Introduction à la Programmation Objet

3h, tous doucuments papier autorisés

23 janvier 2020

Veillez à bien reporter les numéros des questions auxquelles vous répondez. Une rédaction claire, propre et concise est attendue. N'hésitez pas à commenter votre code, sa lisibilité et le respect des bonnes pratiques seront pris en compte dans la notation, tout comme des éléments de réponse incomplets montrant votre compréhension de la programmation objet et/ou votre démarche adoptée face à une machine.

Le sujet se compose de 3 parties indépendantes, dont la plupart des questions peuvent être traitées séparémment. Le barème, sur 20, est indicatif.

La javadoc de l'interface List et de la classe LocalDate vous est fournie en annexe, elle est tirée de https ://docs.oracle.com/

1 Cours et compréhension (6 points)

1.1 Structure de classe

Soit le code suivant :

```
1  public class User {
2     private int age;
3     private String username;
4     private String role;
5
6     public User(int age, String username, String role) {
7        this.age = age;
8        this.username = username;
9        this.role = role;
10    }
11
12    public User(int age, String username) {
13       this(age, username, "suscriber");
14    }
15
16    public int getAge() {
17       return age;
18    }
```

```
19
20      public void setAge(int age) {
21          this.age = age;
22      }
23
24      public String getUsername() {
25          return username;
26      }
27
28      public boolean isAdult() {
29          return age >= 18;
30      }
31
32  }
```

Question 1 Désigner par leur numéros de lignes les parties du code correspondant à la déclaration d'attributs, aux constructeurs et aux méthodes.

Question 2 Quel est le nom de la classe ? Le nom complet du fichier ?

Question 3 Que signifie le mot-clé `private` à la ligne 2 ?

Question 4 Écrire une instruction permettant de créer une instance de cette classe.

Question 5 Écrire une instruction permettant d'en changer l'âge à 25.

Question 6 Que signifie le mot-clé `this` à la ligne 7 ?

Question 7 Que signifie le mot-clé `this` à la ligne 13 ?

1.2 Convention de code

Question 8 On vous demande d'écrire un programme gérant un jeu se déroulant en plusieurs manches. Parmi ceux-ci, quel nom de variable choisir pour désigner le score d'un joueur pour une manche ? Pourquoi ?

1. `score_d_une_manche`
2. `nb`
3. `currentpoints`
4. `scorePlayerRound`
5. `MonScoreCetteManche`

1.3 Debug

À l'execution d'un programme, l'exception suivante est levée :

```
1 Exception in thread "main" java.lang.NullPointerException
2     at cours.Mail.send(Mail.java:7)
3     at cours.Mail.main(Mail.java:14)
```

Question 9 Quel est le soucis ? Que faudrait-il aller regarder dans le code/comment le corriger ?

1.4 static

Soit une classe `Equation` contenant la méthode :

```
public static Double solver(String equation, ArrayList<Double> parameters)
```

Question 10 Comment appeler cette méthode depuis une autre classe ?

1.5 Héritage

Soit la classe Vehicule :

```
1  public class Vehicule {
2      protected String immatriculation;
3      private int puissance;
4
5      public Vehicule(String immatriculation, int puissance) {
6          this.immatriculation = immatriculation;
7          this.puissance = puissance;
8      }
9
10     public String getImmatriculation() {
11         return immatriculation;
12     }
13
14     public void setImmatriculation(String immatriculation) {
15         this.immatriculation = immatriculation;
16     }
17
18     public int getPuissance() {
19         return puissance;
20     }
21
22 }
```

et la classe Voiture :

```
1  public class Vehicule {
2     protected String immatriculation;
3     private int puissance;
4
5     public Vehicule(String immatriculation, int puissance) {
6        this.immatriculation = immatriculation;
7        this.puissance = puissance;
8     }
9
10    public String getImmatriculation() {
11       return immatriculation;
12    }
13
14    public void setImmatriculation(String immatriculation) {
15       this.immatriculation = immatriculation;
16    }
17
18    public int getPuissance() {
19       return puissance;
20    }
21 }
```

Question 11 Après les déclarations suivantes (dans un `main`) :

```
1        Vehicule moto = new Vehicule("ZZZ", 4);
2        Voiture voiture = new Voiture("ABC", 6, 5);
3        Vehicule def = new Voiture("DEF", 6, 3);
```

que donneraient les instructions suivantes ? Expliquer brièvement pourquoi.

1. `System.out.println(voiture.getImmatriculation());`
2. `System.out.println(voiture.getNbPortes());`
3. `System.out.println(moto.getImmatriculation());`
4. `System.out.println(moto.getNbPortes());`
5. `System.out.println(def.getImmatriculation());`
6. `System.out.println(def.getNbPortes());`
7.
```
Vehicule[] mesVehicules = {moto, voiture, def};
for (int i =0; i<3; i++){
   System.out.println(mesVehicules[i].getPuissance());
}
```

2 Gestion de listes d'initiés (10 points)

Un *initié* est une personne qui peut se trouver en possession d'informations privilégiées pouvant avoir un impact sur la spéculation boursière. Il est alors limité dans la manipulation du marché, la divulgation d'informations et le conseil pour certaines périodes et sujets. Par exemple, les membres du conseil d'administration d'une entreprise connaissent les chiffres d'affaires pour l'année avant qu'ils ne soient publiés. Entre la réunion où ils ont appris les résultats et le moment où ils sont divulgués au public, il ne peuvent pas acheter ou vendre d'action pour cette entreprise. Y contrevenir constitue un "délit d'initié".

Chaque entreprise cotée en bourse est tenue de tenir une liste d'initiés informés pour chaque sujet concerné, sur les trois dernières années. Vous écrirez dans cette exercice un petit programme java capable de gérer de telles informations [1].

On suppose ici le fonctionnement suivant :

— une *information* a un titre et une date de divulgation au public, après laquelle elle n'est plus confidentielle ; elle est toujours révélée à des initiés lors d'une *réunion*

— une *réunion* a une date (on ne prend pas l'heure en compte) et porte sur une (et une seule) *information*

— un *initié* a un nom, un rôle, et une liste d'informations privilégiées dont il a ou a eu connaissance lorsqu'elles étaient confidentielles

— l'*entreprise* maintient la liste des inités, des informations et des réunions.

— les absences aux réunions ne sont pas gérées : on suppose que les participants effectifs seront toujours exactement les participants prévus.

Le programme sera composé de 4 classes : `Information`, `Reunion`, `Initie` et `Entreprise`. La classe `Entreprise` contient le `main` et son code vous est donné en annexe, ainsi avec la sortie console apparaissant à son execution.
Les squelettes (à respecter !) des classes `Information` et `Initie` vous sont également fournis en annexe
Les *dates* seront gérées par la classe `LocaleDate`, dont l'API vous est donné en annexe.

Question 1 Écrire le constructeur `public Initie(String nom, String role)` de la classe `Initie`. À sa création, un initié ne connait encore aucune information privilégiée.

Question 2 Écrire le constructeur `public Information(String titre, LocalDate dateDivulgation)` de la classe `Information`.

Question 3 En utilisant le constructeur précédent et une méthode de le classe `LocalDate`, écrire le constructeur `public Information(String titre, int jour, int mois, int annee)` de la classe `Information`, fixant la date au jour, mois et année donnés en paramètres.

1. Ce programme devrait ensuite être intégré à une structure plus globale, permettant nottament de stocker les informations dans une base de donnée et un accès utilisateur.

Question 4 Écrire la méthode `public void prendConnaissance(Information sujet)` de la classe `Initie`, qui ajoute le sujet passé en paramètre aux sujets connus par l'initié.

Question 5 Écrire la méthode `public String toString()` de la classe `Initie`, qui devra afficher les informations sur l'initié sous la forme suivante (des exemples vous sont fournis dans le résultat de l'execution du programme, en annexe) :

```
Nom, rôle connait les sujets :
- titre du premier sujet
- titre du deuxième sujet
- titre du troisième sujet
- ...
```

Question 6 Cette méthode est précédée de l'annotation `@Override`, pourquoi ?

Question 7 Écrire la classe `Reunion`, contenant :

- des attributs `date` (la date à laquelle la réunion se tient), `initiesPresents` (la liste des initiés présents à cette réunion) et `sujet` (l'information, de type `Information`, sur laquelle la réunion porte).
- un constructeur `public Reunion(LocalDate date, Information sujet)`, qui initialise la date et le sujet de la réunion à ceux passés en argument et la liste d'initiés présent à une liste vide.

Question 8 Écrire la méthode `public void ajouteInitie(Initie initie)` de la classe `Reunion`, qui ajoute l'initié passé en paramètre à la liste des présents à la réunion.

Question 9 Écrire le constructeur `public Reunion(Information sujet)` de la classe `Reunion`, qui crée une réunion pour le jour courrant (le "aujourd'hui" de la machine sur laquelle le code est exécuté). Pour avoir tous les points, ce constructeur devra appeler le constructeur de la question 7.

Question 10 Écrire le constructeur`public Reunion(LocalDate date, Information sujet, List<Initie> presents)` de la classe `Reunion`, qui crée une réunion pour la date et le sujet donné, avec la liste de présents passée en argument. Pour avoir tous les points, ce constructeur devra appeler le constructeur de la question 7.

Question 11 Écrire la méthode `public boolean estPassee()` de la classe `Reunion`, qui renvoie vrai si et seulement si la date de la réunion est antérieure ou égale à la date du jour (on considère que les réunions du jour courrant se sont toutes déroulées).

Question 12 Écrire la méthode `public boolean estConfidentielle()` de la classe `Reunion`, qui renvoie vrai si et seulement si l'information dont il est question lors de la réunion est confidentielle : c'est-à-dire si la date de divulgation au public du sujet est dans l'avenir par rapport à la date à laquelle se tient la réunion.

Question 13 Écrire la méthode `public void ajouteInformationConfidentiellesAuxPresents()` de la classe `Reunion`, qui teste si le sujet de la réunion est confidentiel, auquel cas elle ajoute ce sujet à la liste des sujets connus de chacun des initiés présents et ne fait rien sinon.

Question 14 Écrire la méthode `private static void derouleReunionsPasseesConfidentielles(List<Reunion> reunions)` de la classe `Entreprise`, qui ajoute aux initiés les informations confidentielles dont ils ont déjà connaissance, c'est-à-dire dont les réunions sont passées.

Question 15 Écrire la méthode `public boolean estObsolete()` de la classe `Information`, qui renvoie vrai si et seulement si l'information est obsolète, c'est-à-dire si 3 ans ou plus se sont écoulés depuis sa divulgation au public.

On souhaite maintenant supprimer les informations obsolètes de la liste des informations privilégiées connues par les initiés : nous allons pour cela maintenir au sein de chaque information la liste des initiés qui en ont connaissance.

Question 16 On veut ajouter dans la classe `Information` un attribut `connaisseurs` représentant la liste des initiés ayant connaissance ayant connaissance de l'information. Écrire la ligne de code déclarant cet attribut.
Faut-il ajouter quelque chose pour son initialisation ?

Question 17 Écrire la méthode `public void devientConnuPar(Initie initie)` de la classe `Information`, qui ajoute l'initié passé en paramètre à la liste de ceux connaissant l'information.

Question 18 Ré-écricre la méthode `public void prendConnaissance(Information sujet)` de la classe `Initie`, pour que la liste des connaisseurs du sujet soit également mise à jour lorsqu'un initié prend connaissance d'une information.

Question 19 Écrire la méthode `public void supprimeSujet(Information sujet)` de la classe `Initie`, qui supprime l'information passée en paramètre de la liste des sujets connus par l'initié. Cette méthode ne sera appelée que pour des sujets obsolètes, il n'est donc pas nécessaire de vérifier l'obsolescence ici.

Question 20 Écrire la méthode `public void miseAJour()` de la classe `Information`, qui teste si une information est obsolète, et supprime le cas échéant l'information de la liste des sujets connus des initiés concernés.

Question 21 Nous voulons maintenant introduire un nouveau type de réunion : les réunions de suivi. Ces réunions sont des cas particuliers de réunion, qui ont pour particularité d'être liées à une réunion précédente sur le même sujet. Tous les participants à une réunion le seront également aux réunions de suivi, d'autres peuvent être ajoutés.

Ecrivez pour cela une classe ReunionDeSuivi héritant de la classe réunion, ayant pour attribut supplémentaire `reunionPrecedente`. Cette classe comprendra un constructeur prenant une réunion (la réunion dont elle fait suite) et une date (de type `LocalDate`) en paramètres. Le sujet de la réunion est le même que celui de la réunion précédente, la liste des présents contient initialement les mêmes participants que la réunion précédente.

Avez-vous besoin de définir autre chose dans cette classe pour pouvoir l'utiliser de manière cohérente avec le reste du programme ? Si oui, précisez votre réponse.

Question Bonus Voyez-vous des soucis avec cette implémentation ? Quels tests supplémentaires voudriez-vous faire ? Comment amélioreriez-vous ce programme ?

3 Bowling (4 points)

Nous allons ici coder un petit programme permettant de gérer une partie de bowling pour un joueur.

Les règles du bowling sont les suivantes (source : `http://www.bowlingcity.fr/les_regles_du_jeu,12,1.html`)

> Le bowling est un jeu simple. Il consiste à renverser les 10 quilles en faisant rouler la boule sur une piste de 20 m de long.
> Chaque partie comprend 10 jeux ou frames. Les participants jouent chacun leur tour. Le joueur lance 2 boules à chaque jeu et marque le nombre de points correspondant au nombre de quilles tombées.
> Si les 10 quilles tombent en deux lancers, le joueur a fait un SPARE(/) ; il marque 10 points plus les points du lancer suivant.
> Si les 10 quilles tombent au premier lancer, le joueur a fait un STRIKE (X) ; Il marque 10 points plus les points des 2 lancers suivants.
> Au dixième jeu, si le joueur réalise un Spare ou un Strike, il bénéficie d'1 ou 2 lancer(s) supplementaire(s).
> Compte tenu de ces bonifications, le maximum de points par partie est de 300, mais un joueur débutant s'estime satisfait avec 100 points.
> Une boule pèse entre 3 à 7 kg, et l'on effectue de 13 à 21 lancers.
> Bon score = 130 pour un joueur débutant, 180 pour un joueur confirmé.

Pour simplifier cette première version d'un programme gérant une partie de bowling, nous allons systématiquement déclarer 12 frames, chacune de 2 lancers. Le score sera

calculé sur les 10 premières frames. Le nombre de quilles tombées lors des deux suivantes n'est consulté que pour calculer le score des neuvième et dixième frame en cas de strike ou de spare. Autrement dit, on accorde toujours deux frames bonus, quitte à donner des lancers inutiles (non pris en compte dans le calcul du score) au joueur.

Le code est organisé en deux classes : `Frame` et `OnePlayerGame`. Les squelettes de ces deux classes vous sont donnés en annexe. Des méthodes vous sont fournies, vous pouvez les utiliser. Il n'est pas nécessaire de modifier les méthodes fournies. Il vous est ici demandé de compléter ces classes pour permettre de gérer une partie solo. Attention, certaines réponses ne sont pas immédiates. Vous pouvez déclarer des méthodes auxiliaires. N'hésitez pas à expliquer votre démarche ou à donner des réponses partielles *en précisant les éléments qu'il resterait à traiter.*

Question 1 Écrire la méthode `public int score()` de la classe `OnePlayerGame`, qui renvoie le score de la partie, c'est à dire la somme du score des 10 premières frames.

Question 2 Écrire la méthode `public boolean isStrike()` de la classe `Frame`, qui renvoie vrai si la frame contient un strike. Attention, les frames 11 et 12 ne doivent jamais être considérés comme des strikes.

Question 3 Écrire la méthode `public boolean isSpare()` de la classe `Frame`, qui renvoie vrai si la frame contient un spare. Attention, les frames 11 et 12 ne doivent jamais être considérés comme des spares.

Question 4 Écrire la méthode `public int score()` de la classe `Frame`, qui renvoie le score de la frame, selon les règles énoncées ci-dessus.

4 Annexe

4.1 Exercice 2, fichier Entreprise.java

```
1  package exoInit;
2
3  import java.time.LocalDate;
4  import java.util.ArrayList;
5  import java.util.List;
6
7  public class Entreprise {
8
9      private static void derouleReunionsPasseesConfidentielles(
10             List<Reunion> reunions) {
11         //TODO Question 14
12     }
13
14     public static void main(String[] args) {
```

```
15
16          List<Initie> inities = new ArrayList<Initie>();
17          List<Information> informations = new ArrayList<Information>();
18          List<Reunion> reunions = new ArrayList<Reunion>();
19
20          // initialisation
21          // creation des inities
22          Initie pdg = new Initie("A. Boss", "PDG");
23          inities.add(pdg);
24          Initie daf = new Initie("B. Money",
25                  "Directeur administratif financier");
26          inities.add(daf);
27          Initie adir = new Initie("C. Notes", "Assistant de Direction");
28          inities.add(adir);
29          Initie act1 = new Initie("D. Sous", "Actionnaire");
30          inities.add(act1);
31          Initie act2 = new Initie("E. Bourse", "Actionnaire");
32          inities.add(act2);
33          Initie jur = new Initie("F. Legal", "Directeur Juridique");
34          inities.add(jur);
35          Initie com = new Initie("G. Public", "Directeur commercial");
36          inities.add(com);
37
38          // creation de sujets
39          Information bilFin = new Information("Bilan financier 2019", 10, 01,
40                  2020);
41          informations.add(bilFin);
42          Information achat = new Information(
43                  "Proposition d'achat d'une entreprise", 20, 05, 2020);
44          informations.add(achat);
45          Information courr = new Information("Affaires courrantes Janvier 2020",
46                  05, 02, 2020);
47          informations.add(courr);
48
49          // reunion passee
50          Reunion reu1 = new Reunion(LocalDate.of(2020, 01, 02), bilFin);
51          reu1.ajouteInitie(pdg);
52          reu1.ajouteInitie(adir);
53          reu1.ajouteInitie(daf);
54          reu1.ajouteInitie(act1);
55          reu1.ajouteInitie(act2);
56          reunions.add(reu1);
57
58          // reunion future
59          Reunion reu2 = new Reunion(LocalDate.of(2020, 01, 30), achat);
60          reu2.ajouteInitie(pdg);
61          reu2.ajouteInitie(jur);
62          reu2.ajouteInitie(com);
63          reunions.add(reu2);
```

```
64
65          // reunion aujourdhui
66          Reunion reu3 = new Reunion(courr);
67          reu3.ajouteInitie(pdg);
68          reu3.ajouteInitie(adir);
69          reu3.ajouteInitie(daf);
70          reunions.add(reu3);
71
72          derouleReunionsPasseesConfidentielles(reunions);
73
74          for (Initie initie : inities) {
75             System.out.println(initie);
76          }
77
78      }
79
80  }
```

4.2 Exercice 2, affichage à l'execution du programme

```
A. Boss, PDG connait les sujets :
- Bilan financier 2019
- Affaires courrantes Janvier 2020

B. Money, Directeur administratif financier connait les sujets :
- Bilan financier 2019
- Affaires courrantes Janvier 2020

C. Notes, Assistant de Direction connait les sujets :
- Bilan financier 2019
- Affaires courrantes Janvier 2020

D. Sous, Actionnaire connait les sujets :
- Bilan financier 2019

E. Bourse, Actionnaire connait les sujets :
- Bilan financier 2019

F. Legal, Directeur Juridique connait les sujets :

G. Public, Directeur commercial connait les sujets :
```

4.3 Exercice 2, squelette de la classe Initie

```java
package exoInit;

import java.util.ArrayList;
import java.util.List;

public class Initie {
    private String nom;
    private String role;
    private List<Information> sujetsConnus;

    public Initie(String nom, String role) {
        //TODO Question 1
    }

    public String getNom() {
        return nom;
    }

    public List<Information> getSujetsConnus() {
        return sujetsConnus;
    }

    public void prendConnaissance(Information sujet) {
        //TODO Question 4
    }

    @Override
    public String toString() {
        //TODO Question 5
    }
```

4.4 Exercice 2, squelette de la classe Information

```java
package exoInit;

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

public class Information {
    private String titre;
    private LocalDate dateDivulgation;

    public Information(String titre, LocalDate dateDivulgation) {
        //TODO Question 2
    }

```

```
15      public Information(String titre, int jour, int mois, int annee) {
16          //TODO Question 3
17      }
18
19      public String getTitre() {
20          return titre;
21      }
22
23      public LocalDate getDateDivulgation() {
24          return dateDivulgation;
25      }
26
27      public boolean estObsolete() {
28          //TODO Question 15
29      }
30
31  }
```

4.5 Exercice 3, squelette de la classe OnePlayerGame

```
1
2   package bowling;
3
4   import java.util.Scanner;
5
6   public class OnePlayerGame {
7       private Frame[] frames = new Frame[12];
8
9       public OnePlayerGame(){
10          frames[11]= new Frame(12, null);
11          for (int i = 10; i>=0; i --){
12              frames[i]= new Frame(i+1, frames[i+1]);
13          }
14      }
15
16      public void play(){
17          //initialize a scanner for keyboard input
18          Scanner sc = new Scanner(System.in);
19          for(Frame f : this.frames){
20              f.playFrame(sc);
21          }
22
23      }
24
25      public int score(){
26          //TODO Question 1
27      }
```

```
28
29      public void display(){
30          for (Frame f : frames){
31              System.out.println(f);
32          }
33      }
34
35      public static void main(String[] args) {
36          OnePlayerGame myGame = new OnePlayerGame();
37          myGame.play();
38          myGame.display();
39          System.out.println("score : "+ myGame.score());
40      }
41  }
```

4.6 Exercice 3, squelette de la classe Frame

```
1   package bowling;
2
3   import java.util.Scanner;
4
5   public class Frame {
6       private int num;
7       private int nbPinsFirstThrow;
8       private int nbPinsSecondThrow;
9       private Frame nextFrame;
10
11      public Frame(int num, Frame nextFrame) {
12          this.num = num;
13          this.nextFrame = nextFrame;
14      }
15
16      public boolean isStrike() {
17          //TODO Question 2
18      }
19
20      public boolean isSpare() {
21          //TODO Question 3
22      }
23
24      public int score() {
25          //TODO Question 4
26      }
27
28      @Override
29      public String toString() {
30          return "Frame " + num + ": " + (isStrike() ? "XXX"
```

```
31              : isSpare() ? "/" : nbPinsFirstThrow + " " + nbPinsSecondThrow);
32      }
33
34      public void playFrame(Scanner sc) {
35          System.out.println("Frame numero " + this.num + ", premier lancer : ");
36          this.nbPinsFirstThrow = sc.nextInt();
37          if (nbPinsFirstThrow < 0 || nbPinsFirstThrow > 10) {
38              throw new RuntimeException("Nombre de quilles invalide");
39          }
40
41          if (this.nbPinsFirstThrow == 10) {
42              return;
43          }
44
45          System.out.println("Frame numero " + this.num + ", deuxieme lancer : ");
46          this.nbPinsSecondThrow = sc.nextInt();
47          if (nbPinsSecondThrow < 0
48                  || nbPinsSecondThrow > 10 - nbPinsFirstThrow) {
49              throw new RuntimeException("Nombre de quilles invalide");
50          }
51      }
52  }
```

java.util

Interface List<E>

Type Parameters:

`E - the type of elements in this list`

All Superinterfaces:

`Collection`<E>, `Iterable`<E>

All Known Implementing Classes:

`AbstractList`, `AbstractSequentialList`, `ArrayList`, `AttributeList`, `CopyOnWriteArrayList`, `LinkedList`, `RoleList`, `RoleUnresolvedList`, `Stack`, `Vector`

```
public interface List<E>
extends Collection<E>
```

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements `e1` and `e2` such that `e1.equals(e2)`, and they typically allow multiple null elements if they allow null elements at all. It is not inconceivable that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions when the user attempts to insert them, but we expect this usage to be rare.

The `List` interface places additional stipulations, beyond those specified in the `Collection` interface, on the contracts of the `iterator`, `add`, `remove`, `equals`, and `hashCode` methods. Declarations for other inherited methods are also included here for convenience.

The `List` interface provides four methods for positional (indexed) access to list elements. Lists (like Java arrays) are zero based. Note that these operations may execute in time proportional to the index value for some implementations (the `LinkedList` class, for example). Thus, iterating over the elements in a list is typically preferable to indexing through it if the caller does not know the implementation.

The `List` interface provides a special iterator, called a `ListIterator`, that allows element insertion and replacement, and bidirectional access in addition to the normal operations that the `Iterator` interface provides. A method is provided to obtain a list iterator that starts at a specified position in the list.

The `List` interface provides two methods to search for a specified object. From a performance standpoint, these methods should be used with caution. In many implementations they will perform costly linear searches.

The `List` interface provides two methods to efficiently insert and remove multiple elements at an arbitrary point in the list.

Note: While it is permissible for lists to contain themselves as elements, extreme caution is advised: the `equals` and `hashCode` methods are no longer well defined on such a list.

Some list implementations have restrictions on the elements that they may contain. For example, some implementations prohibit null elements, and some have restrictions on the types of their elements. Attempting to add an ineligible element throws an unchecked

exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible element may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible element whose completion would not result in the insertion of an ineligible element into the list may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

This interface is a member of the Java Collections Framework.

Since:

1.2

See Also:

`Collection`, `Set`, `ArrayList`, `LinkedList`, `Vector`, `Arrays.asList(Object[])`, `Collections.nCopies(int, Object)`, `Collections.EMPTY_LIST`, `AbstractList`, `AbstractSequentialList`

Method Summary

All Methods Instance Methods Abstract Methods Default Methods

Modifier and Type	Method and Description
boolean	**add**(**E** e) Appends the specified element to the end of this list (optional operation).
void	**add**(int index, **E** element) Inserts the specified element at the specified position in this list (optional operation).
boolean	**addAll**(**Collection**<? extends **E**> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
boolean	**addAll**(int index, **Collection**<? extends **E**> c) Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
void	**clear**() Removes all of the elements from this list (optional operation).
boolean	**contains**(**Object** o) Returns `true` if this list contains the specified element.
boolean	**containsAll**(**Collection**<?> c) Returns `true` if this list contains all of the elements of the specified collection.
boolean	**equals**(**Object** o) Compares the specified object with this list for equality.
E	**get**(int index)

	Returns the element at the specified position in this list.
int	**hashCode**() Returns the hash code value for this list.
int	**indexOf**(**Object** o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**() Returns `true` if this list contains no elements.
Iterator<**E**>	**iterator**() Returns an iterator over the elements in this list in proper sequence.
int	**lastIndexOf**(**Object** o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<**E**>	**listIterator**() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<**E**>	**listIterator**(int index) Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
E	**remove**(int index) Removes the element at the specified position in this list (optional operation).
boolean	**remove**(**Object** o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).
boolean	**removeAll**(**Collection**<?> c) Removes from this list all of its elements that are contained in the specified collection (optional operation).
default void	**replaceAll**(**UnaryOperator**<**E**> operator) Replaces each element of this list with the result of applying the operator to that element.
boolean	**retainAll**(**Collection**<?> c) Retains only the elements in this list that are contained in the specified collection (optional operation).
E	**set**(int index, **E** element) Replaces the element at the specified position in this list with the specified element (optional operation).
int	**size**() Returns the number of elements in this list.

default void	**sort**(**Comparator**<? super **E**> c) Sorts this list according to the order induced by the specified **Comparator**.
default **Spliterator**<**E**>	**spliterator**() Creates a **Spliterator** over the elements in this list.
List<**E**>	**subList**(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive.
Object[]	**toArray**() Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

Methods inherited from interface java.util.Collection

parallelStream, removeIf, stream

Methods inherited from interface java.lang.Iterable

forEach

java.time

Class LocalDate

java.lang.Object
    java.time.LocalDate

All Implemented Interfaces:

Serializable, Comparable<ChronoLocalDate>, ChronoLocalDate, Temporal, TemporalAccessor, TemporalAdjuster

---

```
public final class LocalDate
extends Object
implements Temporal, TemporalAdjuster, ChronoLocalDate, Serializable
```

A date without a time-zone in the ISO-8601 calendar system, such as `2007-12-03`.

`LocalDate` is an immutable date-time object that represents a date, often viewed as year-month-day. Other date fields, such as day-of-year, day-of-week and week-of-year, can also be accessed. For example, the value "2nd October 2007" can be stored in a `LocalDate`.

This class does not store or represent a time or time-zone. Instead, it is a description of the date, as used for birthdays. It cannot represent an instant on the time-line without additional information such as an offset or time-zone.

The ISO-8601 calendar system is the modern civil calendar system used today in most of the world. It is equivalent to the proleptic Gregorian calendar system, in which today's rules for leap years are applied for all time. For most applications written today, the ISO-8601 rules are entirely suitable. However, any application that makes use of historical dates, and requires them to be accurate will find the ISO-8601 approach unsuitable.

This is a value-based class; use of identity-sensitive operations (including reference equality (==), identity hash code, or synchronization) on instances of `LocalDate` may have unpredictable results and should be avoided. The `equals` method should be used for comparisons.

Implementation Requirements:

`This class is immutable and thread-safe.`

Since:

`1.8`

See Also:

Serialized Form

Field Summary

Fields

Modifier and Type	Field and Description
`static LocalDate`	**MAX** The maximum supported `LocalDate`, '+999999999-12-31'.
`static LocalDate`	**MIN**

	The minimum supported `LocalDate`, '-999999999-01-01'.

Method Summary

All Methods | Static Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
`Temporal`	**`adjustInto`**`(Temporal temporal)` Adjusts the specified temporal object to have the same date as this object.
`LocalDateTime`	**`atStartOfDay`**`()` Combines this date with the time of midnight to create a `LocalDateTime` at the start of this date.
`ZonedDateTime`	**`atStartOfDay`**`(ZoneId zone)` Returns a zoned date-time from this date at the earliest valid time according to the rules in the time-zone.
`LocalDateTime`	**`atTime`**`(int hour, int minute)` Combines this date with a time to create a `LocalDateTime`.
`LocalDateTime`	**`atTime`**`(int hour, int minute, int second)` Combines this date with a time to create a `LocalDateTime`.
`LocalDateTime`	**`atTime`**`(int hour, int minute, int second, int nanoOfSecond)` Combines this date with a time to create a `LocalDateTime`.
`LocalDateTime`	**`atTime`**`(LocalTime time)` Combines this date with a time to create a `LocalDateTime`.
`OffsetDateTime`	**`atTime`**`(OffsetTime time)` Combines this date with an offset time to create an `OffsetDateTime`.
`int`	**`compareTo`**`(ChronoLocalDate other)` Compares this date to another date.
`boolean`	**`equals`**`(Object obj)` Checks if this date is equal to another date.
`String`	**`format`**`(DateTimeFormatter formatter)` Formats this date using the specified formatter.
`static LocalDate`	**`from`**`(TemporalAccessor temporal)` Obtains an instance of `LocalDate` from a temporal object.
`int`	**`get`**`(TemporalField field)` Gets the value of the specified field from this date as an `int`.
`IsoChronology`	**`getChronology`**`()` Gets the chronology of this date, which is the ISO calendar system.

int	**getDayOfMonth**() Gets the day-of-month field.
DayOfWeek	**getDayOfWeek**() Gets the day-of-week field, which is an enum `DayOfWeek`.
int	**getDayOfYear**() Gets the day-of-year field.
Era	**getEra**() Gets the era applicable at this date.
long	**getLong**(**TemporalField** field) Gets the value of the specified field from this date as a `long`.
Month	**getMonth**() Gets the month-of-year field using the `Month` enum.
int	**getMonthValue**() Gets the month-of-year field from 1 to 12.
int	**getYear**() Gets the year field.
int	**hashCode**() A hash code for this date.
boolean	**isAfter**(**ChronoLocalDate** other) Checks if this date is after the specified date.
boolean	**isBefore**(**ChronoLocalDate** other) Checks if this date is before the specified date.
boolean	**isEqual**(**ChronoLocalDate** other) Checks if this date is equal to the specified date.
boolean	**isLeapYear**() Checks if the year is a leap year, according to the ISO proleptic calendar system rules.
boolean	**isSupported**(**TemporalField** field) Checks if the specified field is supported.
boolean	**isSupported**(**TemporalUnit** unit) Checks if the specified unit is supported.
int	**lengthOfMonth**() Returns the length of the month represented by this date.
int	**lengthOfYear**() Returns the length of the year represented by this date.
LocalDate	**minus**(long amountToSubtract, **TemporalUnit** unit) Returns a copy of this date with the specified amount subtracted.
LocalDate	**minus**(**TemporalAmount** amountToSubtract)

	Returns a copy of this date with the specified amount subtracted.
LocalDate	**minusDays**(long daysToSubtract) Returns a copy of this LocalDate with the specified number of days subtracted.
LocalDate	**minusMonths**(long monthsToSubtract) Returns a copy of this LocalDate with the specified number of months subtracted.
LocalDate	**minusWeeks**(long weeksToSubtract) Returns a copy of this LocalDate with the specified number of weeks subtracted.
LocalDate	**minusYears**(long yearsToSubtract) Returns a copy of this LocalDate with the specified number of years subtracted.
static **LocalDate**	**now**() Obtains the current date from the system clock in the default time-zone.
static **LocalDate**	**now**(**Clock** clock) Obtains the current date from the specified clock.
static **LocalDate**	**now**(**ZoneId** zone) Obtains the current date from the system clock in the specified time-zone.
static **LocalDate**	**of**(int year, int month, int dayOfMonth) Obtains an instance of LocalDate from a year, month and day.
static **LocalDate**	**of**(int year, **Month** month, int dayOfMonth) Obtains an instance of LocalDate from a year, month and day.
static **LocalDate**	**ofEpochDay**(long epochDay) Obtains an instance of LocalDate from the epoch day count.
static **LocalDate**	**ofYearDay**(int year, int dayOfYear) Obtains an instance of LocalDate from a year and day-of-year.
static **LocalDate**	**parse**(**CharSequence** text) Obtains an instance of LocalDate from a text string such as 2007-12-03.
static **LocalDate**	**parse**(**CharSequence** text, **DateTimeFormatter** formatter) Obtains an instance of LocalDate from a text string using a specific formatter.
LocalDate	**plus**(long amountToAdd, **TemporalUnit** unit) Returns a copy of this date with the specified amount added.
LocalDate	**plus**(**TemporalAmount** amountToAdd) Returns a copy of this date with the specified amount added.
LocalDate	**plusDays**(long daysToAdd)

	Returns a copy of this LocalDate with the specified number of days added.
LocalDate	**plusMonths**(long monthsToAdd) Returns a copy of this LocalDate with the specified number of months added.
LocalDate	**plusWeeks**(long weeksToAdd) Returns a copy of this LocalDate with the specified number of weeks added.
LocalDate	**plusYears**(long yearsToAdd) Returns a copy of this LocalDate with the specified number of years added.
<R> R	**query**(TemporalQuery<R> query) Queries this date using the specified query.
ValueRange	**range**(TemporalField field) Gets the range of valid values for the specified field.
long	**toEpochDay**() Converts this date to the Epoch Day.
String	**toString**() Outputs this date as a String, such as 2007-12-03.
Period	**until**(ChronoLocalDate endDateExclusive) Calculates the period between this date and another date as a Period.
long	**until**(Temporal endExclusive, TemporalUnit unit) Calculates the amount of time until another date in terms of the specified unit.
LocalDate	**with**(TemporalAdjuster adjuster) Returns an adjusted copy of this date.
LocalDate	**with**(TemporalField field, long newValue) Returns a copy of this date with the specified field set to a new value.
LocalDate	**withDayOfMonth**(int dayOfMonth) Returns a copy of this LocalDate with the day-of-month altered.
LocalDate	**withDayOfYear**(int dayOfYear) Returns a copy of this LocalDate with the day-of-year altered.
LocalDate	**withMonth**(int month) Returns a copy of this LocalDate with the month-of-year altered.
LocalDate	**withYear**(int year) Returns a copy of this LocalDate with the year altered.

Methods inherited from class java.lang.Object

clone, finalize, getClass, notify, notifyAll, wait, wait, wait