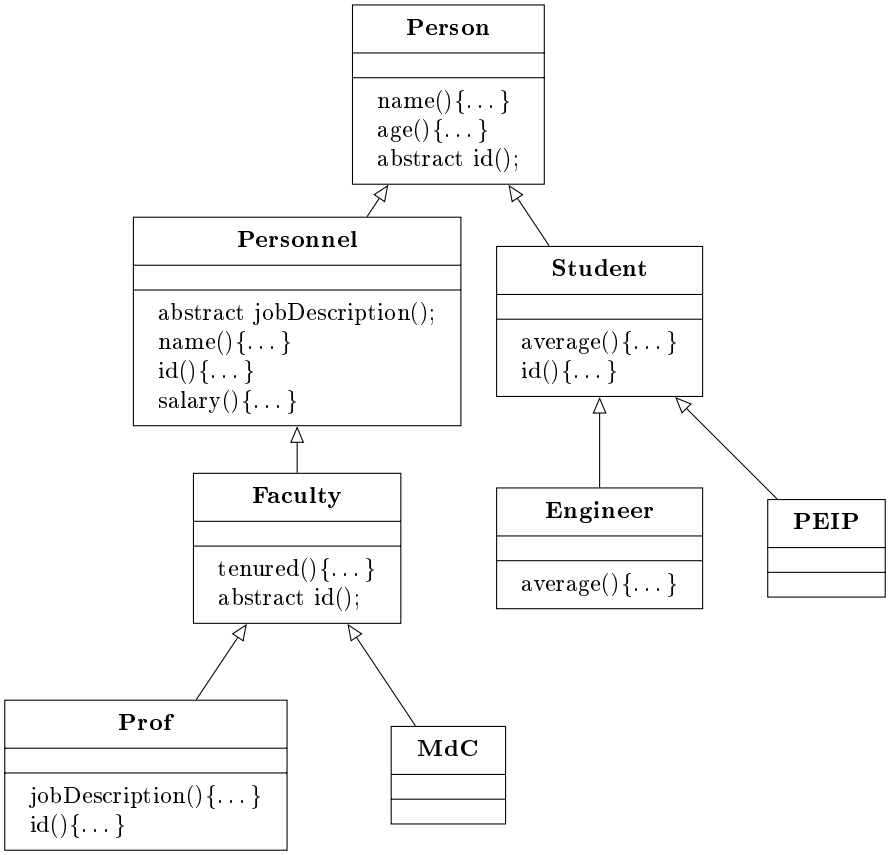
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

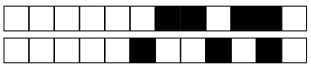
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Student`
- ☐ `MdC`
- ☐ `Person`
- ☐ `Prof`
- ☐ `PEIP`
- ☐ `Faculty`
- ☐ `Personnel`
- ☐ `Engineer`



+54/1/10+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

☐ Affiche "Caught exception" et "Finallied exception"

☐ Erreur de compilation

☐ Erreur d'exécution

☐ Affiche seulement "Finallied exception"

☐ Affiche seulement "Caught exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ package-private
- ☐ `private`
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

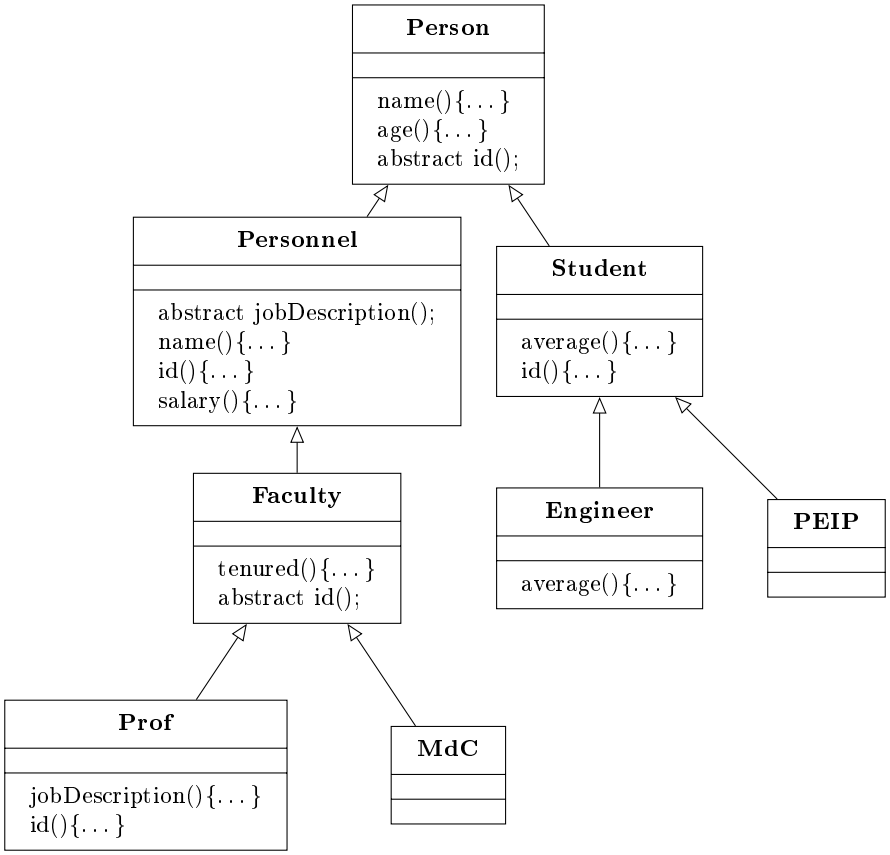
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Faculty`
- ☐ `Engineer`
- ☐ `Prof`
- ☐ `Personnel`
- ☐ `Student`
- ☐ `PEIP`
- ☐ `MdC`
- ☐ `Person`

+55/1/60+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

☐ Erreur d'exécution

☐ Erreur de compilation

☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ `protected`
- ☐ `public`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut implémenter une interface

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

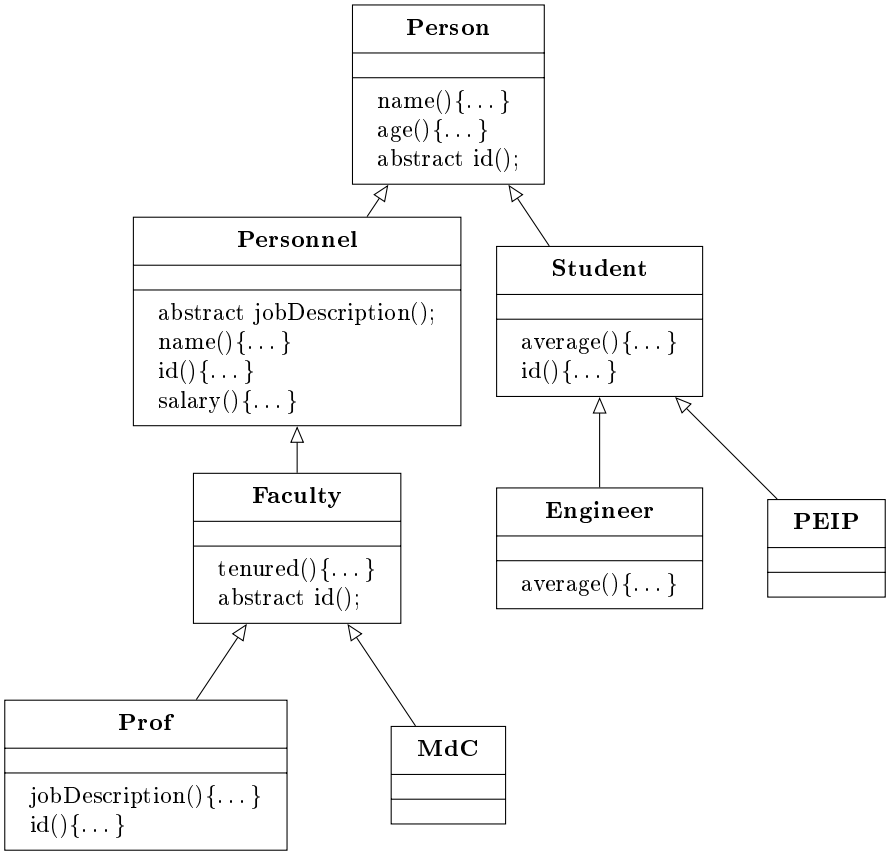
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Student`
- ☐ `Faculty`
- ☐ `Personnel`
- ☐ `MdC`
- ☐ `Engineer`
- ☐ `PEIP`
- ☐ `Person`
- ☐ `Prof`

+56/1/50+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `private`
- ☐ `protected`
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

☐ `whatever` ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `something` ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `whatever` ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe abstraite

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

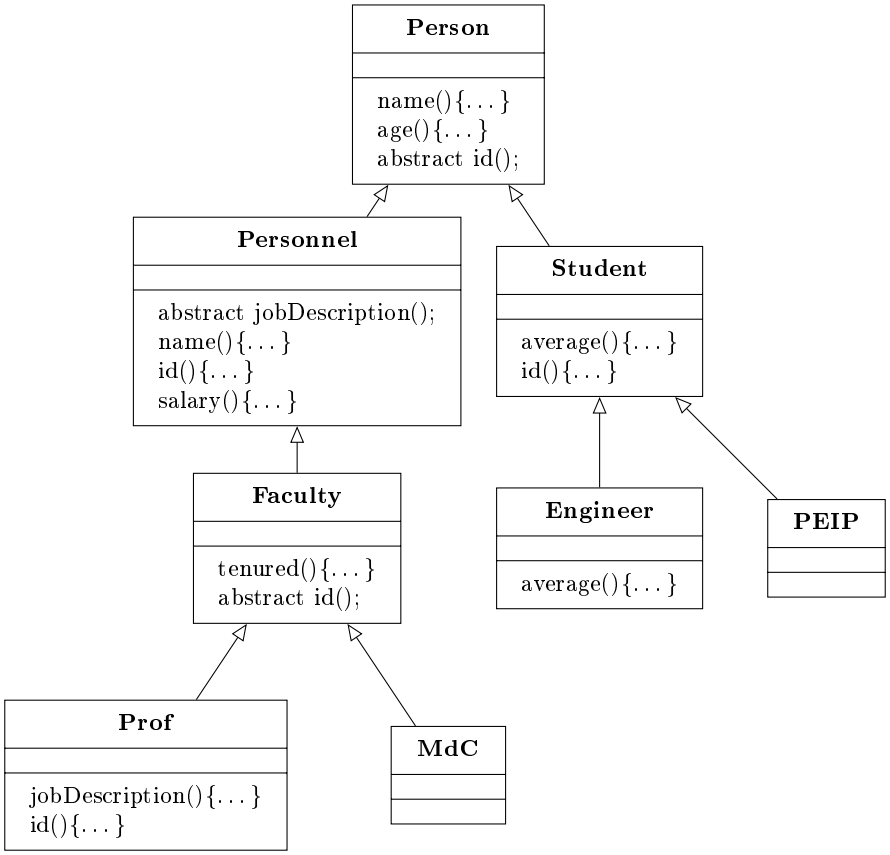
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Prof`
- ☐ `PEIP`
- ☐ `MdC`
- ☐ `Faculty`
- ☐ `Engineer`
- ☐ `Personnel`
- ☐ `Person`
- ☐ `Student`

+57/1/40+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

☐ Erreur de compilation

☐ Affiche "Done"

☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ package-private
- ☐ `public`
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

☐ `something` ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `something` ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `something` ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

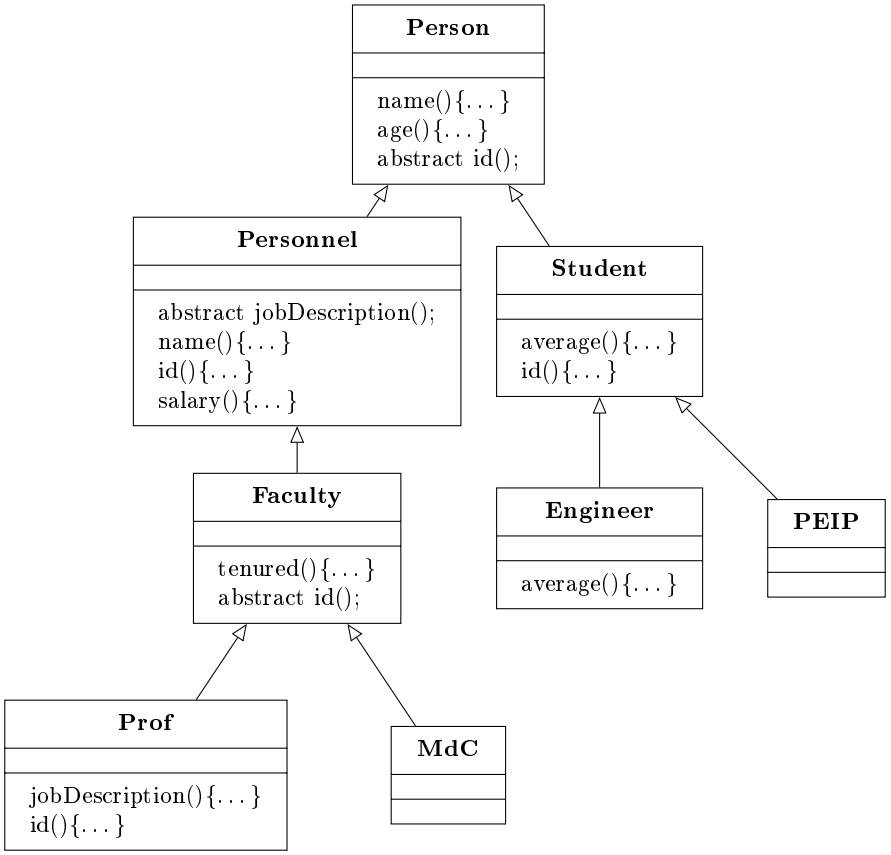
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

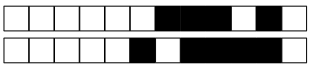
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Student`
- ☐ `Prof`
- ☐ `PEIP`
- ☐ `Faculty`
- ☐ `Person`
- ☐ `Engineer`
- ☐ `Personnel`
- ☐ `MdC`



+58/1/30+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Done"
- ☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Done"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ `public`
- ☐ package-private
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `something` ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `something` ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des méthodes privées

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

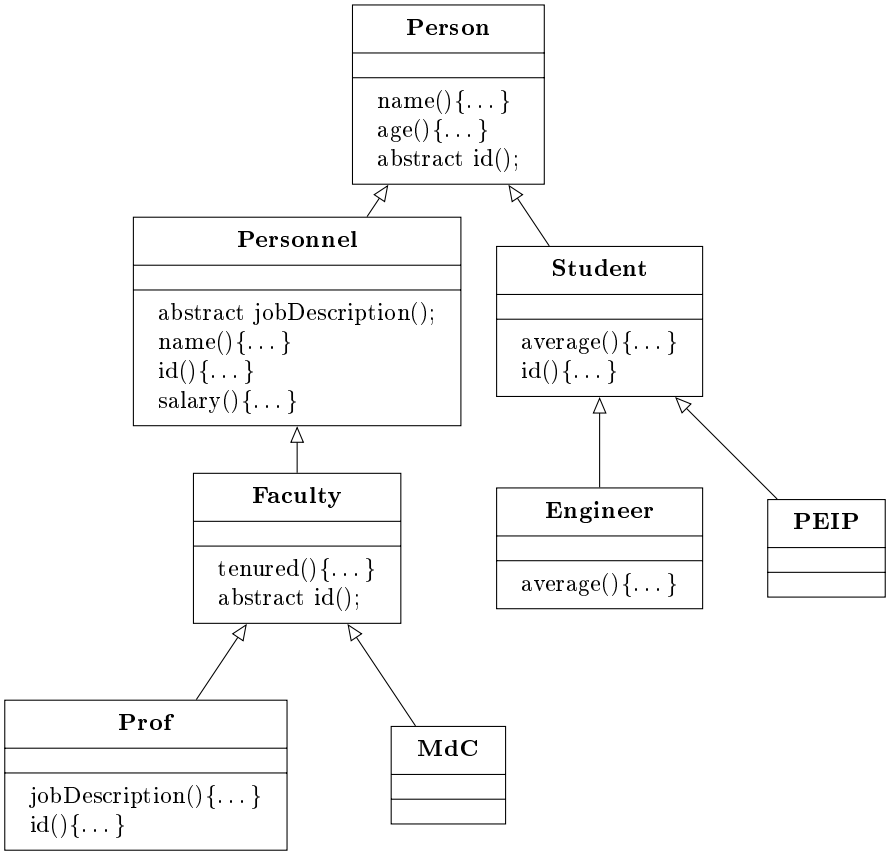
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

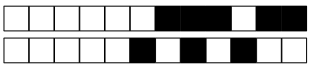
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Prof`
- ☐ `Faculty`
- ☐ `MdC`
- ☐ `Student`
- ☐ `Person`
- ☐ `Engineer`
- ☐ `Personnel`
- ☐ `PEIP`



+59/1/20+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses.
Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés.
Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Done"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `private`
- ☐ `protected`
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

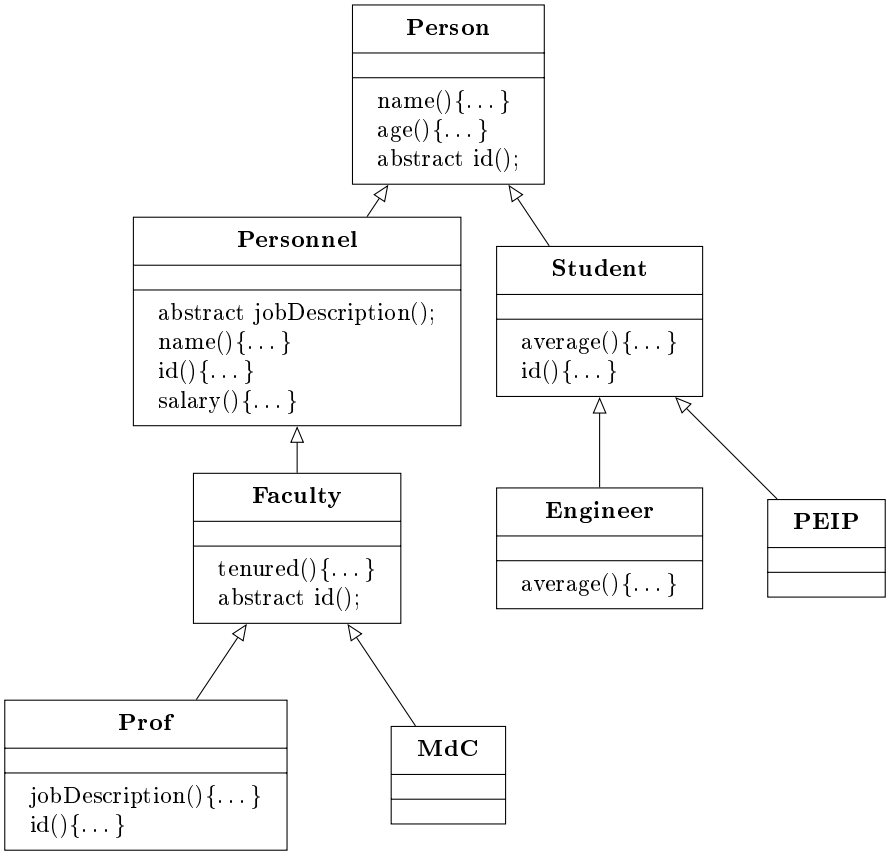
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Person`
- ☐ `MdC`
- ☐ `PEIP`
- ☐ `Student`
- ☐ `Prof`
- ☐ `Engineer`
- ☐ `Personnel`
- ☐ `Faculty`

+60/1/10+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ package-private
- ☐ `protected`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

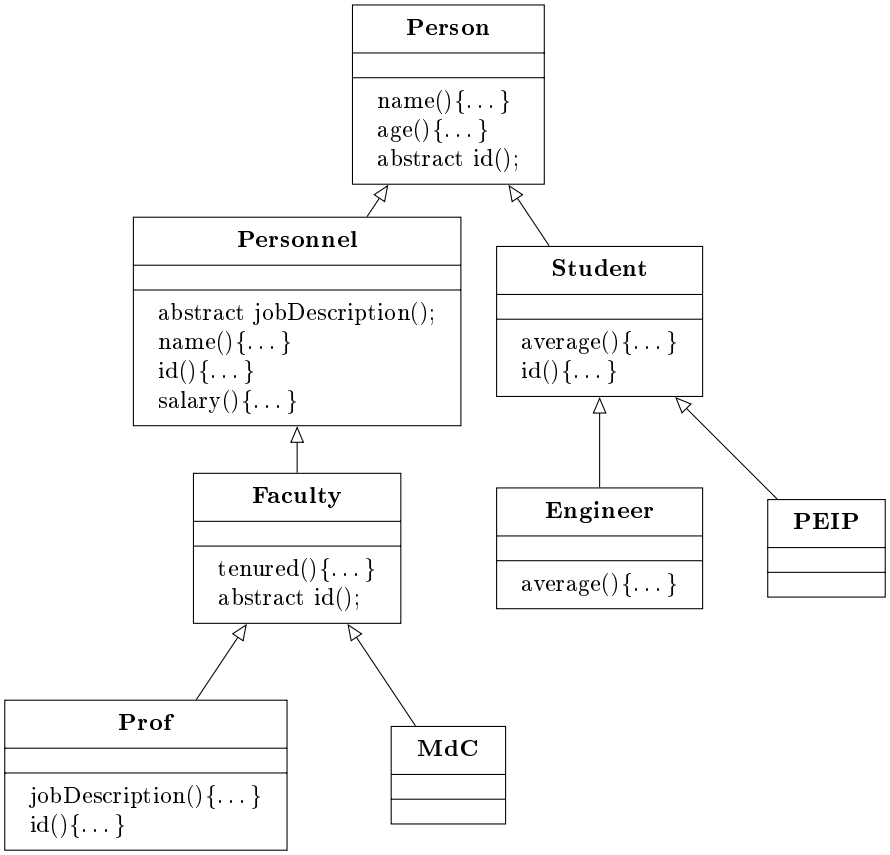
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Faculty`
- ☐ `PEIP`
- ☐ `Personnel`
- ☐ `Student`
- ☐ `Person`
- ☐ `Prof`
- ☐ `MdC`
- ☐ `Engineer`

+61/1/60+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

☐ Erreur de compilation

☐ Erreur d'exécution

☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Done"
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `private`
- ☐ `protected`
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `something` ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `something` ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut implémenter une interface

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

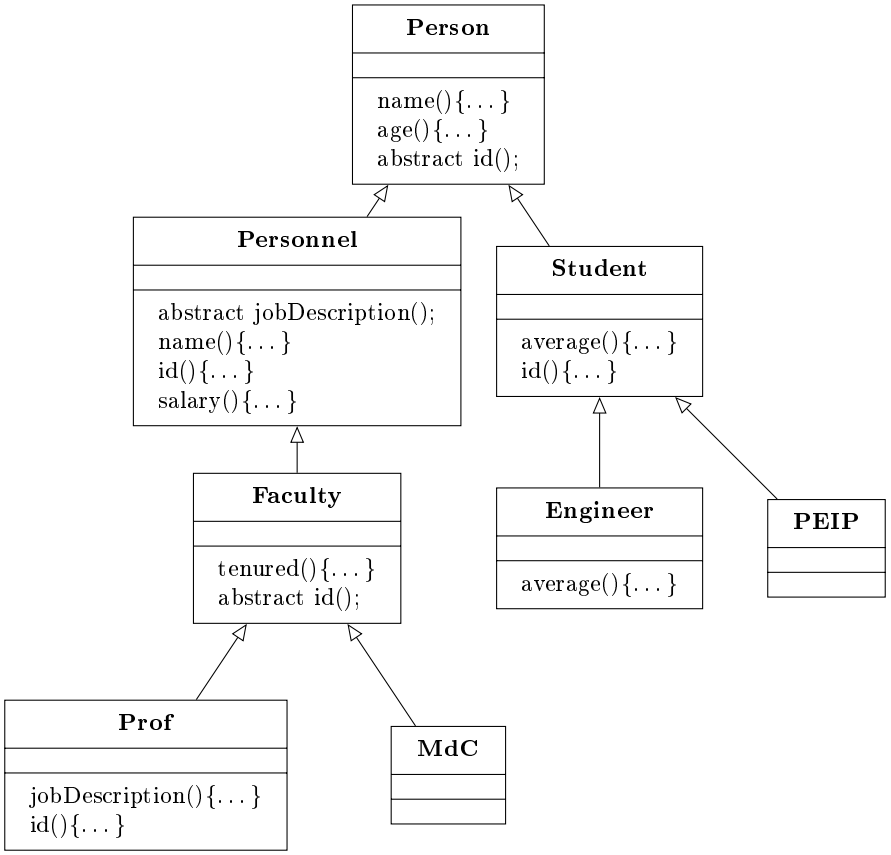
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Student`
- ☐ `Engineer`
- ☐ `Person`
- ☐ `Prof`
- ☐ `PEIP`
- ☐ `Personnel`
- ☐ `MdC`
- ☐ `Faculty`

+62/1/50+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche "Done"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ `private`
- ☐ package-private
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

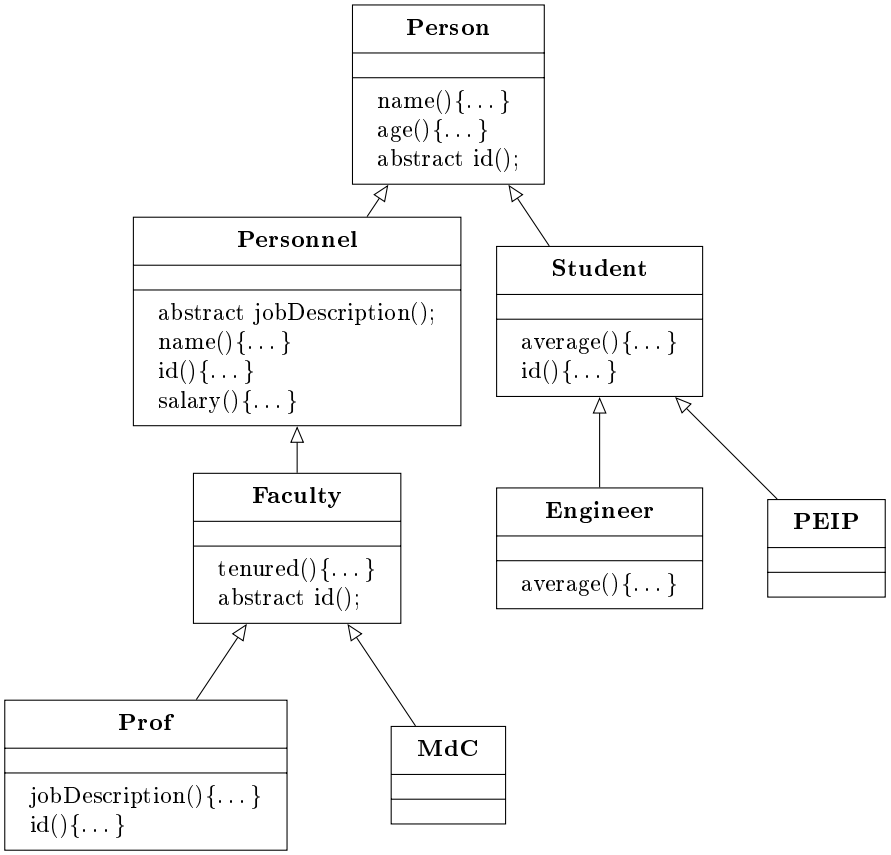
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Person`
- ☐ `PEIP`
- ☐ `Faculty`
- ☐ `MdC`
- ☐ `Engineer`
- ☐ `Personnel`
- ☐ `Student`
- ☐ `Prof`

+63/1/40+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

☐ Erreur d'exécution

☐ Erreur de compilation

☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ `private`
- ☐ package-private
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

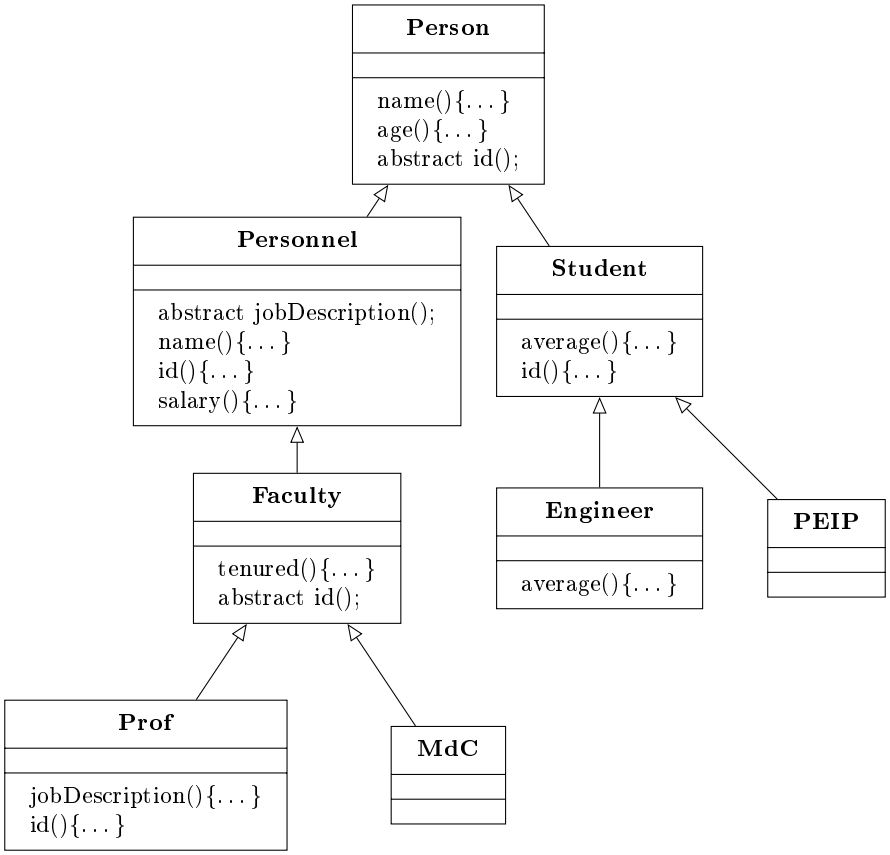
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

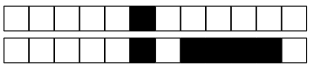
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Prof`
- ☐ `PEIP`
- ☐ `Student`
- ☐ `Person`
- ☐ `Faculty`
- ☐ `Engineer`
- ☐ `Personnel`
- ☐ `MdC`



+64/1/30+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Done"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ `private`
- ☐ `protected`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut implémenter une interface

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

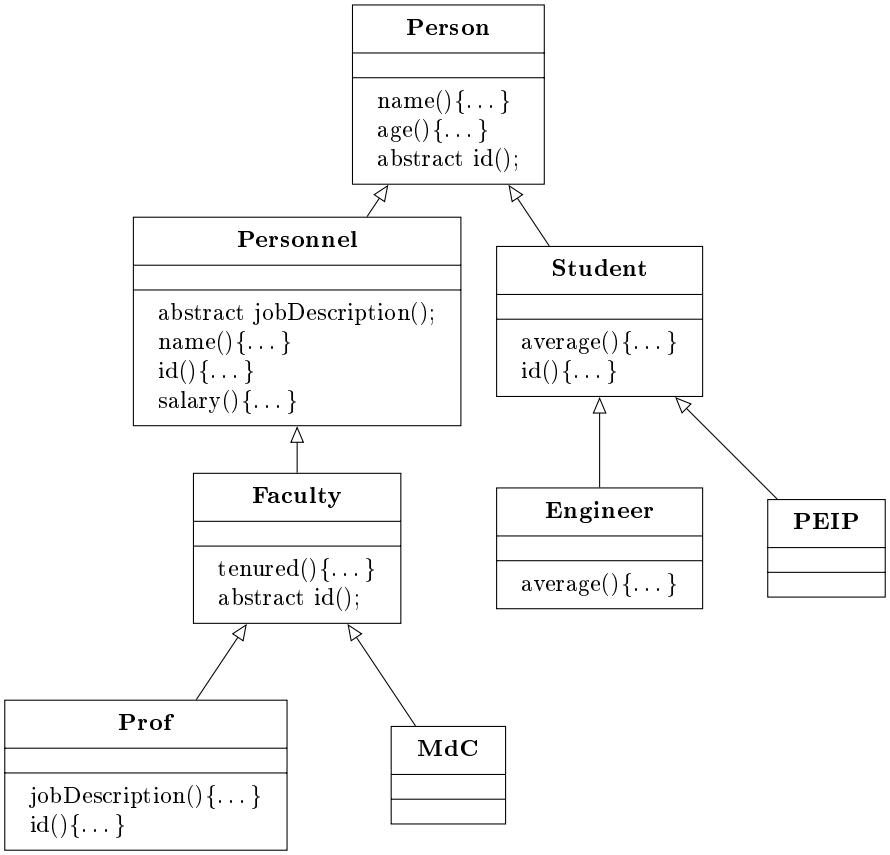
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

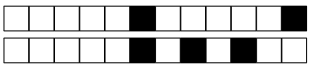
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `PEIP`
- ☐ `Faculty`
- ☐ `Engineer`
- ☐ `Person`
- ☐ `Personnel`
- ☐ `Prof`
- ☐ `Student`
- ☐ `MdC`



+65/1/20+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

☐ Erreur d'exécution

☐ Erreur de compilation

☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ `protected`
- ☐ `private`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

☐ `something` ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `whatever` ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `something` ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des méthodes privées

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

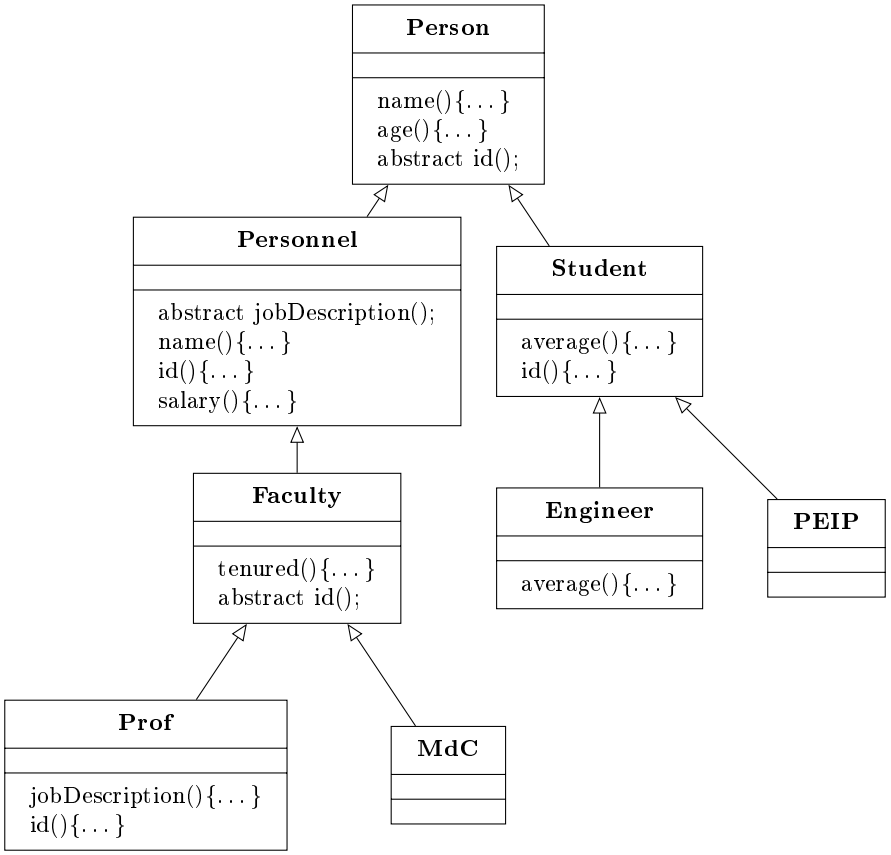
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

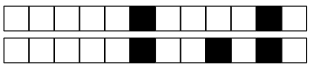
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `MdC`
- ☐ `Student`
- ☐ `Engineer`
- ☐ `PEIP`
- ☐ `Personnel`
- ☐ `Person`
- ☐ `Faculty`
- ☐ `Prof`



+66/1/10+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

☐ Erreur de compilation

☐ Erreur d'exécution

☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ `protected`
- ☐ `public`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

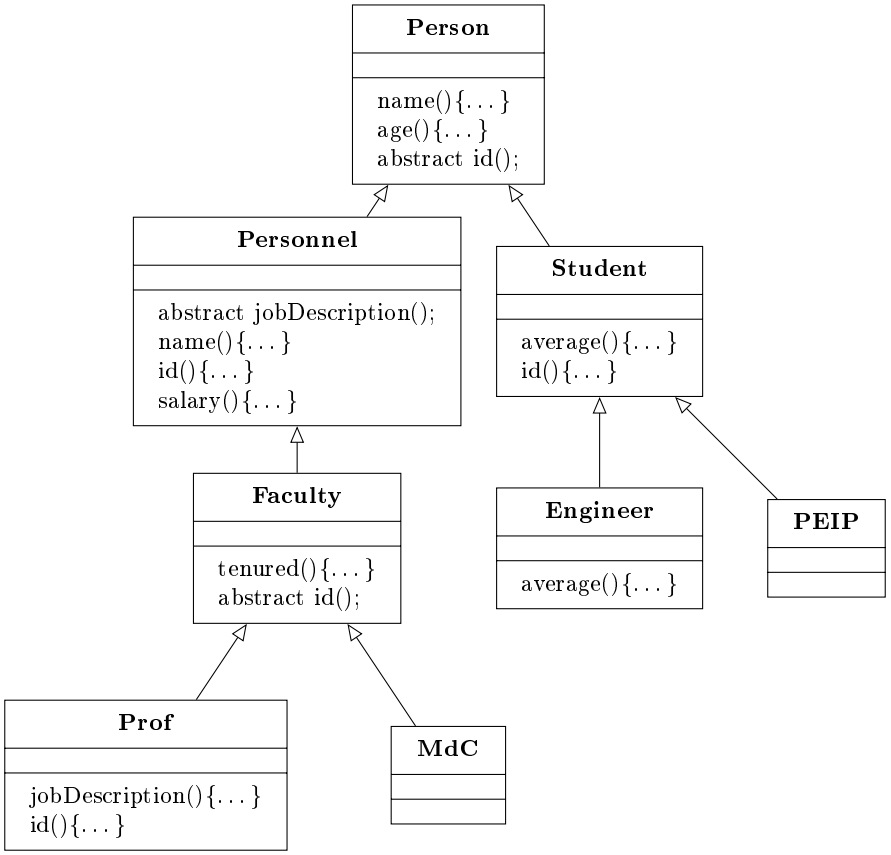
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `PEIP`
- ☐ `MdC`
- ☐ `Student`
- ☐ `Faculty`
- ☐ `Personnel`
- ☐ `Engineer`
- ☐ `Person`
- ☐ `Prof`

+67/1/60+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `public`
- ☐ `protected`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

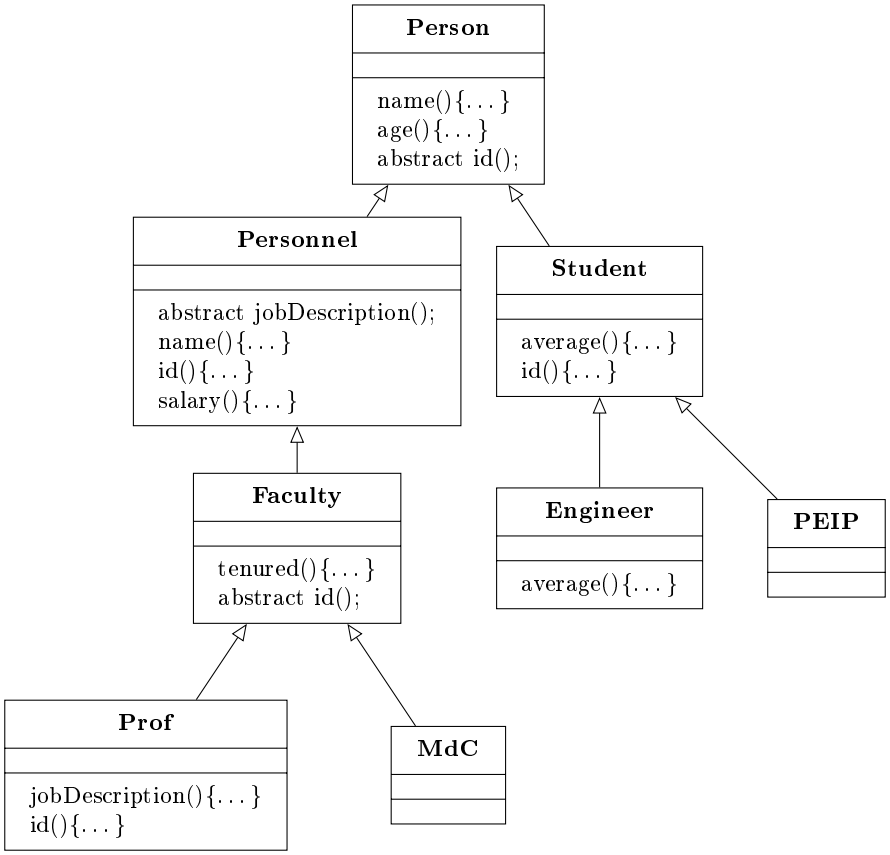
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Personnel`
- ☐ `MdC`
- ☐ `Prof`
- ☐ `Student`
- ☐ `Engineer`
- ☐ `Faculty`
- ☐ `Person`
- ☐ `PEIP`

+68/1/50+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

☐ Erreur de compilation

☐ Erreur d'exécution

☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Done"
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ `private`
- ☐ `public`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `something` ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `something` ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

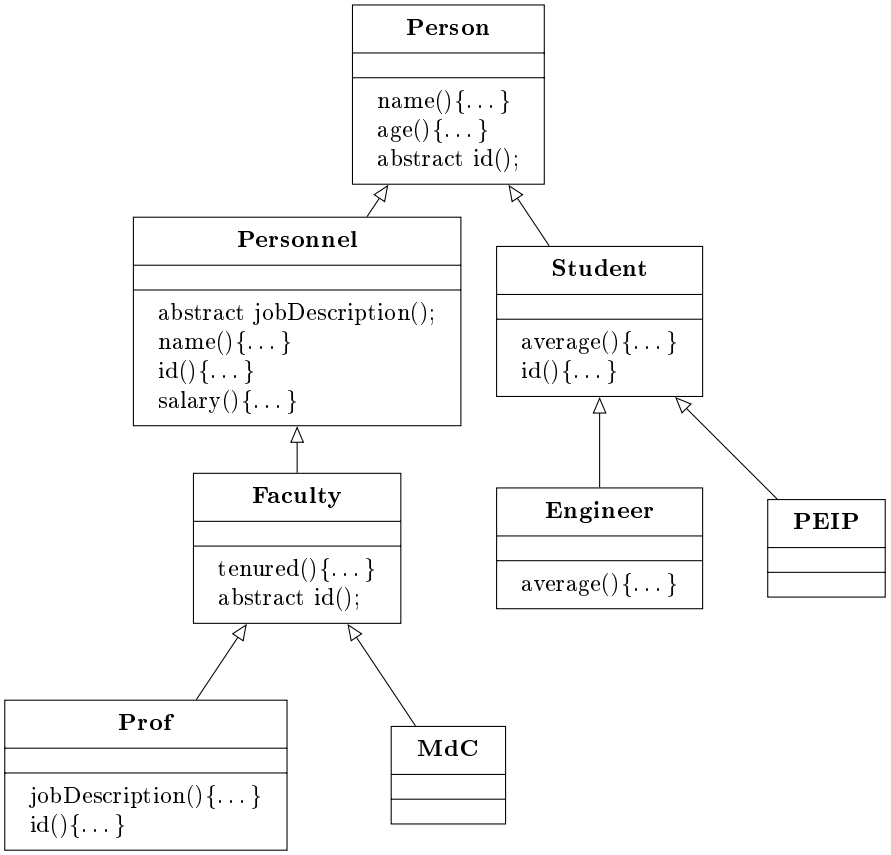
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Student`
- ☐ `PEIP`
- ☐ `Faculty`
- ☐ `MdC`
- ☐ `Prof`
- ☐ `Person`
- ☐ `Personnel`
- ☐ `Engineer`

+69/1/40+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses.
Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés.
Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Done"
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `private`
- ☐ `protected`
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

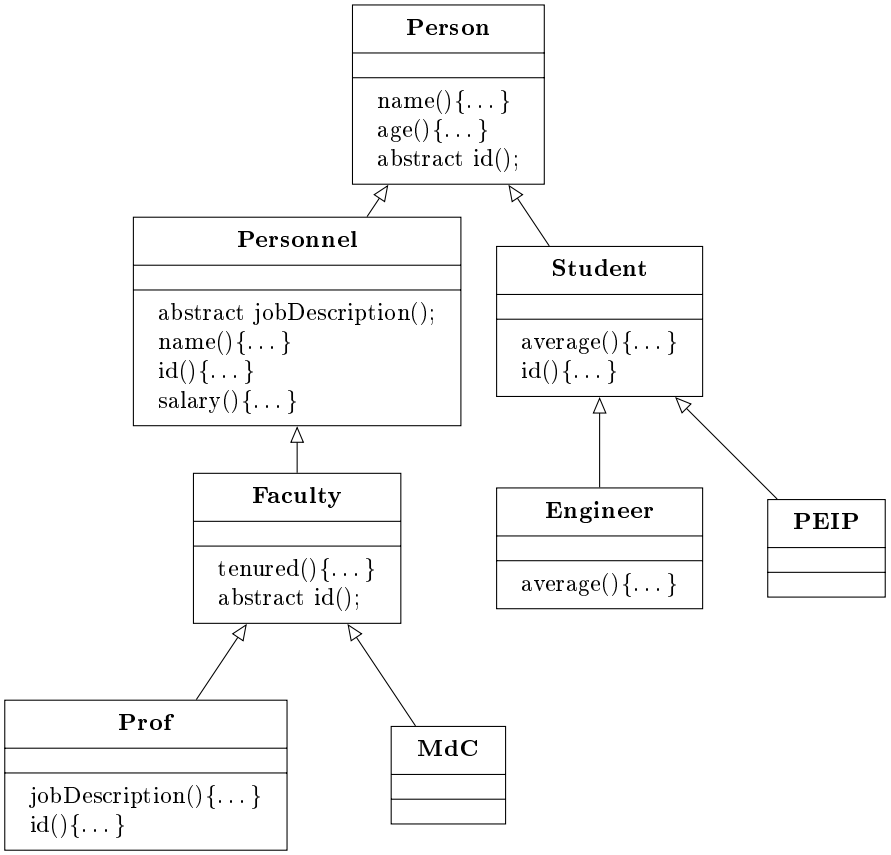
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

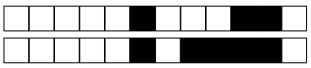
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Prof`
- ☐ `Person`
- ☐ `MdC`
- ☐ `PEIP`
- ☐ `Student`
- ☐ `Engineer`
- ☐ `Faculty`
- ☐ `Personnel`



+70/1/30+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `private`
- ☐ `protected`
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

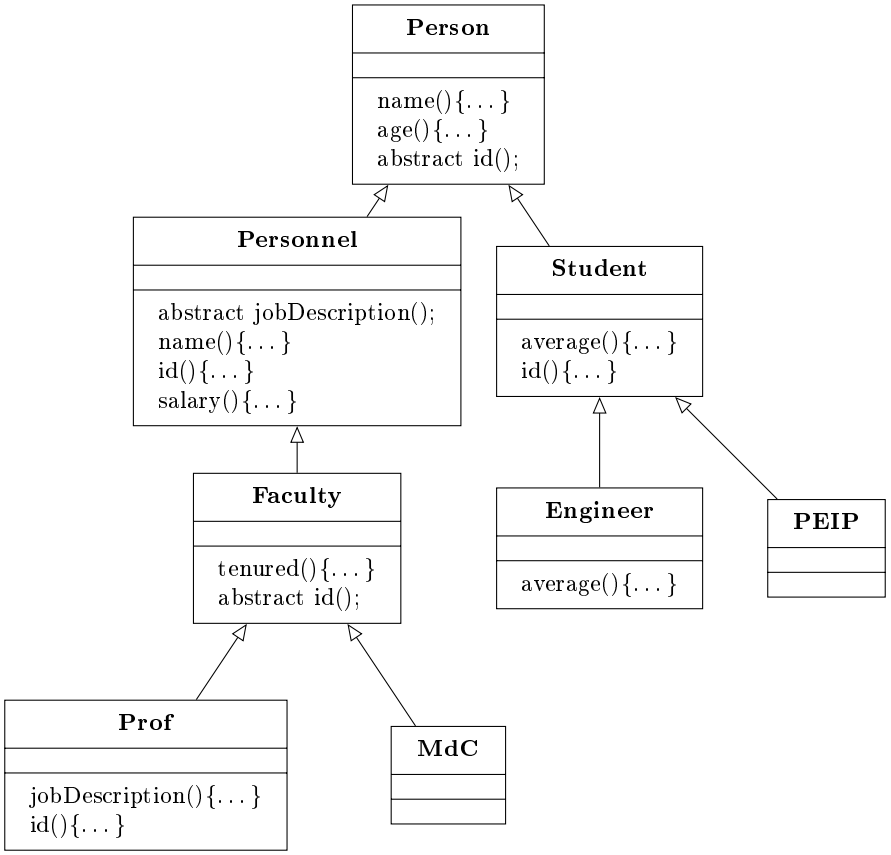
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

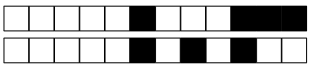
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Person`
- ☐ `MdC`
- ☐ `Student`
- ☐ `Prof`
- ☐ `Personnel`
- ☐ `Faculty`
- ☐ `PEIP`
- ☐ `Engineer`



+71/1/20+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

☐ Erreur d'exécution

☐ Erreur de compilation

☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche "Done"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ package-private
- ☐ `public`
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

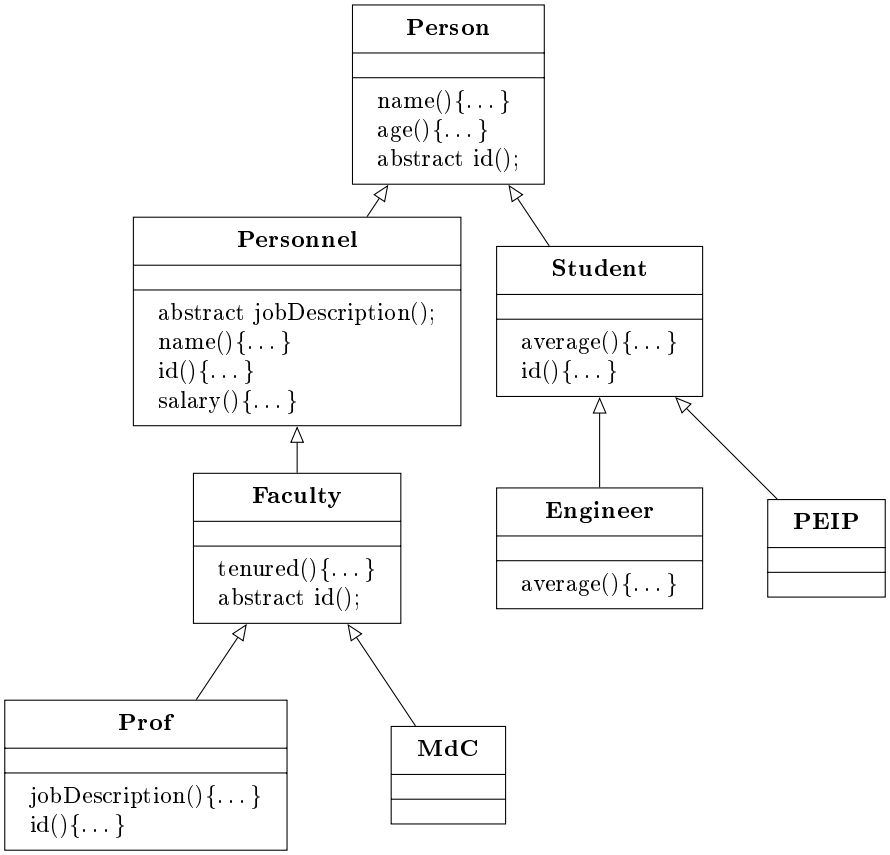
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

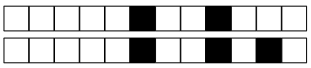
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `MdC`
- ☐ `Faculty`
- ☐ `PEIP`
- ☐ `Engineer`
- ☐ `Prof`
- ☐ `Student`
- ☐ `Person`
- ☐ `Personnel`



+72/1/10+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Done"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `public`
- ☐ `protected`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

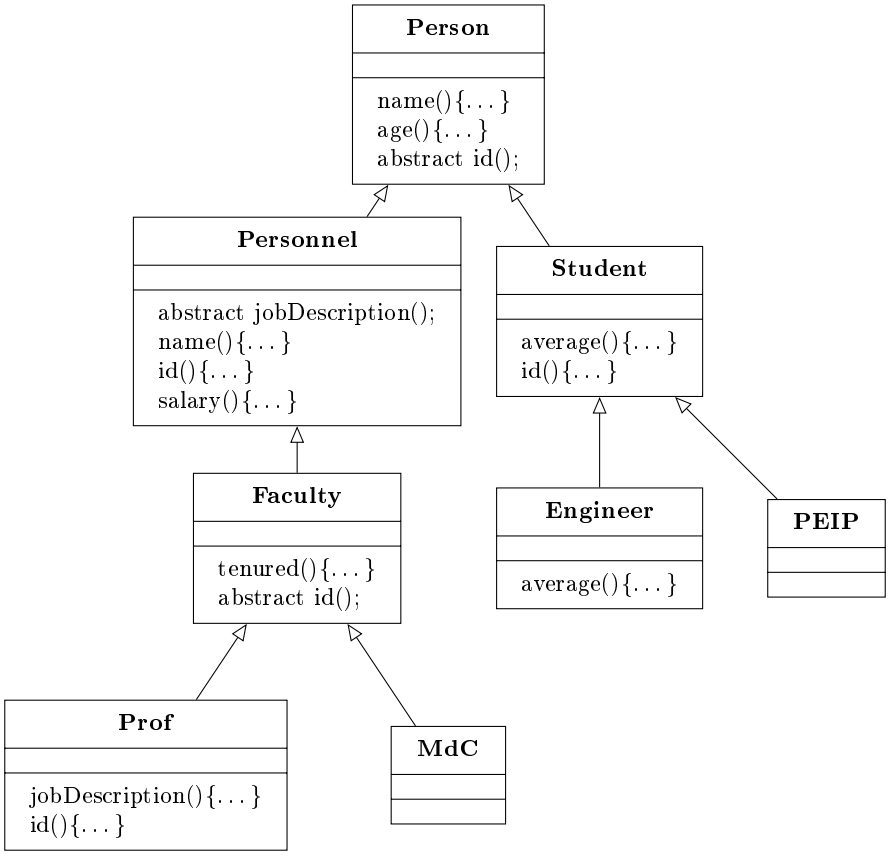
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Faculty`
- ☐ `Person`
- ☐ `Prof`
- ☐ `Engineer`
- ☐ `MdC`
- ☐ `PEIP`
- ☐ `Personnel`
- ☐ `Student`

+73/1/60+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Done"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ `private`
- ☐ package-private
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

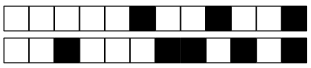
Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0



Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

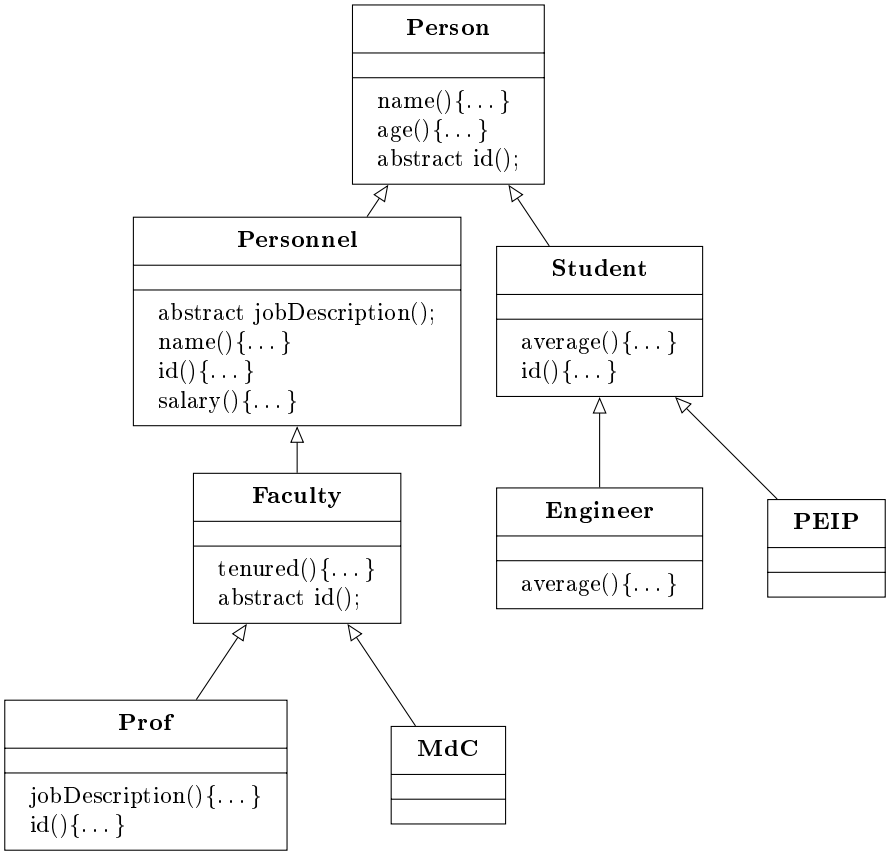
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Faculty`
- ☐ `Personnel`
- ☐ `MdC`
- ☐ `Person`
- ☐ `Engineer`
- ☐ `Student`
- ☐ `Prof`
- ☐ `PEIP`

+74/1/50+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ `protected`
- ☐ `public`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des méthodes privées

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

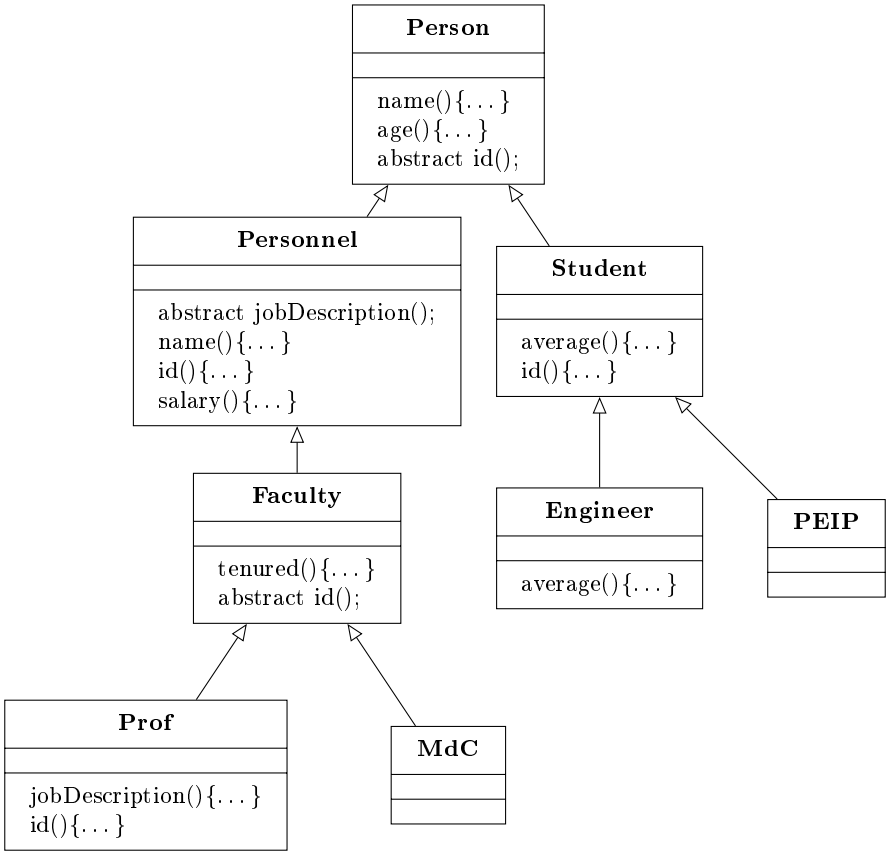
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Engineer`
- ☐ `Prof`
- ☐ `Student`
- ☐ `PEIP`
- ☐ `Personnel`
- ☐ `Faculty`
- ☐ `MdC`
- ☐ `Person`

+75/1/40+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Done"
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ `private`
- ☐ `protected`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

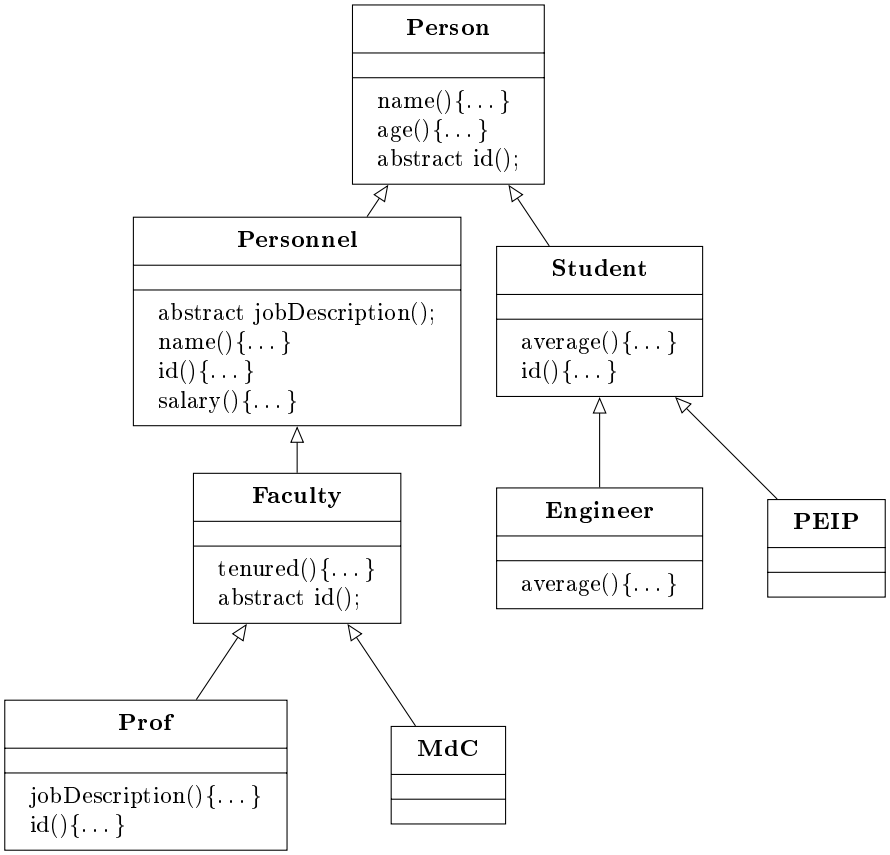
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

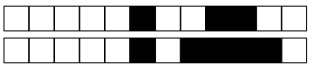
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Personnel`
- ☐ `Engineer`
- ☐ `PEIP`
- ☐ `Faculty`
- ☐ `Student`
- ☐ `Person`
- ☐ `MdC`
- ☐ `Prof`



+76/1/30+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `private`
- ☐ `protected`
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des méthodes privées

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

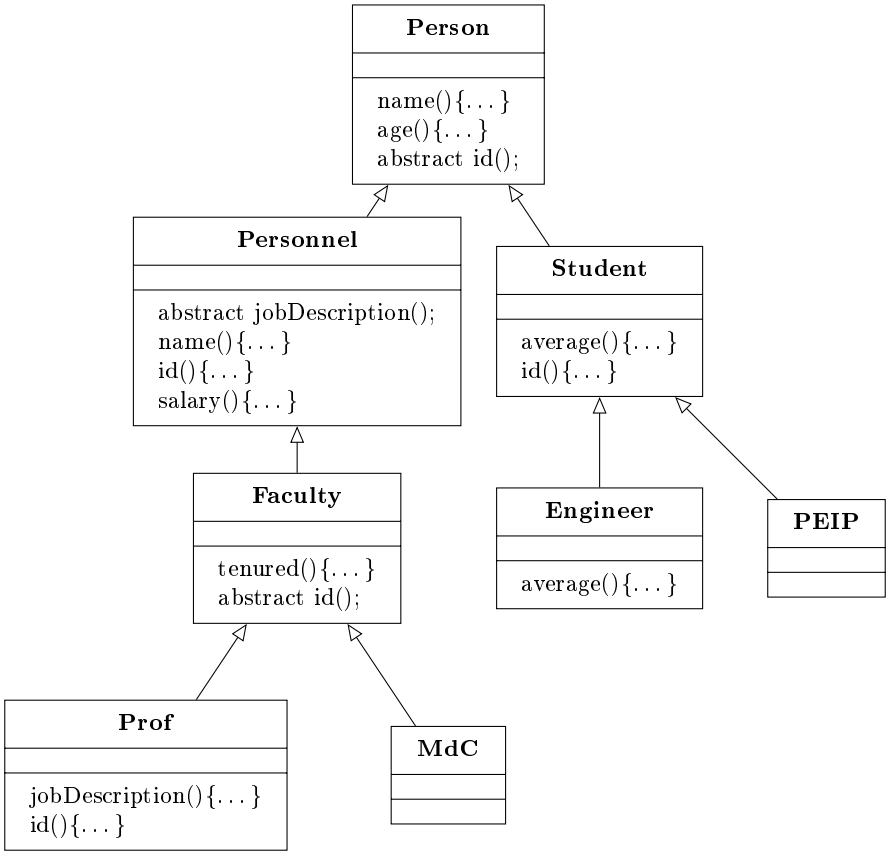
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

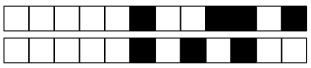
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `PEIP`
- ☐ `Person`
- ☐ `Prof`
- ☐ `Faculty`
- ☐ `MdC`
- ☐ `Personnel`
- ☐ `Engineer`
- ☐ `Student`



+77/1/20+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

☐ Erreur d'exécution

☐ Erreur de compilation

☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche "Done"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `public`
- ☐ `protected`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe abstraite

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

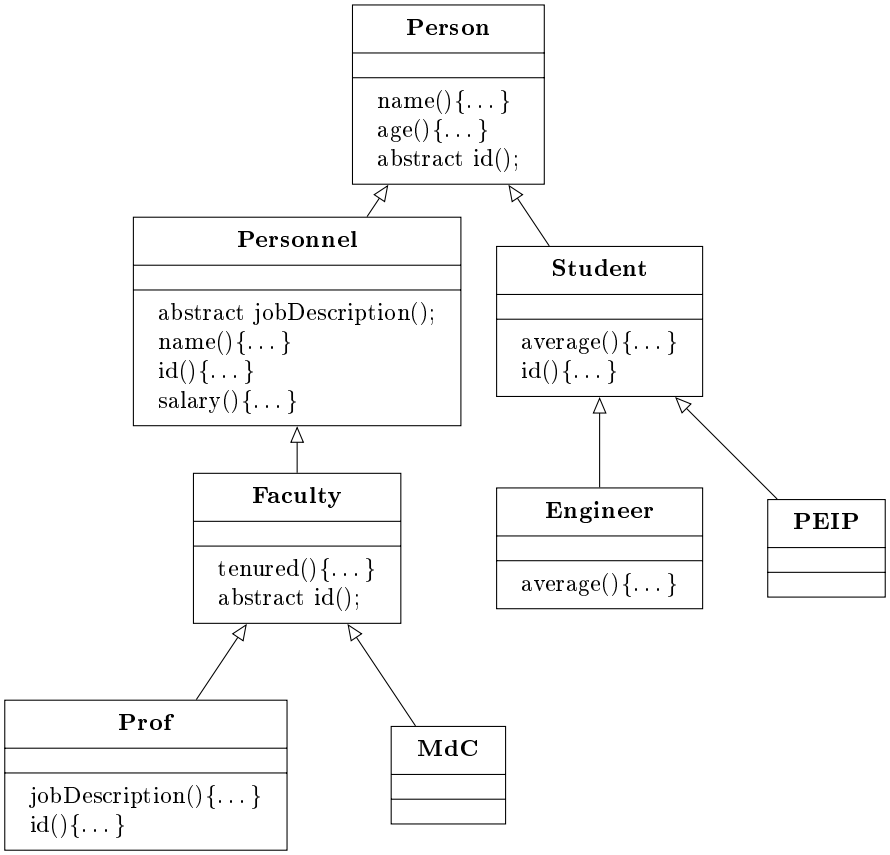
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

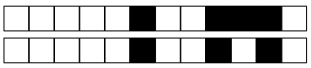
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `MdC`
- ☐ `Student`
- ☐ `Engineer`
- ☐ `PEIP`
- ☐ `Person`
- ☐ `Faculty`
- ☐ `Personnel`
- ☐ `Prof`



+78/1/10+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Done"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `protected`
- ☐ `private`
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre une classe abstraite

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

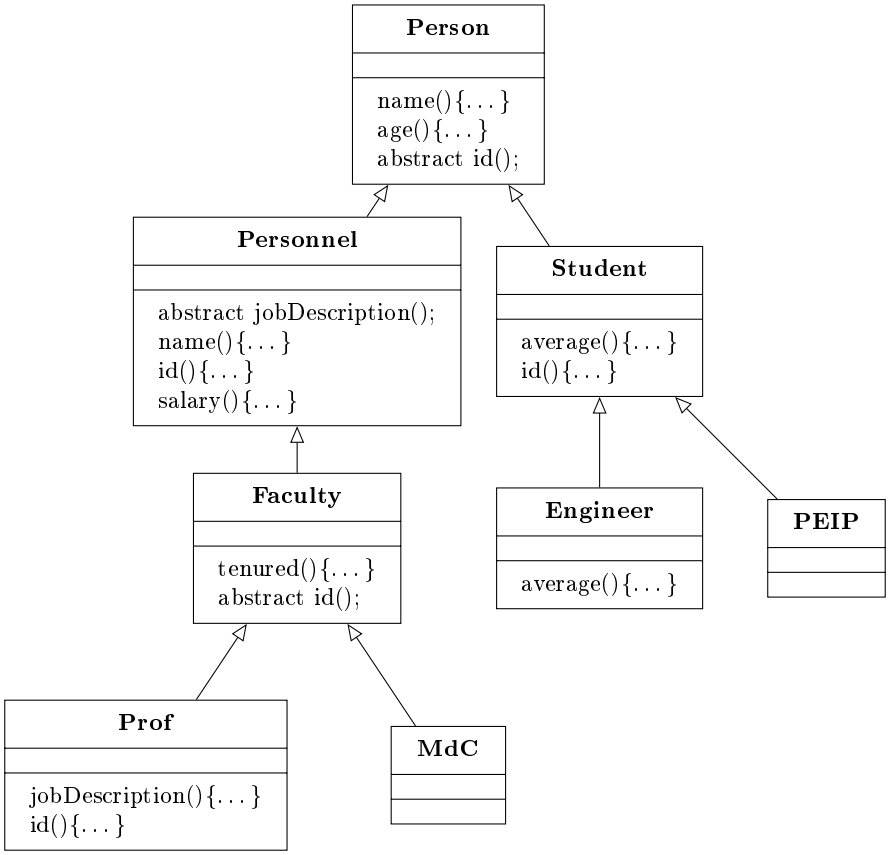
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Person`
- ☐ `Faculty`
- ☐ `Engineer`
- ☐ `PEIP`
- ☐ `MdC`
- ☐ `Student`
- ☐ `Personnel`
- ☐ `Prof`

+79/1/60+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

☐ Erreur d'exécution

☐ Affiche "Done"

☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche "Done"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `public`
- ☐ `protected`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

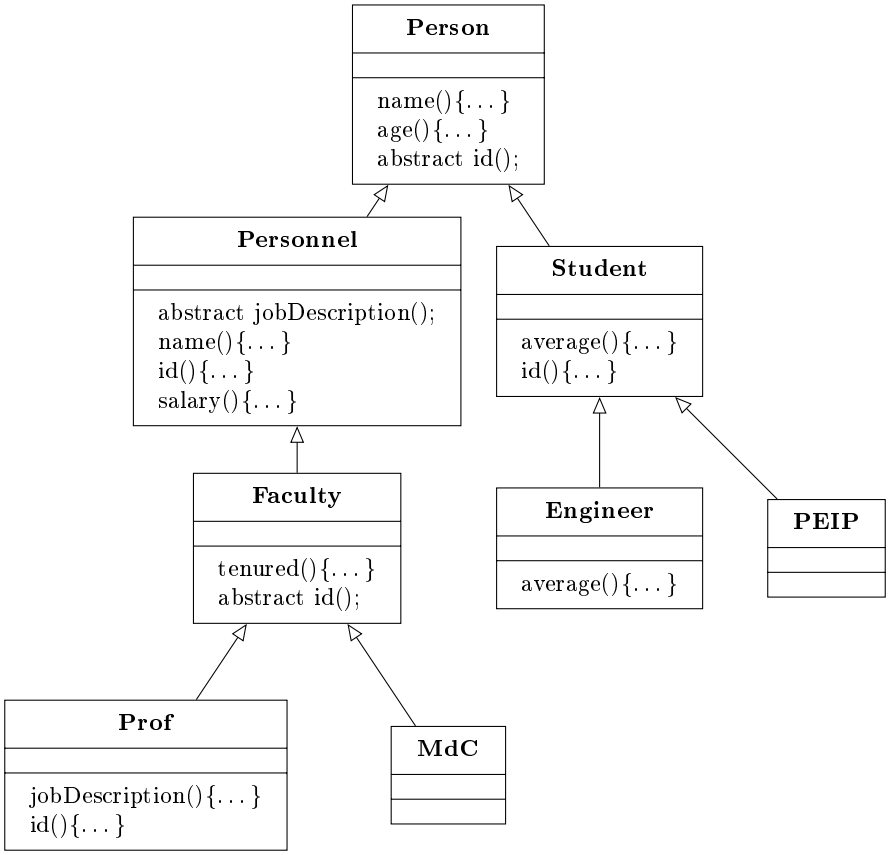
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Person`
- ☐ `PEIP`
- ☐ `MdC`
- ☐ `Prof`
- ☐ `Faculty`
- ☐ `Student`
- ☐ `Personnel`
- ☐ `Engineer`

+80/1/50+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ *indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.*
Dans tout le code, les `package` *et les* `import` *sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.*

Question 1 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche "Done"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ package-private
- ☐ `public`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

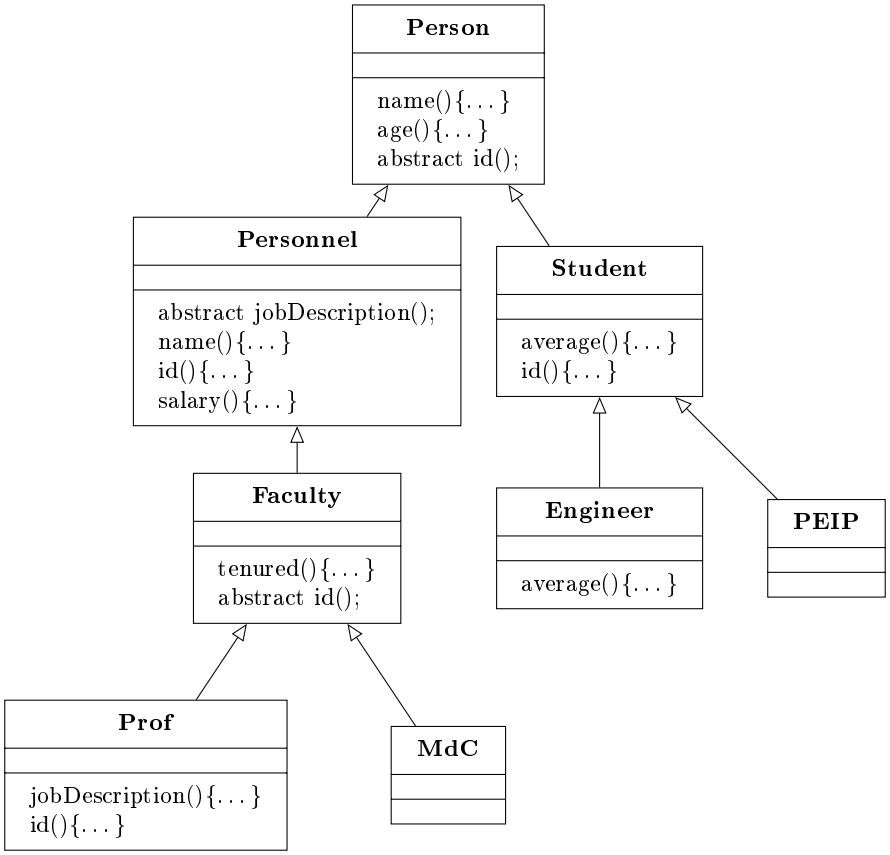
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Personnel`
- ☐ `Faculty`
- ☐ `PEIP`
- ☐ `Person`
- ☐ `Student`
- ☐ `MdC`
- ☐ `Prof`
- ☐ `Engineer`

+81/1/40+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Done"
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ `public`
- ☐ package-private
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `whatever` ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `whatever` ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe abstraite

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

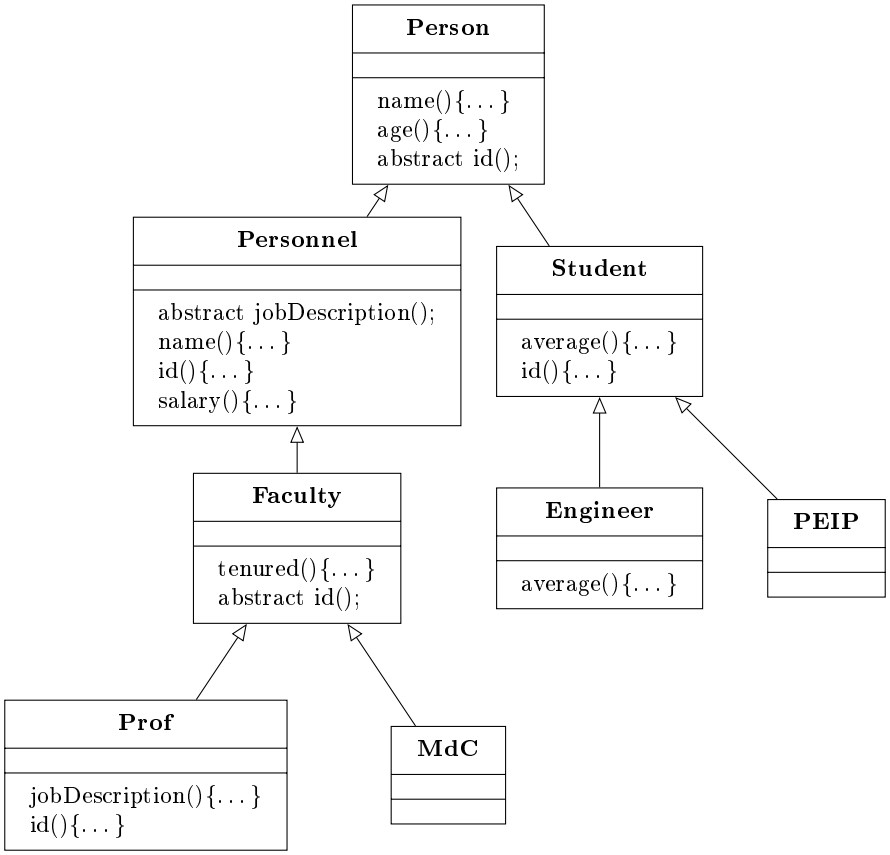
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

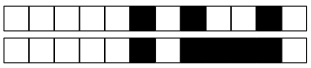
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Engineer`
- ☐ `Personnel`
- ☐ `Student`
- ☐ `MdC`
- ☐ `Prof`
- ☐ `Person`
- ☐ `Faculty`
- ☐ `PEIP`



+82/1/30+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `public`
- ☐ `private`
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `whatever` ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `whatever` ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

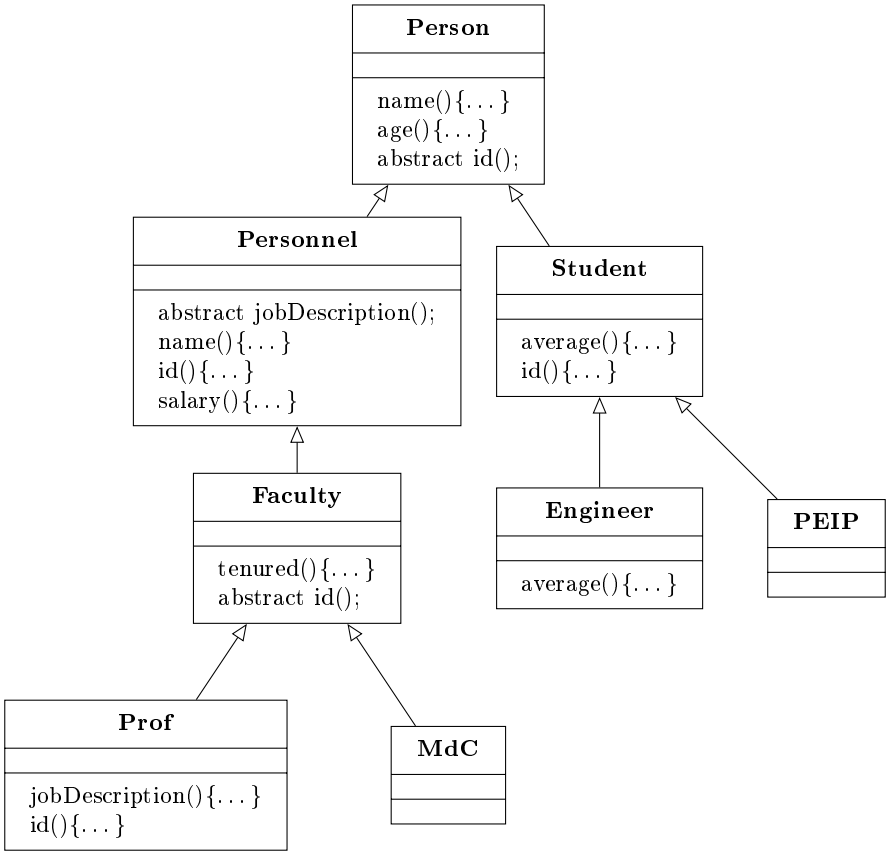
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

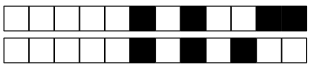
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Prof`
- ☐ `Person`
- ☐ `Student`
- ☐ `MdC`
- ☐ `Personnel`
- ☐ `PEIP`
- ☐ `Faculty`
- ☐ `Engineer`



+83/1/20+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Done"
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `private`
- ☐ `public`
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

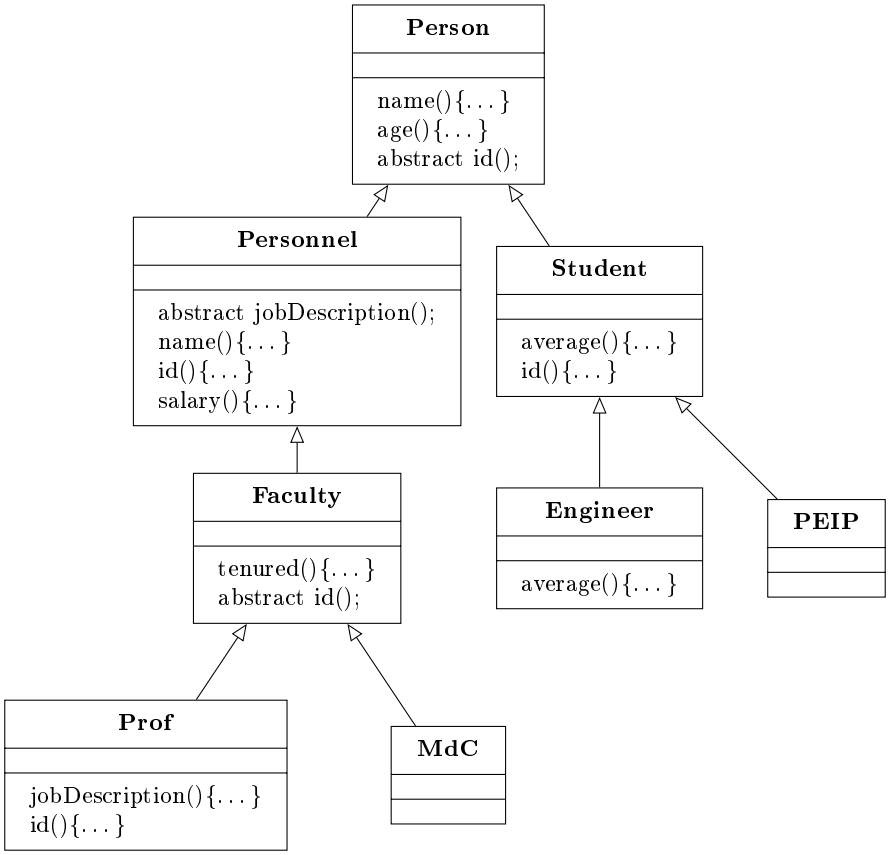
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Person`
- ☐ `Engineer`
- ☐ `Faculty`
- ☐ `Prof`
- ☐ `MdC`
- ☐ `Personnel`
- ☐ `Student`
- ☐ `PEIP`

+84/1/10+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ `protected`
- ☐ package-private
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

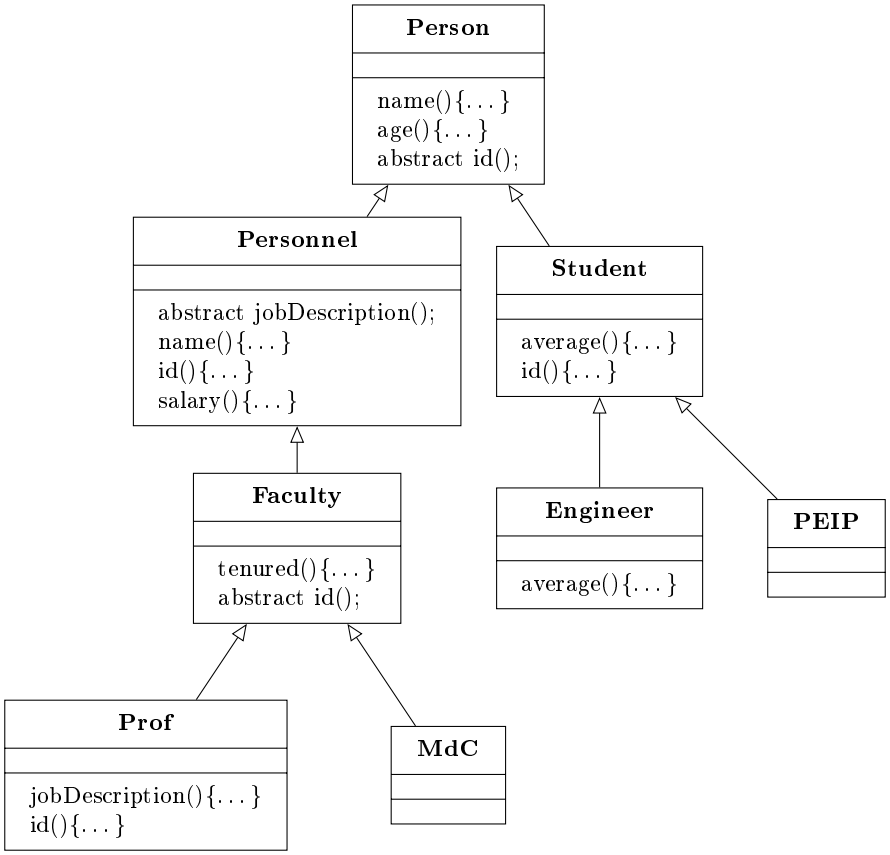
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Student`
- ☐ `Person`
- ☐ `Engineer`
- ☐ `Prof`
- ☐ `MdC`
- ☐ `PEIP`
- ☐ `Faculty`
- ☐ `Personnel`

+85/1/60+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ `private`
- ☐ `public`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

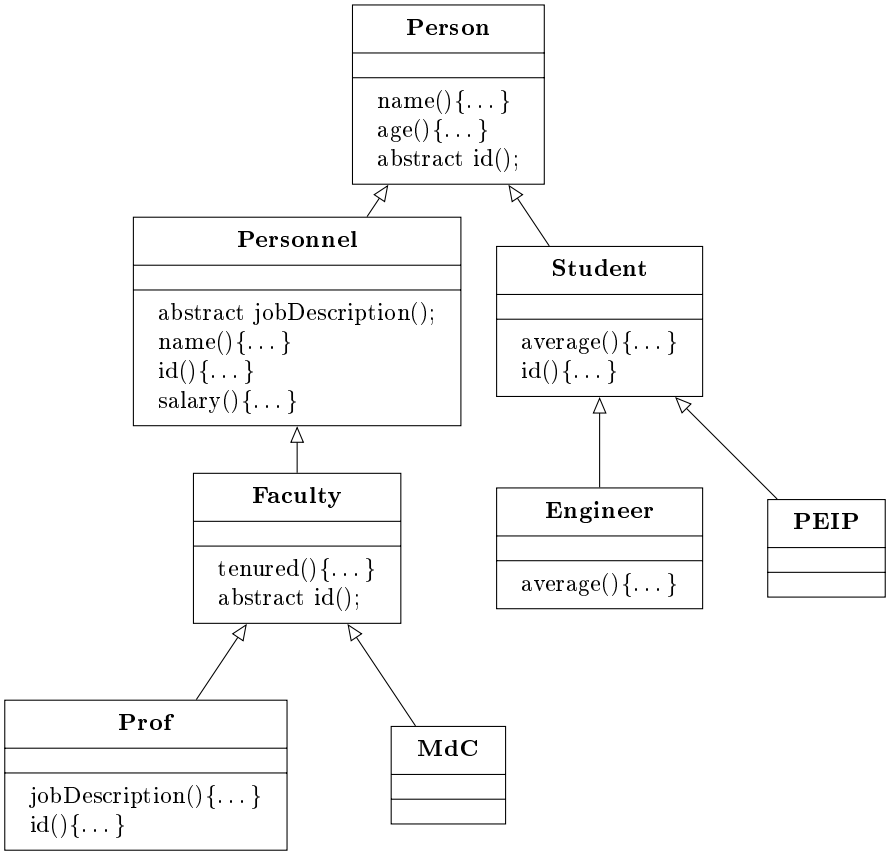
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Personnel`
- ☐ `Prof`
- ☐ `MdC`
- ☐ `PEIP`
- ☐ `Person`
- ☐ `Engineer`
- ☐ `Faculty`
- ☐ `Student`

+86/1/50+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ `protected`
- ☐ `public`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

☐ `something` ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `something` ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `something` ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut implémenter une interface

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

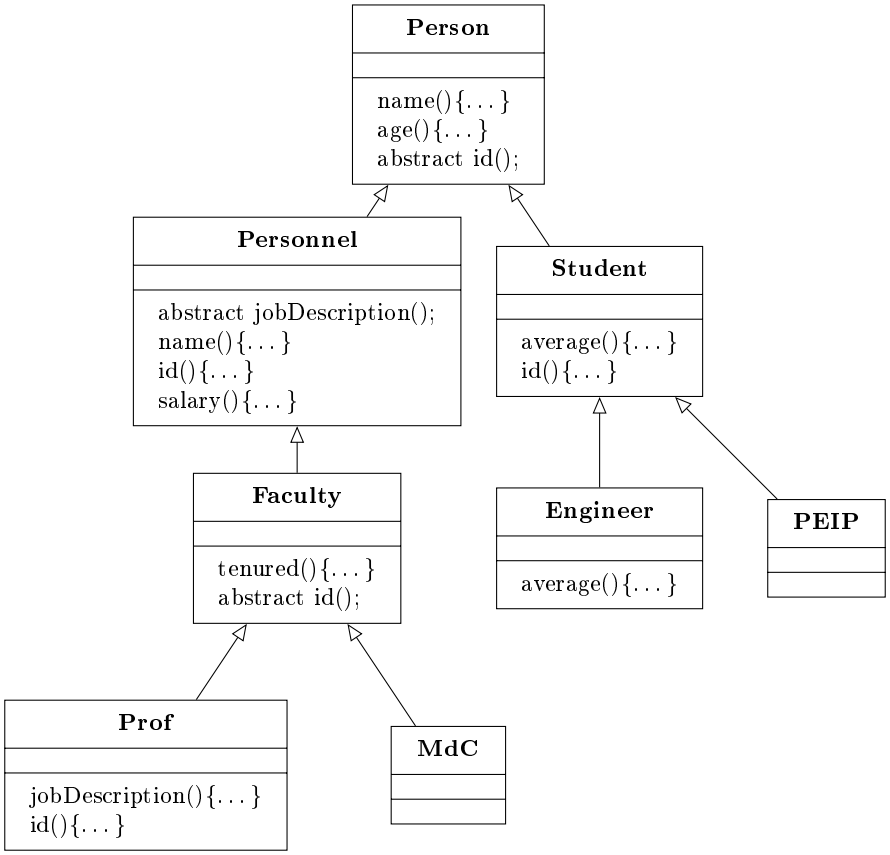
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Faculty`
- ☐ `Student`
- ☐ `PEIP`
- ☐ `Engineer`
- ☐ `MdC`
- ☐ `Person`
- ☐ `Personnel`
- ☐ `Prof`

+87/1/40+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

☐ Erreur de compilation

☐ Affiche "Done"

☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche "Done"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ `public`
- ☐ package-private
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des méthodes privées

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

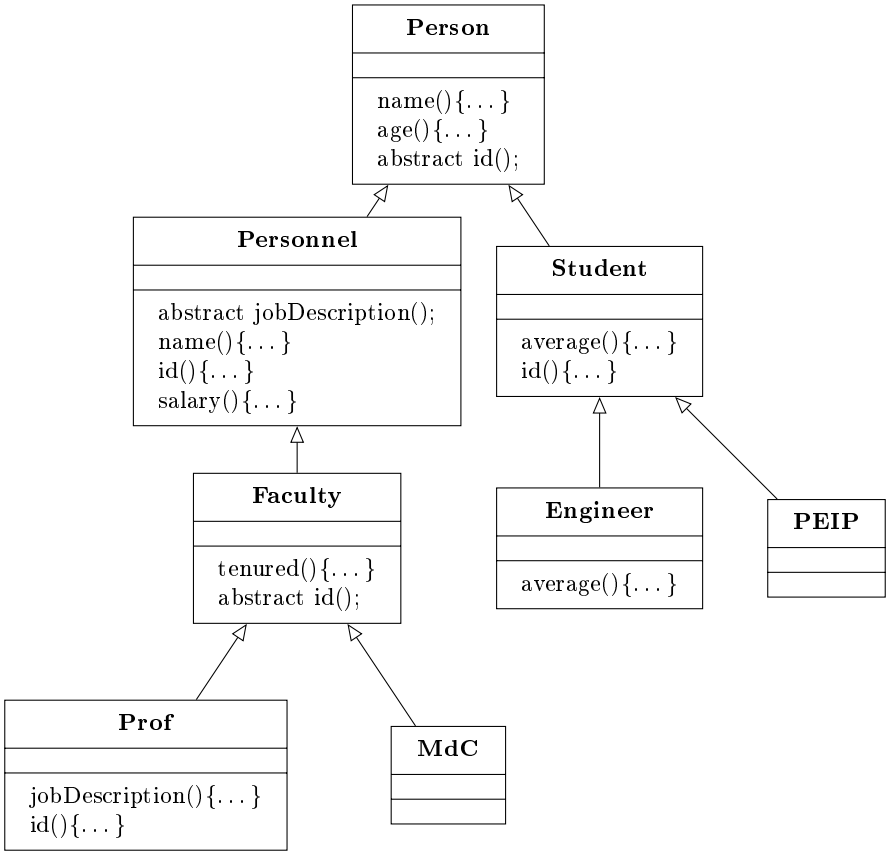
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

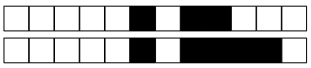
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Prof`
- ☐ `MdC`
- ☐ `Student`
- ☐ `Personnel`
- ☐ `PEIP`
- ☐ `Person`
- ☐ `Engineer`
- ☐ `Faculty`



+88/1/30+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

☐ Affiche "Caught exception" et "Finallied exception"

☐ Erreur d'exécution

☐ Affiche seulement "Finallied exception"

☐ Erreur de compilation

☐ Affiche seulement "Caught exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Done"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ `public`
- ☐ package-private
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `something` ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `something` ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre une classe abstraite

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

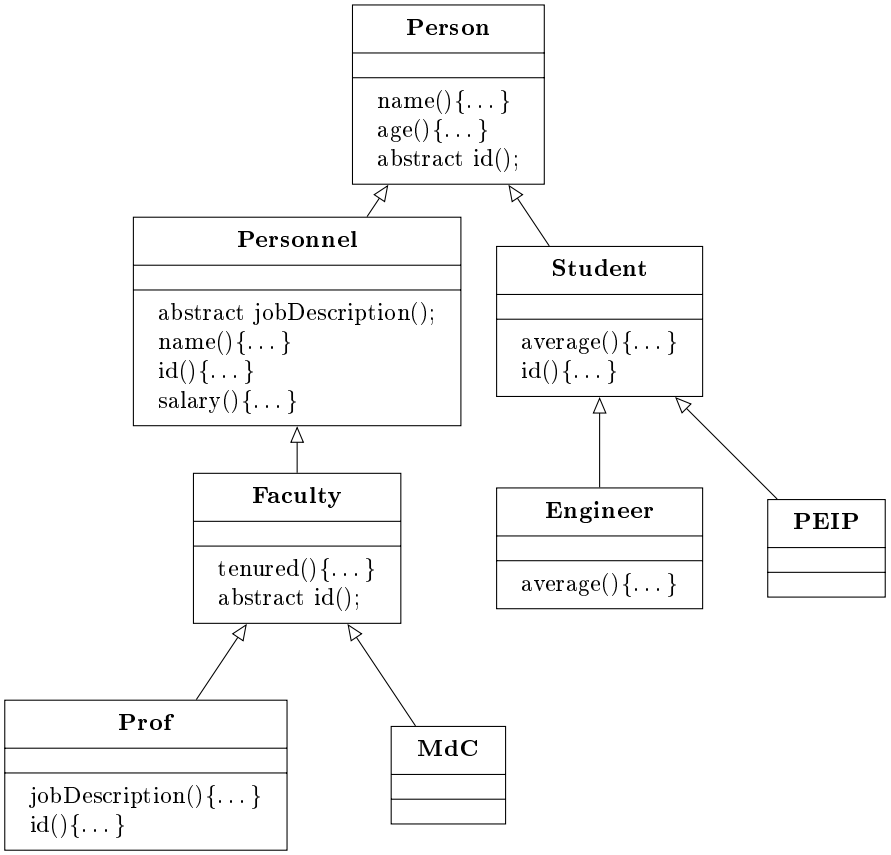
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

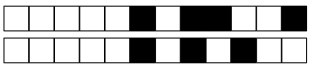
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Engineer`
- ☐ `MdC`
- ☐ `Person`
- ☐ `Faculty`
- ☐ `Prof`
- ☐ `PEIP`
- ☐ `Student`
- ☐ `Personnel`



+89/1/20+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

☐ Affiche "Done"

☐ Erreur de compilation

☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ `public`
- ☐ package-private
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

☐ `whatever` ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `whatever` ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `whatever` ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

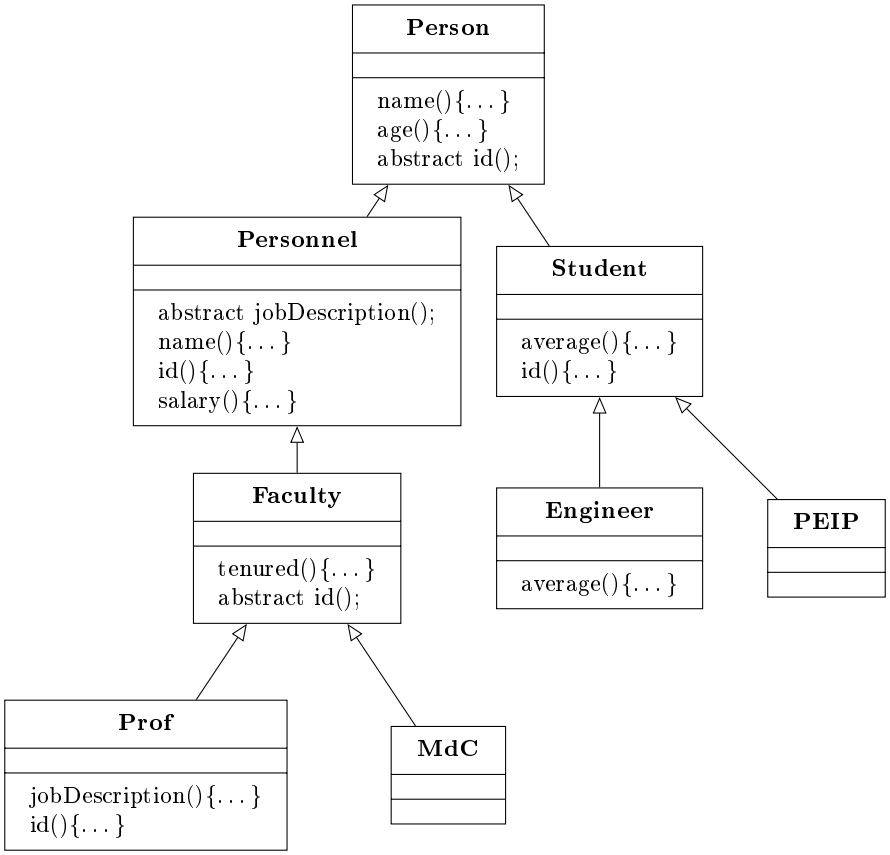
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Personnel`
- ☐ `Engineer`
- ☐ `PEIP`
- ☐ `Faculty`
- ☐ `Student`
- ☐ `Person`
- ☐ `Prof`
- ☐ `MdC`

+90/1/10+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Done"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ package-private
- ☐ `protected`
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

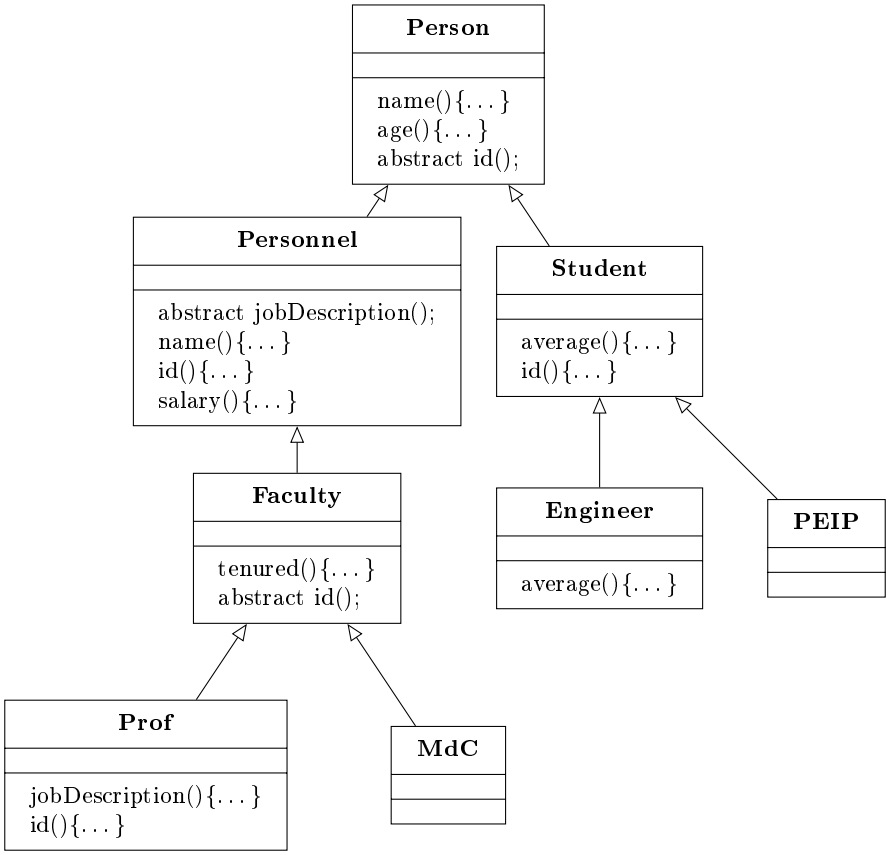
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Student`
- ☐ `PEIP`
- ☐ `Personnel`
- ☐ `Faculty`
- ☐ `Prof`
- ☐ `Engineer`
- ☐ `Person`
- ☐ `MdC`

+91/1/60+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

☐ Erreur de compilation

☐ Affiche "Done"

☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ package-private
- ☐ `protected`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `something` ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `something` ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

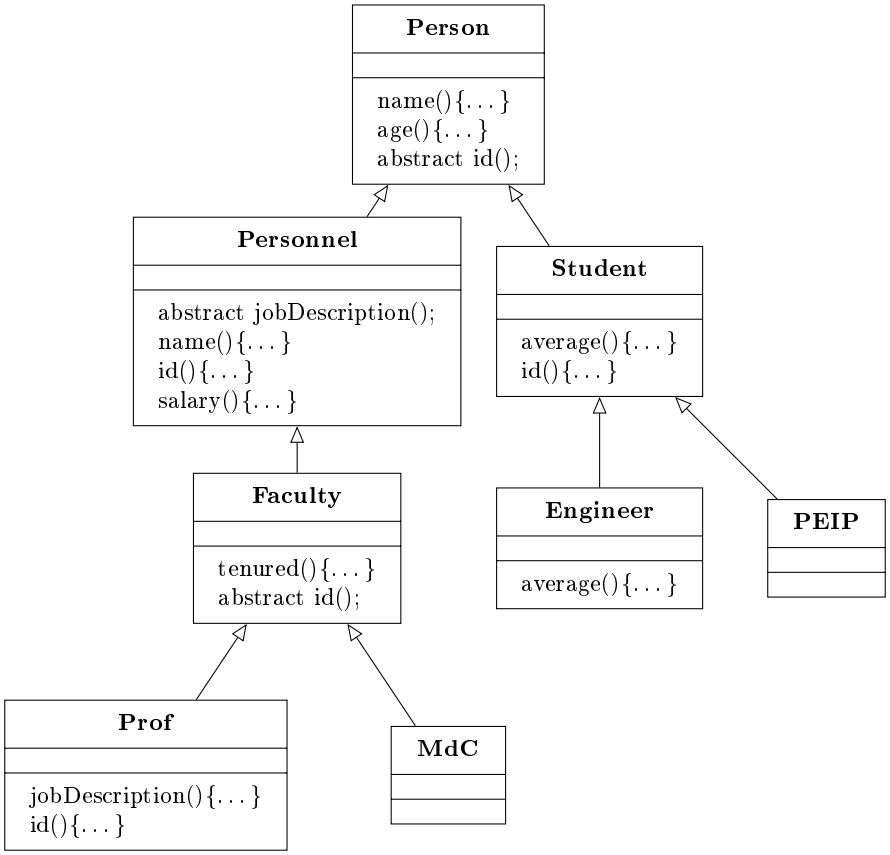
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `PEIP`
- ☐ `Personnel`
- ☐ `Engineer`
- ☐ `Prof`
- ☐ `MdC`
- ☐ `Person`
- ☐ `Faculty`
- ☐ `Student`

+92/1/50+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ *indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.*
Dans tout le code, les `package` *et les* `import` *sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.*

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ package-private
- ☐ `protected`
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

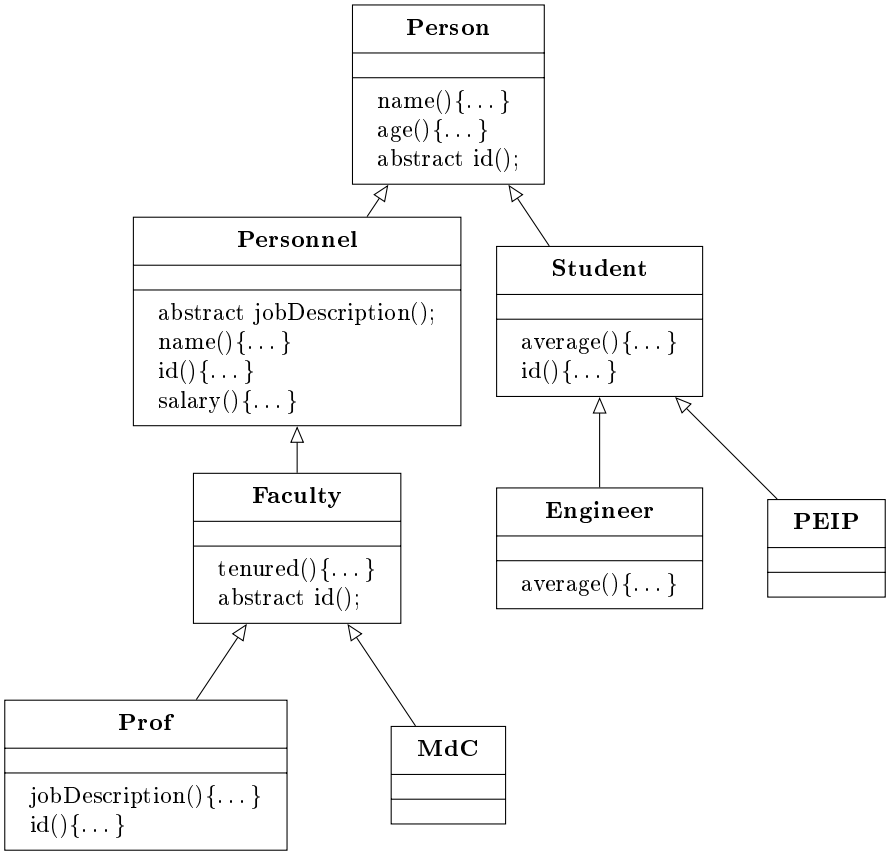
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `PEIP`
- ☐ `Personnel`
- ☐ `Student`
- ☐ `Engineer`
- ☐ `Prof`
- ☐ `MdC`
- ☐ `Person`
- ☐ `Faculty`

+93/1/40+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche "Done"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ package-private
- ☐ `public`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `something` ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `something` ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

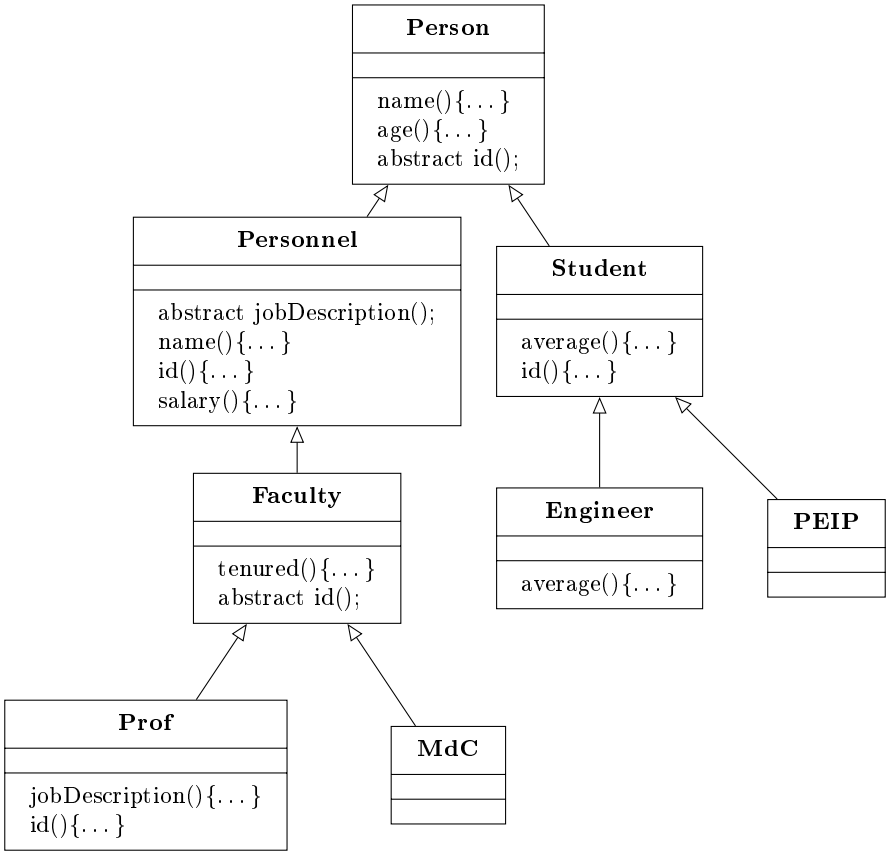
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

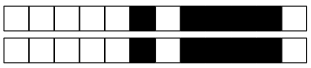
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Prof`
- ☐ `Faculty`
- ☐ `MdC`
- ☐ `Student`
- ☐ `Person`
- ☐ `Engineer`
- ☐ `PEIP`
- ☐ `Personnel`



+94/1/30+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `private`
- ☐ `protected`
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut implémenter une interface

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

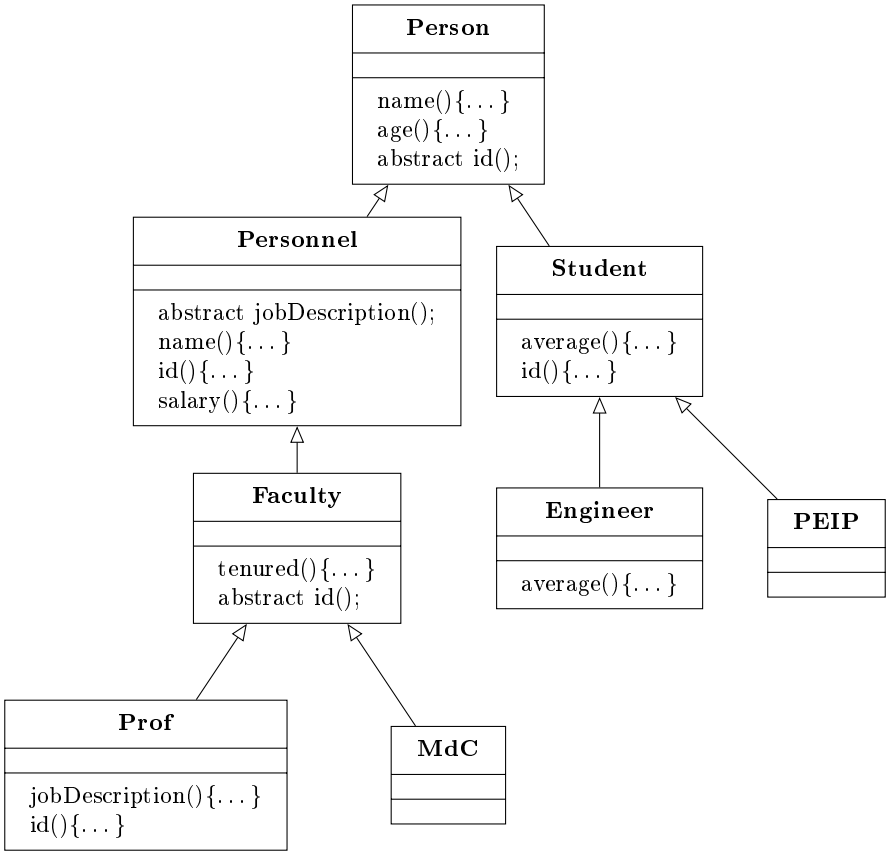
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

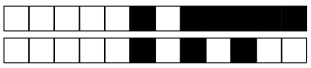
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `MdC`
- ☐ `Personnel`
- ☐ `Faculty`
- ☐ `Prof`
- ☐ `Student`
- ☐ `Engineer`
- ☐ `Person`
- ☐ `PEIP`



+95/1/20+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Done"
- ☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Done"
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ package-private
- ☐ `protected`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `something` ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `something` ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

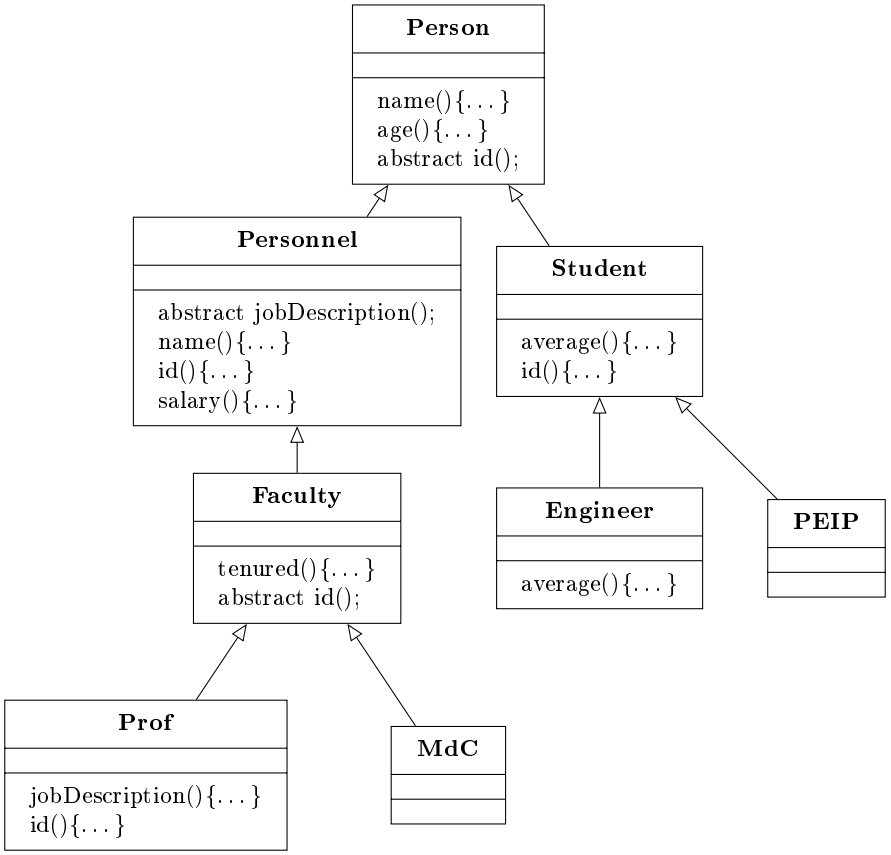
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Personnel`
- ☐ `MdC`
- ☐ `PEIP`
- ☐ `Student`
- ☐ `Person`
- ☐ `Prof`
- ☐ `Faculty`
- ☐ `Engineer`

+96/1/10+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Done"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `private`
- ☐ package-private
- ☐ `public`
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des méthodes privées
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe peut étendre une classe abstraite

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

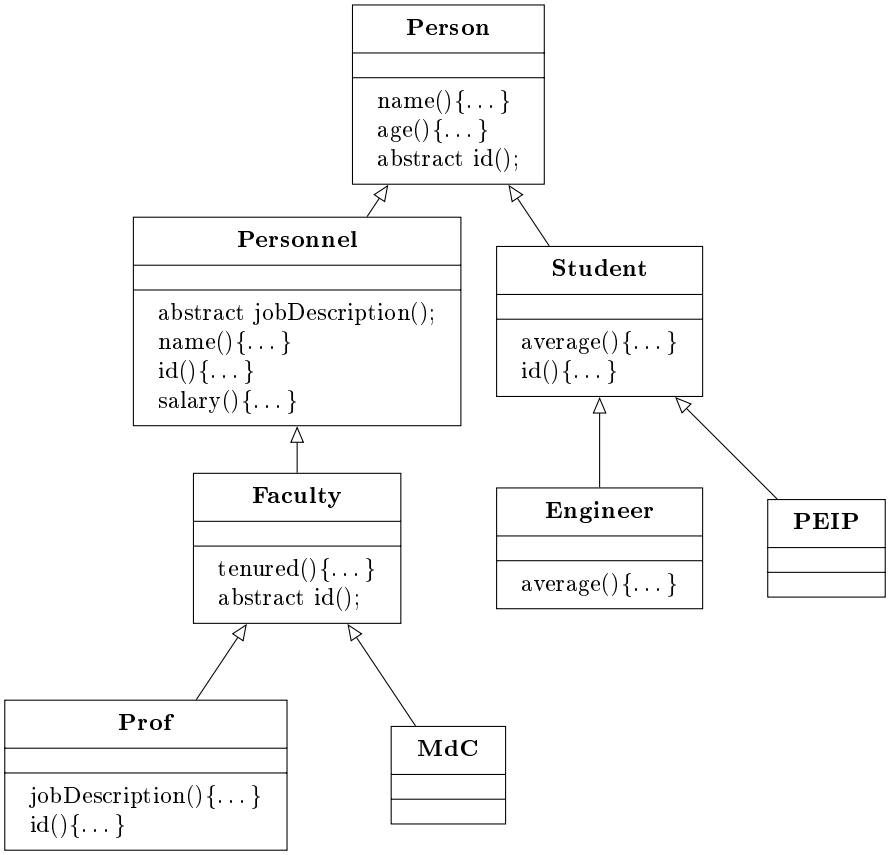
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Prof`
- ☐ `MdC`
- ☐ `Student`
- ☐ `Personnel`
- ☐ `Faculty`
- ☐ `Person`
- ☐ `Engineer`
- ☐ `PEIP`

+97/1/60+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ package-private
- ☐ `public`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

☐ `something` ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `something` ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `whatever` ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

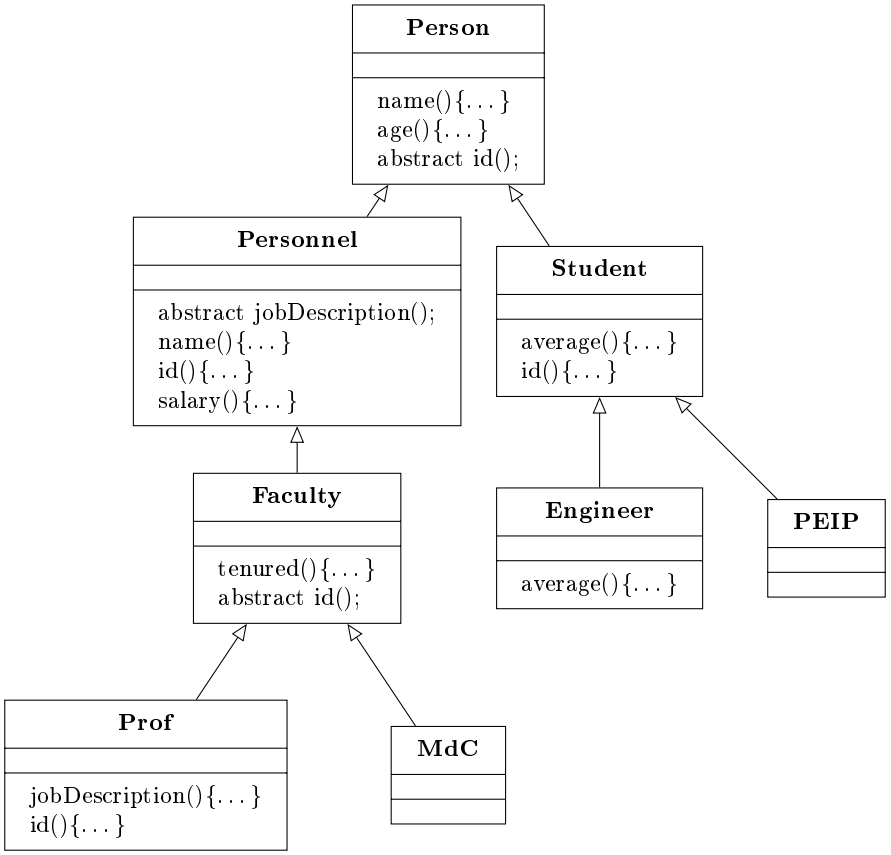
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Prof`
- ☐ `Person`
- ☐ `Faculty`
- ☐ `MdC`
- ☐ `Student`
- ☐ `PEIP`
- ☐ `Personnel`
- ☐ `Engineer`

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Done"
- ☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Done"
- ☐ Erreur de compilation

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ `private`
- ☐ `public`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `whatever` ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `whatever` ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

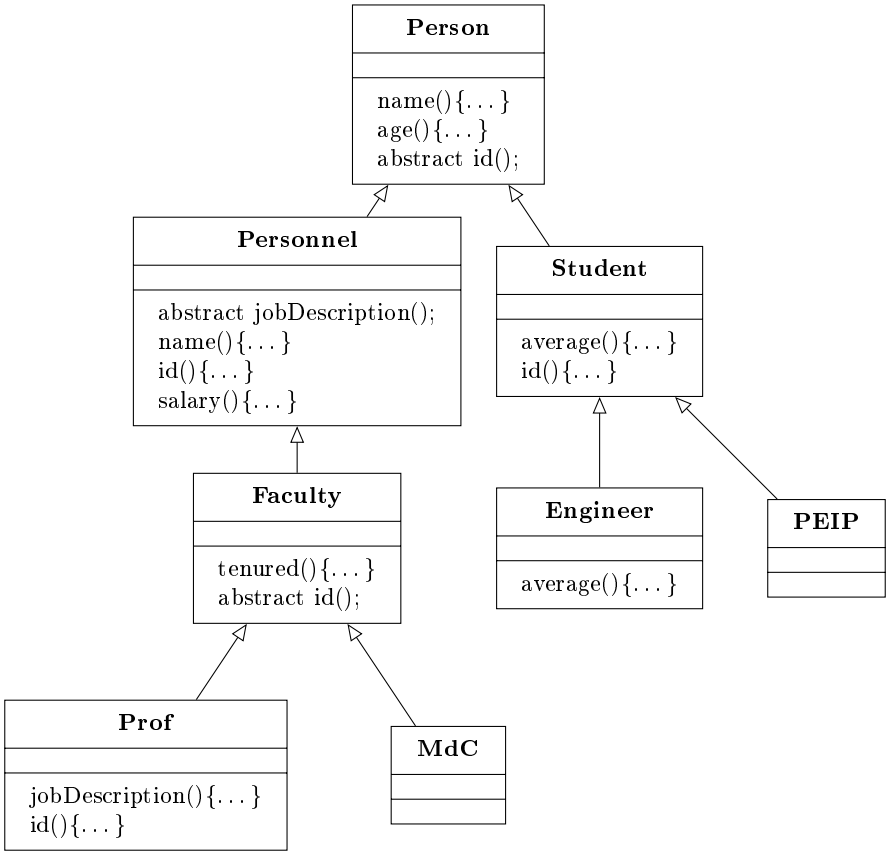
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Person`
- ☐ `Personnel`
- ☐ `MdC`
- ☐ `Engineer`
- ☐ `PEIP`
- ☐ `Student`
- ☐ `Prof`
- ☐ `Faculty`

+99/1/40+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

☐ Erreur de compilation

☐ Affiche "Caught exception" et "Finallied exception"

☐ Erreur d'exécution

☐ Affiche seulement "Finallied exception"

☐ Affiche seulement "Caught exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Affiche "Done"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ `private`
- ☐ `public`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

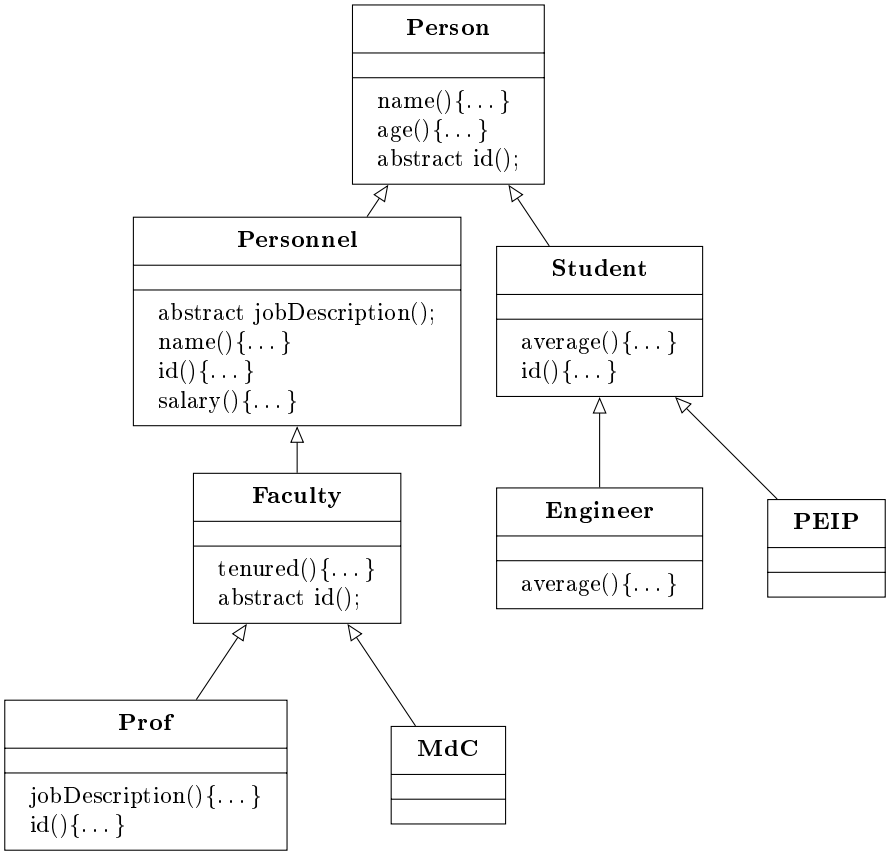
☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Student`
- ☐ `PEIP`
- ☐ `Person`
- ☐ `MdC`
- ☐ `Prof`
- ☐ `Personnel`
- ☐ `Engineer`
- ☐ `Faculty`

+100/1/30+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Done"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ package-private
- ☐ `public`
- ☐ `protected`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `something`
- ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

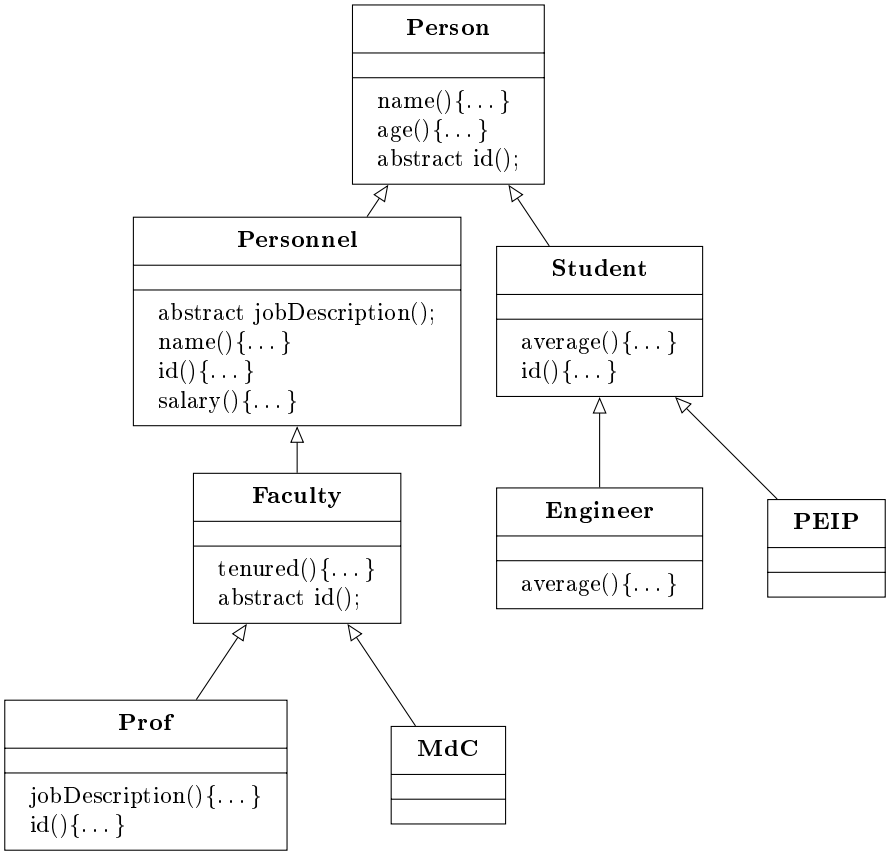
☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Personnel`
- ☐ `Engineer`
- ☐ `Faculty`
- ☐ `PEIP`
- ☐ `Person`
- ☐ `Student`
- ☐ `MdC`
- ☐ `Prof`

+101/1/20+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

☐ Erreur de compilation

☐ Affiche "Done"

☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Erreur d'exécution
- ☐ Affiche "Done"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ `public`
- ☐ `private`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `something`
- ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `something` ☐ `whatever`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `something` ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre plusieurs classes

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

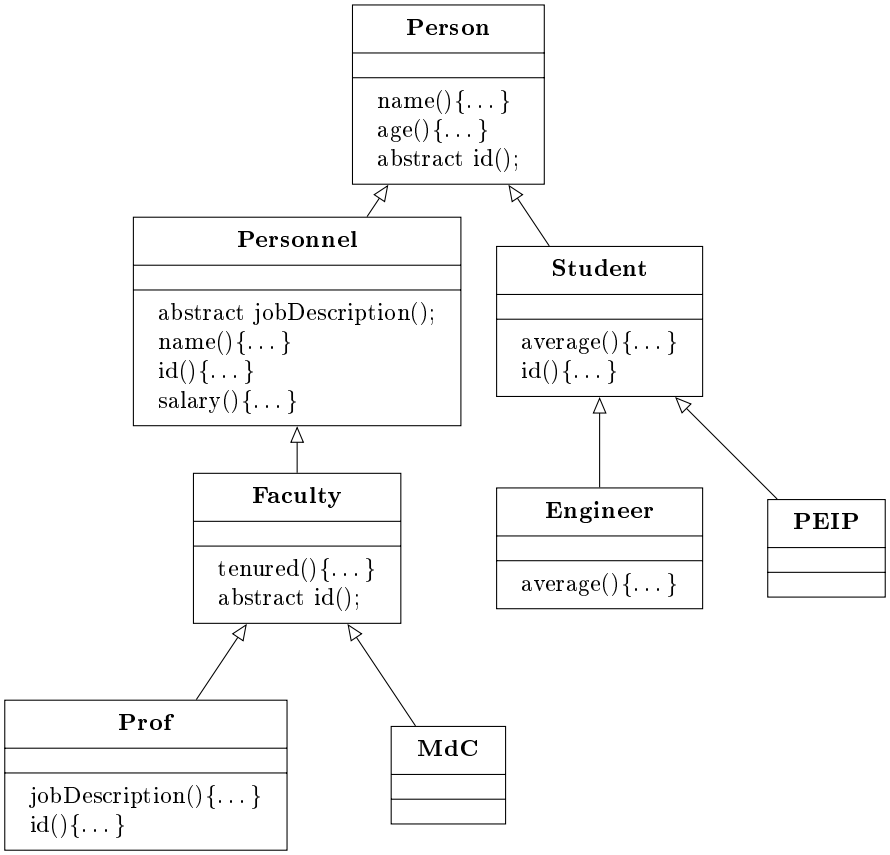
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Person`
- ☐ `MdC`
- ☐ `PEIP`
- ☐ `Personnel`
- ☐ `Faculty`
- ☐ `Prof`
- ☐ `Engineer`
- ☐ `Student`

+102/1/10+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur d'exécution

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Done"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ `private`
- ☐ `public`
- ☐ package-private

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `whatever`
- ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une interface peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

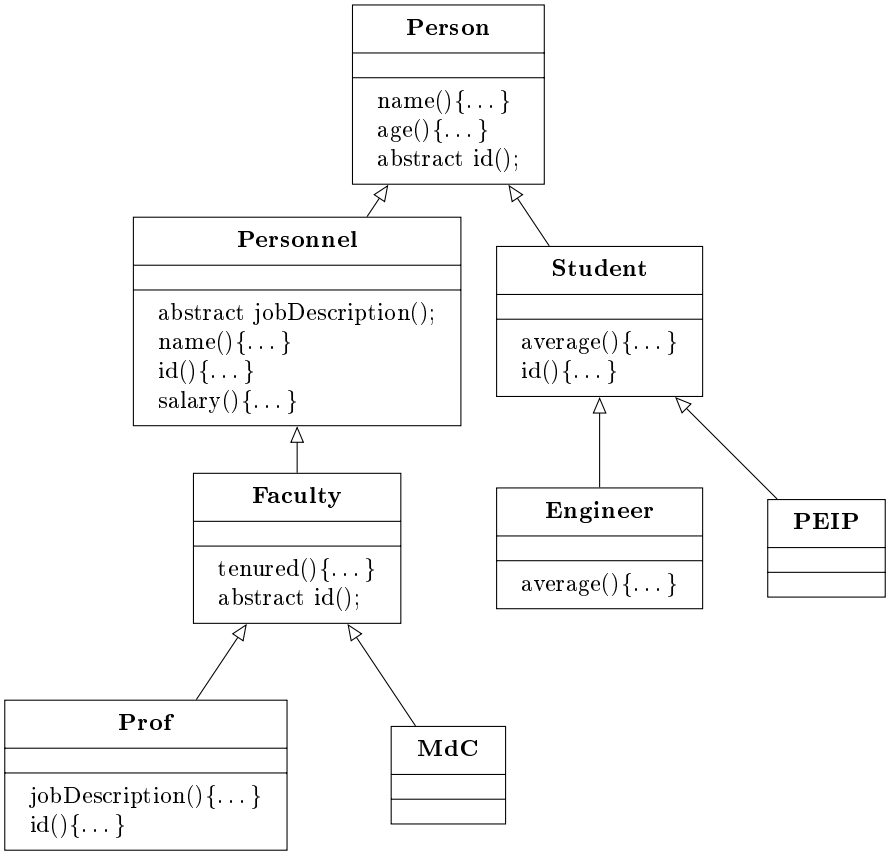
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Engineer`
- ☐ `Student`
- ☐ `MdC`
- ☐ `Person`
- ☐ `Prof`
- ☐ `PEIP`
- ☐ `Personnel`
- ☐ `Faculty`

+103/1/60+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Caught exception"
- ☐ Erreur de compilation
- ☐ Erreur d'exécution

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche "Done"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `protected`
- ☐ `private`
- ☐ package-private
- ☐ `public`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des constructeurs
- ☐ Une interface peut contenir des méthodes privées

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

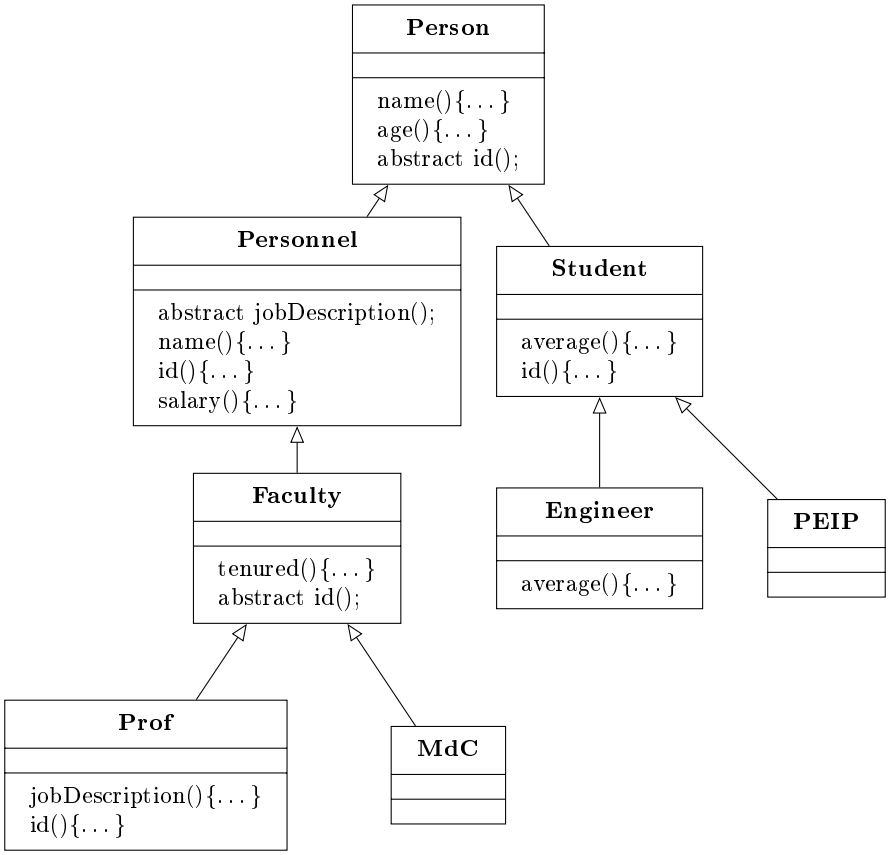
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Student`
- ☐ `Person`
- ☐ `MdC`
- ☐ `Prof`
- ☐ `PEIP`
- ☐ `Engineer`
- ☐ `Faculty`
- ☐ `Personnel`

+104/1/50+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

☐ pourrait être instanciée par `new ClasseDonnee()`

☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {
    private ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 3 ⊕ La classe donnée

```
abstract class ClasseDonnee {
    ClasseDonnee() {}
}
```

☐ pourrait être sous-classée par `extends ClasseDonnee`

☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche seulement "Finallied exception"
- ☐ Affiche seulement "Caught exception"

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Delta {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Delta delta = new Delta();
        try {
            delta.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
            return;
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Alpha {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
      Alpha alpha = new Alpha();
      alpha.throwup();
      System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Erreur de compilation
- ☐ Affiche "Done"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ `private`
- ☐ package-private
- ☐ `protected`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

- ☐ `whatever`
- ☐ `something`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

- ☐ `whatever`
- ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

- ☐ `something`
- ☐ `whatever`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une classe abstraite peut contenir des constructeurs
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

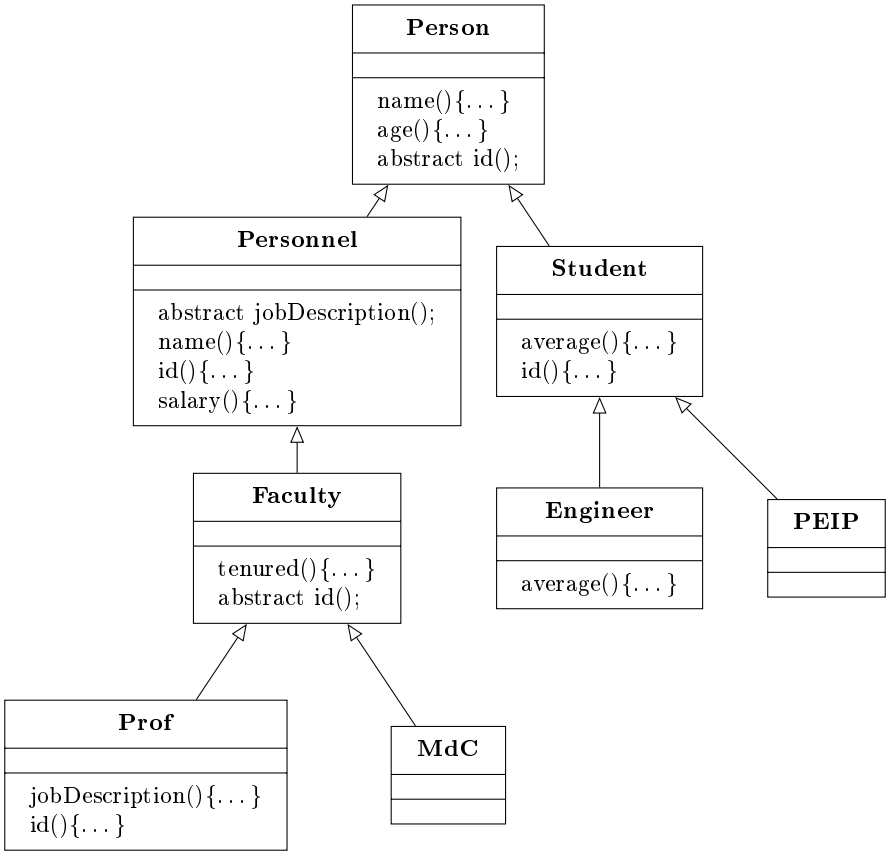
☐
```
ComparePerson byAge = new ComparePersonByAge();
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge
       = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Personnel`
- ☐ `Person`
- ☐ `MdC`
- ☐ `Student`
- ☐ `PEIP`
- ☐ `Prof`
- ☐ `Faculty`
- ☐ `Engineer`

+105/1/40+

QCM **TEST**

Introduction à la programmation orientée objet
9/01/2020

Nom et prénom :

.....

Groupe : .....

Cochez les cases en mettant une ×.
Le symbole ⊕ indique que la question peut avoir zéro, une ou plusieurs bonnes réponses. Pour ces questions, cocher une bonne réponse apporte des points positifs ; cocher une mauvaise réponse peut apporter des points négatifs.
Dans tout le code, les `package` et les `import` sont censés être correctement déclarés. Toute classe est supposée être dans le bon package, dans le bon fichier, avec les bons import.

Question 1 ⊕ La classe donnée

```
final class ClasseDonnee {
    ClasseDonnee() {}
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 2 ⊕ La classe donnée

```
class ClasseDonnee {}
```

- ☐ pourrait être instanciée par `new ClasseDonnee()`
- ☐ pourrait être sous-classée par `extends ClasseDonnee`

Question 3 ⊕ La classe (enum) donnée

```
enum ClasseDonnee {
    INSTANCE;
}
```

- ☐ pourrait être sous-classée par `extends ClasseDonnee`
- ☐ pourrait être instanciée par `new ClasseDonnee()`

Question 4 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Beta {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Beta beta = new Beta();
        beta.throwup();
        System.out.println("Done");
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Done"
- ☐ Erreur de compilation

Question 5 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Gamma {
    public void throwup() throws Exception {
        throw new Exception();
    }
    public static void main(String[] args) {
        Gamma gamma = new Gamma();
        try {
            gamma.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur d'exécution
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur de compilation
- ☐ Affiche seulement "Caught exception"

Question 6 ⊕

Soit le code ci-dessous ; que se passe-t-il ?

```
public class Epsilon {
    public void throwup() throws RuntimeException {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        Epsilon epsilon = new Epsilon();
        try {
            epsilon.throwup();
        } catch (Exception e) {
            System.out.println("Caught exception");
        } finally {
            System.out.println("Finallied exception");
        }
    }
}
```

- ☐ Erreur de compilation
- ☐ Affiche "Caught exception" et "Finallied exception"
- ☐ Affiche seulement "Finallied exception"
- ☐ Erreur d'exécution
- ☐ Affiche seulement "Caught exception"

Question 7 Soit le code :

```
interface Somethingable {
    void something();
}
```

```
interface Whateverable extends Somethingable {
    void whatever();
}
```

```
class TheThing implements Whateverable{
    PPP void something() {
        System.out.println("Somethinging");
    }

    PPP void whatever() {
        System.out.println("Whatevering");
    }
}
```

```
class Doofer {
    void doo(Somethingable sable) {
        sable.XXX();
    }

    void doo(Whateverable sable) {
        sable.YYY();
    }

    void fer(TheThing tt) {
        tt.ZZZ();
    }

    public static void main(String[] args) {
        Doofer doofer = new Doofer();
        TheThing the = new TheThing();
        doofer.doo(the);
        doofer.fer(the);
    }
}
```

Par quels niveaux d'accès pourrait-on remplacer `PPP` ?

- ☐ `public`
- ☐ package-private
- ☐ `protected`
- ☐ `private`

Question 8 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `XXX` ?

☐ `something` ☐ `whatever`

Question 9 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `YYY` ?

☐ `whatever` ☐ `something`

Question 10 ⊕ Pour le code de la question précédente, par quelles méthodes pourrait-on remplacer `ZZZ` ?

☐ `whatever` ☐ `something`

Question 11

Rappel :

- la forme cartésienne d'un nombre complexe s'écrit $x + yi$, où x est la partie réelle et y est la partie imaginaire. $i^2 = -1$

Java n'a pas de nombres complexes. Le stagiaire de l'année dernière avait développé la classe donnée :

```
public class Complex {
    public double re;
    public double im;

    public Complex(double a, double b) {
        re = a;
        im = b;
    }

    public double getRe() {
        return re;
    }

    public void setRe(double re) {
        this.re = re;
    }

    public double getIm() {
        return im;
    }

    public void setIm(double im) {
        this.im = im;
    }

    public void add(Complex c) {
        re += c.re;
        im += c.im;
    }

    public void mult(Complex c) {
        re = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
    }

    @Override
    public String toString() {
        return "(" + re + ", " + im + ")";
    }
}
```

A la fin du stage, la responsable de stage trouve le code pas adapté à l'usage – tout d'abord, elle aurait voulu que la classe `Complex` soit immuable, à l'image de `Integer` ou `Double`. Evidemment il revient à vous de le faire. Alors, faites-le.

Rappel : une classe `A` est *immuable* si

- un objet de classe `A` ne peut plus être modifié un fois construit
- `A` ne peut pas avoir de sous-classe, c'est à dire `...extends A...` est interdit

☐0 ☐1 ☐2 ☐3 ☐4

Question 12

Rappel :

- la forme polaire d'un nombre complexe s'écrit (r, θ), où r est le rayon (radius) et θ est l'angle (angle). θ est la lettre grecque theta
- la conversion entre les formes cartésienne et polaire se fait par

$$x = r\cos(\theta) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(\theta) \qquad \theta = \tan^{-1}\frac{y}{x}$$

  En Java ce dernier est calculé par `theta = Math.atan2(y, x)`
- addition (de formes cartésiennes) : $(x, y\imath) + (u, v\imath) = (x + u, (y + v)\imath)$
- multiplication (de formes cartésiennes) : $(x, y\imath) * (u, v\imath) = (xu - yv, (xv + yu)\imath)$
- multiplication (de formes polaires) : $(r, \theta) * (s, \phi) = (r * s, \theta + \phi)$
- pour simplifier, nous ne traitons pas les cas limites, par exemple où $r = 0$, et nous nous limitons aux seules opérations ci-dessus.

```
public class Cartesian {

    public Cartesian(double a, double b) {...}

    public Cartesian add(Cartesian c) {...}

    public Cartesian add(Polar p) {...}

    public Cartesian mult(Cartesian c) {...}

    public Cartesian mult(Polar p) {...}

    @Override
    public String toString() {
        return "Cartesian + (" + re + ", " + im + ")";
    }
}
```

Le code donné ici avait été dévéloppé par un deuxième stagiaire, guère plus doué, pour représenter la forme cartésienne d'un nombre complexe (l'implémentation des méthodes est omise ici par souci d'économiser du papier). Il avait aussi développé la classe `Polar`, avec les mêmes méthodes mais renvoyant des objets de type `Polar`.

Hélas, la responsable est toujours très exigente, elle trouve qu'il y a beaucoup trop de redondances de code (pourquoi deux méthodes `Cartesian#add` par exemple ? Une seule devrait suffir). Pas de bol, c'est encore à vous de refaire l'application. Vous pouvez refactoriser les classes `Cartesian` et `Polar`, et vous avez droit à toute autre modification de l'application que vous pouvez juger nécessaire. Implémentez toutes les méthodes de toutes vos classes.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6

Question 13 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 14 Continuez votre réponse à la question précédente ici si nécessaire. ☐0 ☐0

Question 15 Pour la question précédente, développez la méthode `Polar#equals`.

☐0 ☐1 ☐2 ☐3 ☐4

Question 16 ⊕ Parmi les affirmations ci-dessous, cocher celles qui sont vraies :

- ☐ Une classe peut étendre plusieurs classes
- ☐ Une interface peut contenir des constructeurs
- ☐ Une classe peut étendre une classe abstraite
- ☐ Une classe abstraite peut implémenter une interface
- ☐ Une interface peut contenir des méthodes privées
- ☐ Une classe peut étendre une classe et implémenter plusieurs interfaces
- ☐ Une classe abstraite peut contenir des constructeurs

Question 17 ⊕ Soit le code ci-dessous :

```
@FunctionalInterface
interface ComparePerson {
    /**
     * @return negative if p1 < p2, 0 if p1 = p2,
     * positive if p1 > p2
     */
    int compare(Person p1, Person P2);
}
```

```
class ComparePersonByAge implements ComparePerson {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

```
class Person {
    private int age;
    private String name;

    Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}
```

```
class Main {
    public static void main(String[] args) {

        // code extract here

        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 24)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 23)));
        System.out.print(byAge.compare(
            new Person("Fred", 23),
            new Person("Not Fred", 22)));
    }
}
```

Lequel des extraits de code ci-dessous donne le résultat `-101` quand substitué dans la méthode `main` :

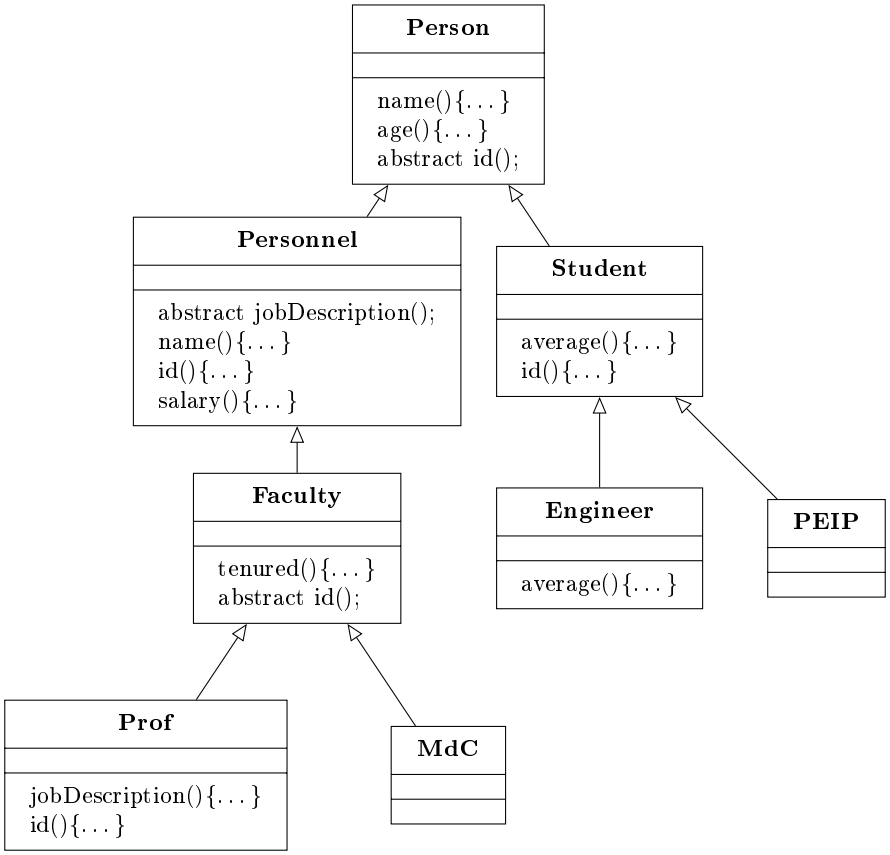
☐
```
ComparePerson byAge = new ComparePersonByAge() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
};
```

☐
```
ComparePerson byAge
        = new ComparePersonByAge::compare;
```

☐
```
ComparePerson byAge = (p1, p2) -> p1.age - p2.age;
```

☐
```
ComparePerson byAge = new ComparePersonByAge();
```

Question 18 ⊕ Soit l'organisation ci-dessous de Polytech'Groland :



Les méthodes sont toutes en niveau d'accès package-private.
Pour que ce schéma compile, quelles classes doivent être déclarées `abstract` ?

- ☐ `Prof`
- ☐ `Engineer`
- ☐ `Student`
- ☐ `Faculty`
- ☐ `Person`
- ☐ `MdC`
- ☐ `PEIP`
- ☐ `Personnel`