Java8

- Interface par défaut
- Stream Exercice
- Stream devoxx 2014
- Producteur Consommateur
- Stream et JPA
- ParalleleStream
- Stream
- Java 8 Date
- Exercice Date

Interface par défaut

Cas d'usage, le comportement par défaut

Le premier cas d'usage est la mise en place d'un comportement par "défaut" pour les objets

```
public interface MyInterface {


    // regular interface methods


    default void defaultMethod() {
        // default method implementation
    }
}
```

La raison pour laquelle la version Java 8 incluait des méthodes par défaut est assez évidente.

Dans une conception typique basée sur des abstractions, où une interface a une ou plusieurs implémentations, si une ou plusieurs méthodes sont ajoutées à l'interface, toutes les implémentations seront obligées de les implémenter également. Sinon, la conception s'effondrera.

Les méthodes d'interface par défaut sont un moyen efficace de traiter ce problème. Ils nous permettent d'ajouter de nouvelles méthodes à une interface qui sont automatiquement disponibles dans les implémentations. Par conséquent, nous n'avons pas besoin de modifier les classes d'implémentation.

De cette façon, la rétrocompatibilité est soigneusement préservée sans avoir à refactoriser les implémentations.

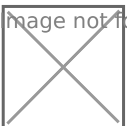
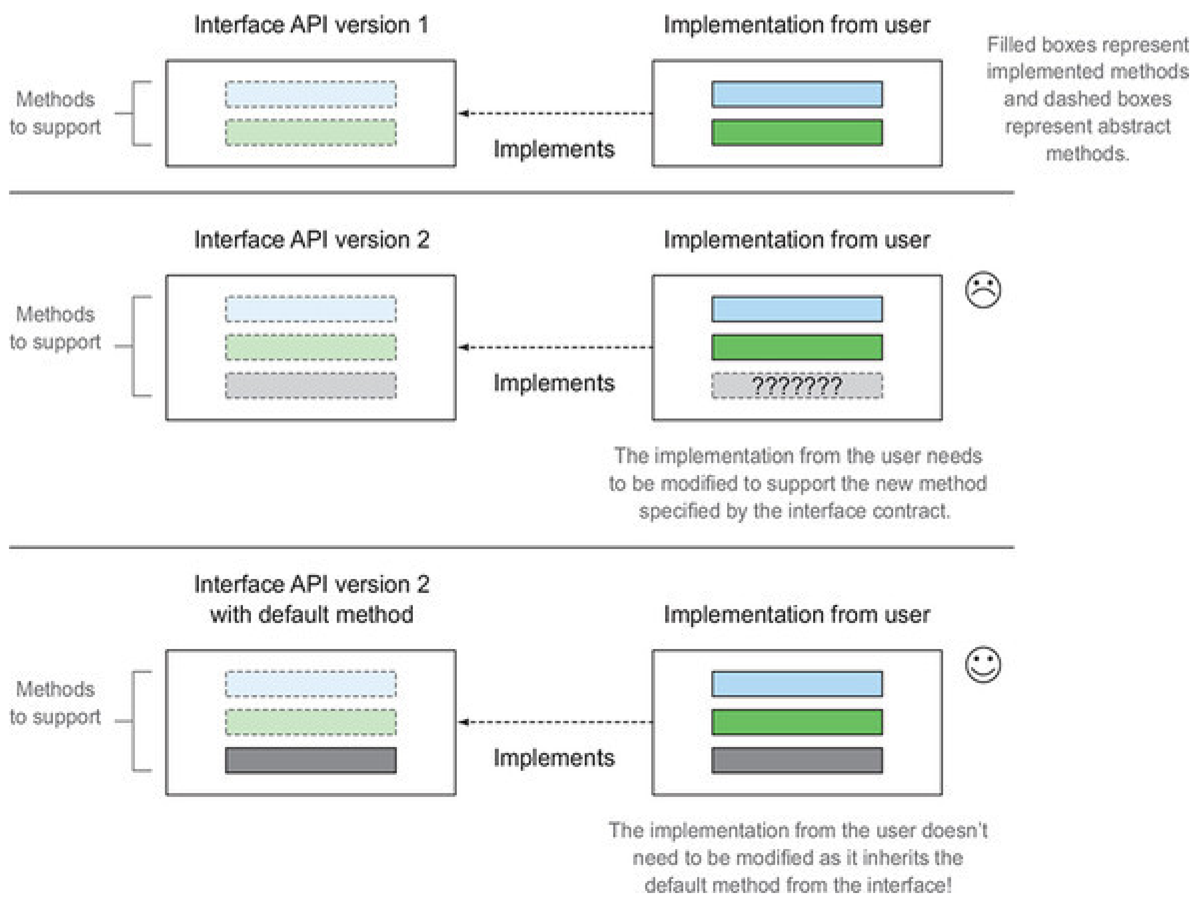


Image not found or type unknown



La problématique du diamand est soulevé ainsi:

```
public interface Vehicle {

    String getBrand();

    String speedUp();

    String slowDown();

    default String turnAlarmOn() {
        return "Turning the vehicle alarm on.";
    }

    default String turnAlarmOff() {
        return "Turning the vehicle alarm off.";
    }
}
public interface Alarm {

    default String turnAlarmOn() {
```

```
        return "Turning the alarm on.";
    }

    default String turnAlarmOff() {
        return "Turning the alarm off.";
    }
}
```

Il faut si on veux hériter des deux interface explicitement implémenté les méthodes quitte a rappeler les methodes supérierus

```
@Override
public String turnAlarmOn() {
    return Vehicle.super.turnAlarmOn();
}

@Override
public String turnAlarmOff() {
    return Vehicle.super.turnAlarmOff();
}
```

Methode statique

En plus de déclarer des méthodes par défaut dans les interfaces, Java 8 nous permet également de définir et d'implémenter des méthodes statiques dans les interfaces.

Puisque les méthodes statiques n'appartiennent pas à un objet particulier, elles ne font pas partie de l'API des classes implémentant l'interface ; par conséquent, ils doivent être appelés en utilisant le nom de l'interface précédant le nom de la méthode.

Pour comprendre le fonctionnement des méthodes statiques dans les interfaces, refactorisons l'interface Vehicle et ajoutons-y une méthode utilitaire statique :

```
public interface Vehicle {

    // regular / default interface methods
```

```
    static int getHorsePower(int rpm, int torque) {
        return (rpm * torque) / 5252;
    }
}
```

Stream Exercice

Contexte

Soit le model de donnée suivant:

```
@Data
@Entity
@NoArgsConstructor
public class Customer {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String name;
  private Integer tier;
}

@Data
@NoArgsConstructor
@Entity
@Table(name = "product_order")
public class Order {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private LocalDate orderDate;
  private LocalDate deliveryDate;
  private String status;

  @ManyToOne
  @JoinColumn(name = "customer_id")
  private Customer customer;

  @ManyToMany
  @JoinTable(
      name = "order_product_relationship",
```

```
      joinColumns = { @JoinColumn(name = "order_id") },
      inverseJoinColumns = { @JoinColumn(name = "product_id") }
  )
  @ToString.Exclude
  Set<Product> products;

}



@Data
@NoArgsConstructor
@Entity
public class Product {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String name;
  private String category;
  @With private Double price;

  @ManyToMany(mappedBy = "products")
  @ToString.Exclude
  private Set<Order> orders;
}
```

Opération Non terminal

- filter()
- map()
- flatMap
- distinct()
- sorted()
- peek()

Operation terminal

- anyMatch()
- collect()
- count()
- findFirst()
- min()

- max()
- sum()
- average()

Exercice 1 — Obtenir une liste de produits appartenant à la catégorie "Livres" avec un prix > 100

L'idée est d'utiliser un double filter et une collection.

Solution

```
List<Product> result = productRepo.findAll()
  .stream()
  .filter(p -> p.getCategory().equalsIgnoreCase("Livres"))
  .filter(p -> p.getPrice() > 100)
  .collect(Collectors.toList());
```

Exercice 2 — Obtenir une liste de commandes avec des produits appartenant à la catégorie "Bébé"

Vous devez partir du flux de données des entités de la commande, puis vérifier si les produits de la commande appartiennent à la catégorie "Bébé". Par conséquent, la logique de filtrage examine le flux de produits de chaque enregistrement de commande et utilise anyMatch() pour déterminer si un produit remplit les critères.

Solution

```
  List<Order> result = orderRepo.findAll()
        .stream()
        .filter(o ->
          o.getProducts()
          .stream()
```

```
            .anyMatch(p -> p.getCategory().equalsIgnoreCase("Bébé"))
          )
          .collect(Collectors.toList());
```

Exercice 3 - Obtenez une liste de produits avec la catégorie = "Jouets" puis appliquez une remise de 10 %

Dans cet exercice, vous verrez comment transformer des données à l'aide de l'API de flux. Après avoir obtenu une liste de produits avec une catégorie appartenant à "Jouets" à l'aide de filter(), vous pouvez ensuite appliquer une remise de 10 % sur le prix du produit en utilisant map().

Solution

```
    List<Product> result = productRepo.findAll()
          .stream()
          .filter(p -> p.getCategory().equalsIgnoreCase("Jouets"))
          .map(p -> p.withPrice(p.getPrice() * 0.9))
          .collect(Collectors.toList());
```

Exercice 4 — Obtenir une liste des produits commandés par le client du niveau 2 entre le 1er février 2021 et le 1er avril 2021

Cet exercice illustre l'utilisation de flatMap(). Vous pouvez d'abord partir d'une liste de commandes, puis filtrer la liste par niveau de client et date de commande. Ensuite, récupérez la liste des produits de chaque enregistrement de commande et utilisez flatMap() pour émettre des enregistrements de produits dans le flux. Par exemple, si nous avons 3 enregistrements de commande et que chaque commande contient 10 produits, alors flatMap() émettra 10 éléments de données pour chaque enregistrement de commande, ce qui entraînera une sortie d'enregistrement de produit de 30 (3 x 10) dans le flux.

Étant donné que la liste de produits contiendrait des enregistrements de produits en double si plusieurs commandes incluaient le même produit. Afin de générer une liste de produits unique, l'application de l'opération distinct() peut aider à produire la liste unique.

Solution

```
List<Product> result = orderRepo.findAll()
 .stream()
 .filter(o -> o.getCustomer().getTier() == 2)
 .filter(o -> o.getOrderDate().compareTo(LocalDate.of(2021, 2, 1)) >= 0)
 .filter(o -> o.getOrderDate().compareTo(LocalDate.of(2021, 4, 1)) <= 0)
 .flatMap(o -> o.getProducts().stream())
 .distinct()
 .collect(Collectors.toList());
```

Exercice 5 — Obtenir les produits les moins chers de la catégorie « Livres »

L'un des moyens efficaces d'obtenir le produit au prix le plus bas est de trier la liste de produits par prix dans un ordre croissant et d'obtenir le premier élément. L'API Java Stream fournit une opération **sorted ()** pour le tri des données de flux en fonction d'attributs de champ spécifiques. Afin d'obtenir le premier élément du flux, vous pouvez utiliser l'opération de terminal **findFirst().** L'opération renvoie le premier élément de données enveloppé par Facultatif car il est possible que le flux de sortie soit vide.

Cette solution ne peut renvoyer qu'un seul enregistrement de produit avec le prix le plus bas. S'il existe plusieurs enregistrements de produit avec le même prix le plus bas, la solution doit être modifiée de sorte qu'elle recherche d'abord le montant du prix le plus bas, puis filtre les enregistrements de produit par le montant du prix afin d'obtenir une liste de produits avec le même prix le plus bas.

Solution

```
Optional<Product> result = productRepo.findAll()
        .stream()
        .filter(p -> p.getCategory().equalsIgnoreCase("Livres"))
        .sorted(Comparator.comparing(Product::getPrice))
        .findFirst();
```

ou

```
Optional<Product> result = productRepo.findAll()
        .stream()
        .filter(p -> p.getCategory().equalsIgnoreCase("Livres"))
        .min(Comparator.comparing(Product::getPrice));
```

Exercice 6 — Obtenir les 3 dernières commandes passées

Semblable à l'exercice précédent, la solution évidente consiste à trier les enregistrements de commande par champ de date de commande. La partie délicate est que le tri doit cette fois être dans l'ordre décroissant afin que vous puissiez obtenir les enregistrements de commande avec la date de commande la plus récente. Cela peut être réalisé simplement en appelant **Comparator.reversed().**

Solution

```
List<Order> result = orderRepo.findAll()
        .stream()
        .sorted(Comparator.comparing(Order::getOrderDate).reversed())
        .limit(3)
        .collect(Collectors.toList());
```

Exercice 7 - Obtenez une liste des commandes qui ont été commandées le 15 mars 2021, enregistrez les enregistrements de commande sur la console, puis renvoyez sa liste de produits

Vous pouvez voir que cet exercice implique deux actions - (1) écrire des enregistrements de commande sur la console et (2) produire une liste de produits. Générer une sortie différente à partir d'un flux n'est pas possible, comment pouvons-nous répondre à cette exigence ? En plus d'exécuter le flux de flux deux fois, l'opération **peek**() permet l'exécution de la logique système dans le cadre d'un flux de flux. L'exemple de solution exécute peek() pour écrire les

enregistrements de commande dans la console juste après le filtrage des données, puis les opérations suivantes telles que **flatMap**() seront exécutées pour la sortie des enregistrements de produit.

Solution

```
List<Product> result = orderRepo.findAll()
    .stream()
    .filter(o -> o.getOrderDate().isEqual(LocalDate.of(2021, 3, 15)))
    .peek(o -> System.out.println(o.toString()))
    .flatMap(o -> o.getProducts().stream())
    .distinct()
    .collect(Collectors.toList());
```

Exercice 8 — Calculer la somme forfaitaire totale de toutes les commandes passées en février 2021

Tous les exercices précédents consistaient à générer une liste d'enregistrements par une opération de terminal, faisons quelques calculs cette fois. Cet exercice consiste à résumer tous les produits commandés en février 2021. Comme vous avez parcouru les exercices précédents, vous pouvez facilement obtenir la liste des produits à l'aide des opérations filter() et flatMap(). Ensuite, vous pouvez utiliser l'opération mapToDouble() pour convertir le flux en un flux de type de données Double en spécifiant le champ de prix comme valeur de mappage. Enfin, l'opération de terminal sum() vous aidera à additionner toutes les valeurs et à renvoyer la valeur totale.

Solution

```
Double result = orderRepo.findAll()
    .stream()
    .filter(o -> o.getOrderDate().compareTo(LocalDate.of(2021, 2, 1)) >= 0)
    .filter(o -> o.getOrderDate().compareTo(LocalDate.of(2021, 3, 1)) < 0)
    .flatMap(o -> o.getProducts().stream())
    .mapToDouble(p -> p.getPrice())
    .sum();
```

Exercice 9 — Calculer le paiement moyen d'une commande passée le 14 mars 2021

En plus de la somme totale, l'API de flux offre également une opération pour le calcul de la valeur moyenne. Vous constaterez peut-être que le type de données de retour est différent de **sum**() car il s'agit d'un type de données facultatif. La raison en est que le flux de données serait vide et que le calcul ne produira donc pas de valeur moyenne pour un flux de données vide.

Solution

```
    Double result = orderRepo.findAll()
        .stream()
        .filter(o -> o.getOrderDate().isEqual(LocalDate.of(2021, 3, 15)))
        .flatMap(o -> o.getProducts().stream())
        .mapToDouble(p -> p.getPrice())
        .average().getAsDouble();
```

Exercice 10 - Obtenir une collection de chiffres statistiques (c'est-à-dire somme, moyenne, max, min, nombre) pour tous les produits de la catégorie "Livres"

Que faire si vous avez besoin d'obtenir la somme, la moyenne, le max, le min et le compte en même temps ? Devrions-nous exécuter le flux de données 5 fois pour obtenir ces chiffres un par un ? Une telle approche n'est pas tout à fait efficace. Heureusement, l'API de flux fournit un moyen pratique d'obtenir toutes ces valeurs à la fois en utilisant l'opération de terminal **summaryStatistics**(). Il renvoie un type de données **DoubleSummaryStatistics** qui contient tous les chiffres requis.

Solution

```
DoubleSummaryStatistics statistics = productRepo.findAll()
    .stream()
    .filter(p -> p.getCategory().equalsIgnoreCase("Livres"))
    .mapToDouble(p -> p.getPrice())
```

```
    .summaryStatistics();

  System.out.println(String.format("count = %1$d, average = %2$f, max = %3$f, min = %4$f, sum = %5$f",
    statistics.getCount(), statistics.getAverage(), statistics.getMax(), statistics.getMin(), statistics.getSum())));
```

Exercice 11 - Obtenir une carte de données avec l'identifiant de la commande et le nombre de produits de la commande

À l'exception du calcul de valeur, tous les exercices précédents génèrent simplement une liste d'enregistrements. La classe d'assistance Collectors fournit un certain nombre d'opérations utiles pour la consolidation des données et la sortie de la collecte de données. Examinons l'exercice pour créer une carte de données avec l'ID de commande comme clé tandis que la valeur associée est le nombre de produits. L'opération de terminal **Collectors.toMap**() accepte deux arguments pour spécifier respectivement la clé et la valeur.

Solution

```
    Map<Long, Integer>  result = orderRepo.findAll()
        .stream()
        .collect(
            Collectors.toMap(
                order -> order.getId(),
                order -> order.getProducts().size()
                )
            );
```

Exercice 12 — Produire une carte de données avec des enregistrements de commande regroupés par client

Cet exercice consiste à consolider une liste de commandes par client. Collectors.groupingBy() est une fonction pratique, vous pouvez simplement spécifier quel est l'élément de données clé, il regroupera ensuite les données pour vous.

Solution

```
    Map<Customer, List<Order>> result = orderRepo.findAll()
        .stream()
        .collect(
            Collectors.groupingBy(Order::getCustomer)
            );
```

Exercice 13 — Produire une carte de données avec l'enregistrement de la commande et la somme totale des produits

La sortie de la carte de données cette fois n'est pas une simple extraction de champs de données du flux, vous devez créer un sous-flux pour chaque commande afin de calculer la somme totale du produit. Depuis, l'élément clé est l'enregistrement de la commande lui-même au lieu d'un identifiant de commande, donc Function.identity() est utilisé pour indiquer à Collectors.toMap() d'utiliser l'élément de données comme clé.

Solution

```
Map<Order, Double> result = orderRepo.findAll()
        .stream()
        .collect(
          Collectors.toMap(
              Function.identity(),
              order -> order.getProducts().stream()
                    .mapToDouble(p -> p.getPrice()).sum()
            )
          );
```

Exercice 14 — Obtenir une carte de données avec la liste des noms de produits par catégorie

Cet exercice vous aide à vous familiariser avec la manière de transformer la sortie de données des entrées de carte de données. Si vous utilisez uniquement Collectors.groupingBy(Product::getCategory), la sortie sera Map<Category, List of Products> mais la sortie attendue devrait être Map<Category, List of Product Name>. Vous pouvez utiliser Collectors.mapping() pour convertir les objets produit en noms de produit pour la construction de la carte de données.

Solution

```
Map<String, List<String>> result = productRepo.findAll()
        .stream()
        .collect(
            Collectors.groupingBy(
                Product::getCategory,
                Collectors.mapping(product -> product.getName(), Collectors.toList()))
            );
```

Exercice 15 — Obtenez le produit le plus cher par catégorie

Semblable à la transformation de données à l'aide de Collectors.mapping(), Collectors.maxBy() permet d'obtenir l'enregistrement avec la valeur maximale dans le cadre de la construction de la carte de données. En fournissant un comparateur de prix de produit, maxBy() est capable d'obtenir le produit avec la plus grande valeur pour chaque catégorie.

Solution

```
  Map<String, Optional<Product>> result = productRepo.findAll()
        .stream()
        .collect(
            Collectors.groupingBy(
                Product::getCategory,
                Collectors.maxBy(Comparator.comparing(Product::getPrice)))
```

Stream devoxx 2014

Le but du jeu est d'écrire une fonction permettant de concaténer un certain nombre de listes, passées en paramètre sous forme de *var-arg* :

```
public <T> List<T> concatLists(List<T>... lists);
```

Version 1 : foreach

Algorithme

Boucler sur chaque liste et les rajouter dans une liste.

Solution

```
public <T> List<T> concatLists1(List<T>... lists) {
    ArrayList<T> result = new ArrayList<>();
    for (List<T> list : lists) {
        result.addAll(list);
    }
    return result;
}
```

Version 2 : Reduce

Algorithme

L'opérateur Reduce est

```
T reduce(T identity,
         BinaryOperator<T> accumulator)
```

Effectue une réduction sur les éléments de ce flux, à l'aide de la valeur d'identité fournie et d'une fonction d'accumulation associative, et renvoie la valeur réduite. Cela équivaut à :

```
    T result = identity;
      for (T element : this stream)
          result = accumulator.apply(result, element)
      return result;
```

L'idée est de concater les listes avec un stream

Solution

```
public <T> List<T> concatLists2(List<T>... lists) {
    return Stream.of(lists).reduce(new ArrayList<>(), (list1, list2) -> {
        list1.addAll(list2);
        return list1;
    });
}
```

Version 3 via flatMap

Algorithme

```
<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)
```

La méthode **`flatMap()`** permet d'appliquer une fonction à chaque élément du flux puis d'aplatir le résultat en un nouveau flux.

Solution

```
public static <T> List<T> concatLists(List<T>... lists) {
       return Stream.of(lists).flatMap(List::stream).collect(Collectors.toList());
   }
```

Producteur Consommateur

La création d'un thread consomme une quantité importante de mémoire. Dans une application où il y a beaucoup de programmes clients, la création d'un thread par client n'évoluera pas. Ainsi, Java a proposé un framework d'exécuteur pour fournir un pool de threads pour l'exécution limitant le nombre de threads servant la demande du client à tout moment. Cela améliore les performances et réduit les besoins en mémoire.

Java 5 fournit également des implémentations de file d'attente de blocage et nous n'avons plus besoin de contrôler les applications producteur/consommateur à l'aide de wait/notify. Ceci est automatiquement pris en charge par les implémentations de BlockingQueue.

Un exemple de producteur/consommateur utilisant une implémentation de file d'attente bloquante et un framework d'exécuteur est le suivant :

```
public class ProducerConsumer {

    private static final int NUM_OF_MSGS = 20;
    private static final BlockingQueue<String> queue
                                          = new ArrayBlockingQueue<String>(5);
    private static ExecutorService producerPool = Executors.newFixedThreadPool(3);
    private static ExecutorService consumerPool = Executors.newFixedThreadPool(1);

    private static Logger logger =  Logger.getLogger(ProducerConsumer.class.getName());

    public static void main(String[] args) {
        Runnable producerTask = () -> {
            try {
                queue.put("test Message");
                System.out.println(Thread.currentThread().getName()
                                   + " put message queue.size() " + queue.size());
            } catch (InterruptedException e) {
                logger.log(Level.SEVERE, e.getMessage(), e);
            }
        };
        Runnable consumerTask = () -> {
            try {
                System.out.println(Thread.currentThread().getName()
```

```
                                    + " received msg " + queue.take());

            } catch (InterruptedException e) {
                logger.log(Level.SEVERE, e.getMessage(), e);
            }
        };
        try {
            for (int i = 0; i < NUM_OF_MSGS; i++) {
                producerPool.submit(producerTask);
            }
            for (int i = 0; i < NUM_OF_MSGS; i++) {
                consumerPool.submit(consumerTask);
            }
        } finally {
            if (producerPool != null) {
                producerPool.shutdown();
            }
            if (consumerPool != null) {
                consumerPool.shutdown();
            }
        }
    }

}
```

Le but de cela est de faire des opérations longues en mode asynchrone.

Ici nous avons une double queue avec une conversion HTML->PDF

```
public class ProducerConsumer2 {

    private static final int NUM_OF_MSGS = 20;
    private static final BlockingQueue<String> queueHtml
                                          = new ArrayBlockingQueue<String>(5);
    private static final BlockingQueue<File> queueFile= new ArrayBlockingQueue<File>(5);

    private static ExecutorService consumerHtmlPool = Executors.newFixedThreadPool(4);
    private static ExecutorService consumerFilePool = Executors.newFixedThreadPool(4);
```

```
    private static Logger logger =  Logger.getLogger(ProducerConsumer2.class.getName());

    public static void main(String[] args) throws IOException {

        Runnable consumerTask = () -> {
            try {

                String html=queueHtml.take();
                UUID uuid=UUID.randomUUID();
                File outputFile=new File(uuid.toString()+".pdf");
                HtmlConverter.convertToPdf(html, new FileOutputStream(outputFile));
                queueFile.put(outputFile);
            } catch (InterruptedException e) {
                logger.log(Level.SEVERE, e.getMessage(), e);
            } catch (FileNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        };
        Runnable consumerFileTask = () -> {
            try {

                File pdfFile=queueFile.take();
                System.err.println(pdfFile.getAbsolutePath());
            } catch (InterruptedException e) {
                logger.log(Level.SEVERE, e.getMessage(), e);
            }
        };
        try {
            for (int i = 0; i < NUM_OF_MSGS; i++) {
                consumerHtmlPool.submit(consumerFileTask);
            }
            for (int i = 0; i < NUM_OF_MSGS; i++) {
                consumerFilePool.submit(consumerTask);
            }
            queueHtml.add("\"<h1>Hello</h1>\"\r\n"
```

```
                + "    + \"<p>This was created using iText</p>\"\r\n"
                + "    ");
            System.in.read();
        } finally {
            if (consumerHtmlPool != null) {
                consumerHtmlPool.shutdown();
            }
            if (consumerFilePool != null) {
                consumerFilePool.shutdown();
            }
        }
    }

}
```

Le pom doit être modifié ainsi:

```
   <dependency>
        <groupId>com.itextpdf</groupId>
        <artifactId>itext7-core</artifactId>
        <version>7.1.9</version>
        <type>pom</type>
    </dependency>

    <!-- iText pdfHTML add-on -->
    <dependency>
        <groupId>com.itextpdf</groupId>
        <artifactId>html2pdf</artifactId>
        <version>2.1.6</version>
    </dependency>
```

Stream et JPA

L'utilisation des streams est pratique en JPA pour rajouter du filtrage sur des enité qui reviennent de la base.

```
  @Transactional(readOnly = true)
  public List<Employe> processEmploye(String email) {
    Stream<Employe> employeStream = bookRepository.getAll();
    return
    employeStream .filter(employe -> {
    
   boolean result= employe.getEmail().contains(email);
   entityManager.detach(employe);
   return result;
    }
    )
    .collect(Collectors.toList());


  }
}
```

ParalleleStream

Test Simple

```
public class SimpleParallele {

    public static void main(String[] args) {

        System.out.println("Normal...");

        IntStream range = IntStream.rangeClosed(1, 10);
        range.forEach(System.out::println);

        System.out.println("Parallel...");

        IntStream range2 = IntStream.rangeClosed(1, 10);
        range2.parallel().forEach(System.out::println);

    }

}
```

Résultat

```
Normal...
1
2
3
4
5
6
7
8
9
10
```

```
Parallel...
7
6
8
1
2
4
10
3
9
5
```

Test 2 Test des nombres premiers

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

      long now=System.currentTimeMillis();

        long count = Stream.iterate(0, n -> n + 1)
                .limit(1_00000)
                .filter(Prime::isPrime)
                .peek(x -> System.out.format("%s\t", x))
                .count();

        System.out.println("\nTotal: " + count);
        System.err.println("Time :" + (System.currentTimeMillis()-now));

  now=System.currentTimeMillis();

         count = Stream.iterate(0, n -> n + 1)
                .limit(1_00000)
                .parallel()
                .filter(Prime::isPrime)
                .peek(x -> System.out.format("%s\t", x))
                .count();

        System.out.println("\nTotal: " + count);
        System.err.println("Time :" + (System.currentTimeMillis()-now));
    }
```

```
    public static boolean isPrime(int number) {
        if (number <= 1) return false;
        return !IntStream.rangeClosed(2, number / 2).anyMatch(i -> number % i == 0);
    }
```

Résultat

```
Total: 9592
Time :1455

Total: 9592
Time :392
```

Stream

Conversion d'un liste en stream

```
mport java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

class Main
{
    // Generic method to convert a list to stream
    private static <T> Stream<T> listToStream (List<T> list) {
        return list.stream();
    }

    // Program to convert a list to stream in Java 8 and above
    public static void main(String[] args)
    {
        List<String> cities = Arrays.asList("New York","Tokyo","New Delhi");

        Stream<String> stream = listToStream(cities);
        System.out.println(Arrays.toString(stream.toArray()));
    }
}
```

Filtrage d'un stream

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;
import java.util.stream.Stream;

class Main
{
```

```
    // Program to convert a list to stream and filter it in Java 8 and above
    public static void main(String[] args)
    {
        List<String> cities = Arrays.asList("New York", "Tokyo", "New Delhi");

        Predicate<String> predicate = new Predicate<String>() {
            @Override
            public boolean test(String s) {
                // filter cities that start with `N`
                return s.startsWith("N");
            }
        };

        cities.stream()
                .filter(predicate)
                .forEach(System.out::println);
    }
}
```

Java 8 Date

Avec Java 8, une nouvelle API date-heure est introduite pour couvrir les inconvénients suivants de l'ancienne API date-heure.

Non thread-safe - java.util.Date n'est pas thread-safe, les développeurs doivent donc faire face à un problème de concurrence lors de l'utilisation de la date. La nouvelle API date-heure est immuable et n'a pas de méthodes de définition.

Mauvaise conception - La date par défaut commence à partir de 1900, le mois commence à partir de 1 et le jour commence à partir de 0, donc pas d'uniformité. L'ancienne API avait des méthodes moins directes pour les opérations de date. La nouvelle API fournit de nombreuses méthodes utilitaires pour de telles opérations.

Gestion difficile des fuseaux horaires − Les développeurs ont dû écrire beaucoup de code pour gérer les problèmes de fuseau horaire. La nouvelle API a été développée en gardant à l'esprit la conception spécifique au domaine.

Java 8 introduit une nouvelle API date-heure sous le package java.time. Voici quelques-unes des classes importantes introduites dans le package java.time.

Local - API date-heure simplifiée sans complexité de gestion du fuseau horaire.

Zoned - API date-heure spécialisée pour gérer différents fuseaux horaires.

LocalDate/LocalTime et LocalDateTime

Les classes LocalDate/LocalTime et LocalDateTime simplifient le développement là où les fuseaux horaires ne sont pas nécessaires.

```
import java.time.LocalDate;
import java.time.LocalTime;
import java.time.LocalDateTime;
import java.time.Month;

public class Java8Tester {

   public static void main(String args[]) {
      Java8Tester java8tester = new Java8Tester();
      java8tester.testLocalDateTime();
```

```
    }

    public void testLocalDateTime() {
        // Get the current date and time
        LocalDateTime currentTime = LocalDateTime.now();
        System.out.println("Current DateTime: " + currentTime);

        LocalDate date1 = currentTime.toLocalDate();
        System.out.println("date1: " + date1);

        Month month = currentTime.getMonth();
        int day = currentTime.getDayOfMonth();
        int seconds = currentTime.getSecond();

        System.out.println("Month: " + month +"day: " + day +"seconds: " + seconds);

        LocalDateTime date2 = currentTime.withDayOfMonth(10).withYear(2012);
        System.out.println("date2: " + date2);

        //12 december 2014
        LocalDate date3 = LocalDate.of(2014, Month.DECEMBER, 12);
        System.out.println("date3: " + date3);

        //22 hour 15 minutes
        LocalTime date4 = LocalTime.of(22, 15);
        System.out.println("date4: " + date4);

        //parse a string
        LocalTime date5 = LocalTime.parse("20:15:30");
        System.out.println("date5: " + date5);
    }
}
```

Il devrait produire la sortie suivante :

```
Current DateTime: 2014-12-09T11:00:45.457
date1: 2014-12-09
Month: DECEMBERday: 9seconds: 45
date2: 2012-12-10T11:00:45.457
date3: 2014-12-12
```

```
date4: 22:15
date5: 20:15:30
```

ZonedDateTime

L'API date-heure zonée doit être utilisée lorsque le fuseau horaire doit être pris en compte.

```
import java.time.ZonedDateTime;
import java.time.ZoneId;

public class Java8Tester {

   public static void main(String args[]) {
      Java8Tester java8tester = new Java8Tester();
      java8tester.testZonedDateTime();
   }

   public void testZonedDateTime() {
      // Get the current date and time
      ZonedDateTime date1 = ZonedDateTime.parse("2007-12-03T10:15:30+05:30[Asia/Karachi]");
      System.out.println("date1: " + date1);

      ZoneId id = ZoneId.of("Europe/Paris");
      System.out.println("ZoneId: " + id);

      ZoneId currentZone = ZoneId.systemDefault();
      System.out.println("CurrentZone: " + currentZone);
   }
}
```

Il devrait produire la sortie suivante :

```
date1: 2007-12-03T10:15:30+05:00[Asia/Karachi]
ZoneId: Europe/Paris
CurrentZone: Etc/UTC
```

ChronoUnit

L'énumération java.time.temporal.ChronoUnit est ajoutée dans Java 8 pour remplacer les valeurs entières utilisées dans l'ancienne API pour représenter le jour, le mois, etc.

```
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public class Java8Tester {

   public static void main(String args[]) {
      Java8Tester java8tester = new Java8Tester();
      java8tester.testChromoUnits();
   }

   public void testChromoUnits() {
      //Get the current date
      LocalDate today = LocalDate.now();
      System.out.println("Current date: " + today);

      //add 1 week to the current date
      LocalDate nextWeek = today.plus(1, ChronoUnit.WEEKS);
      System.out.println("Next week: " + nextWeek);

      //add 1 month to the current date
      LocalDate nextMonth = today.plus(1, ChronoUnit.MONTHS);
      System.out.println("Next month: " + nextMonth);

      //add 1 year to the current date
      LocalDate nextYear = today.plus(1, ChronoUnit.YEARS);
      System.out.println("Next year: " + nextYear);

      //add 10 years to the current date
      LocalDate nextDecade = today.plus(1, ChronoUnit.DECADES);
      System.out.println("Date after ten year: " + nextDecade);
   }
}
```

La sortie est

```
Current date: 2014-12-10
Next week: 2014-12-17
Next month: 2015-01-10
Next year: 2015-12-10
Date after ten year: 2024-12-10
```

Duration

Avec Java 8, deux classes spécialisées sont introduites pour gérer les décalages horaires.

Période - Il traite de la durée basée sur la date.

Durée - Il s'agit d'une durée basée sur le temps.

```
import java.time.temporal.ChronoUnit;

import java.time.LocalDate;
import java.time.LocalTime;
import java.time.Duration;
import java.time.Period;

public class Java8Tester {

   public static void main(String args[]) {
      Java8Tester java8tester = new Java8Tester();
      java8tester.testPeriod();
      java8tester.testDuration();
   }

   public void testPeriod() {
      //Get the current date
      LocalDate date1 = LocalDate.now();
      System.out.println("Current date: " + date1);

      //add 1 month to the current date
      LocalDate date2 = date1.plus(1, ChronoUnit.MONTHS);
      System.out.println("Next month: " + date2);

      Period period = Period.between(date2, date1);
      System.out.println("Period: " + period);
```

```
   }

   public void testDuration() {
      LocalTime time1 = LocalTime.now();
      Duration twoHours = Duration.ofHours(2);

      LocalTime time2 = time1.plus(twoHours);
      Duration duration = Duration.between(time1, time2);

      System.out.println("Duration: " + duration);
   }
}
```

Compatibilité

Une méthode toInstant() est ajoutée aux objets Date et Calendar d'origine, qui peuvent être utilisés pour les convertir vers la nouvelle API Date-Heure. Utilisez une méthode ofInstant(Insant,ZoneId) pour obtenir un objet LocalDateTime ou ZonedDateTime.

```
import java.time.LocalDateTime;
import java.time.ZonedDateTime;

import java.util.Date;

import java.time.Instant;
import java.time.ZoneId;

public class Java8Tester {

   public static void main(String args[]) {
      Java8Tester java8tester = new Java8Tester();
      java8tester.testBackwardCompatability();
   }

   public void testBackwardCompatability() {
      //Get the current date
      Date currentDate = new Date();
      System.out.println("Current date: " + currentDate);
```

```
		//Get the instant of current date in terms of milliseconds
		Instant now = currentDate.toInstant();
		ZoneId currentZone = ZoneId.systemDefault();

		LocalDateTime localDateTime = LocalDateTime.ofInstant(now, currentZone);
		System.out.println("Local date: " + localDateTime);

		ZonedDateTime zonedDateTime = ZonedDateTime.ofInstant(now, currentZone);
		System.out.println("Zoned date: " + zonedDateTime);
	}
}
```

Il devrait produire la sortie:

```
Current date: Wed Dec 10 05:44:06 UTC 2014
Local date: 2014-12-10T05:44:06.635
Zoned date: 2014-12-10T05:44:06.635Z[Etc/UTC]
```

Exercice Date

Ecrire un programme Java pour créer un objet Date en utilisant la classe Calendar

```
import java.util.*;
public class Exercise1 {
    public static void main(String[] args)
    {
     int year = 2016;
     int month = 0; // January
     int date = 1;

     Calendar cal = Calendar.getInstance();
     // Sets the given calendar field value and the time value
     // (millisecond offset from the Epoch) of this Calendar undefined.
     cal.clear();
     System.out.println();
     cal.set(Calendar.YEAR, year);
     cal.set(Calendar.MONTH, month);
     cal.set(Calendar.DATE, date);

     System.out.println(cal.getTime());
     System.out.println();
  }
}
```

Écrire un programme Java pour obtenir et afficher les informations (année, mois, jour, heure, minute) d'un calendrier par défaut

```
import java.util.*;
public class Exercise2 {
 public static void main(String[] args)
    {
       // Create a default calendar
        Calendar cal = Calendar.getInstance();
       // Get and display information of current date from the calendar:
     System.out.println();
        System.out.println("Year: " + cal.get(Calendar.YEAR));
        System.out.println("Month: " + cal.get(Calendar.MONTH));
        System.out.println("Day: " + cal.get(Calendar.DATE));
        System.out.println("Hour: " + cal.get(Calendar.HOUR));
        System.out.println("Minute: " + cal.get(Calendar.MINUTE));
     System.out.println();
    }
}
```

Écrivez un programme Java pour obtenir la valeur maximale de l'année, du mois, de la semaine et de la date à partir de la date actuelle d'un calendrier par défaut.

```
import java.util.*;
public class Exercise3 {
public static void main(String[] args)
    {
     // Create a default calendar
        Calendar cal = Calendar.getInstance();
  System.out.println();
  System.out.println("\nCurrent Date and Time:" + cal.getTime());
  int actualMaxYear = cal.getActualMaximum(Calendar.YEAR);
  int actualMaxMonth = cal.getActualMaximum(Calendar.MONTH);
  int actualMaxWeek = cal.getActualMaximum(Calendar.WEEK_OF_YEAR);
  int actualMaxDate = cal.getActualMaximum(Calendar.DATE);
```

```
    System.out.println("Actual Maximum Year: "+actualMaxYear);
    System.out.println("Actual Maximum Month: "+actualMaxMonth);
    System.out.println("Actual Maximum Week of Year: "+actualMaxWeek);
    System.out.println("Actual Maximum Date: "+actualMaxDate+"\n");
    System.out.println();
    
  }
}
```

Écrivez un programme Java pour obtenir la valeur minimale de l'année, du mois, de la semaine et de la date à partir de la date actuelle d'un calendrier par défaut.

```
import java.util.*;
public class Exercise4 {
 public static void main(String[] args)
    {
     // Create a default calendar
        Calendar cal = Calendar.getInstance();
    System.out.println();
    System.out.println("\nCurrent Date and Time:" + cal.getTime());
    int actualMaxYear = cal.getActualMinimum(Calendar.YEAR);
    int actualMaxMonth = cal.getActualMinimum(Calendar.MONTH);
    int actualMaxWeek = cal.getActualMinimum(Calendar.WEEK_OF_YEAR);
    int actualMaxDate = cal.getActualMinimum(Calendar.DATE);
    
    System.out.println("Actual Minimum Year: "+actualMaxYear);
    System.out.println("Actual Minimum Month: "+actualMaxMonth);
    System.out.println("Actual Minimum Week of Year: "+actualMaxWeek);
    System.out.println("Actual Minimum Date: "+actualMaxDate+"\n");
    System.out.println();
  }
}
```

Écrivez un programme Java pour obtenir l'heure actuelle à New York.

```
import java.util.*;
public class Exercise5 {
   public static void main(String[] args)
    {
     Calendar calNewYork = Calendar.getInstance();
     calNewYork.setTimeZone(TimeZone.getTimeZone("America/New_York"));
     System.out.println();
      System.out.println("Time in New York: " + calNewYork.get(Calendar.HOUR_OF_DAY) + ":"
        + calNewYork.get(Calendar.MINUTE)+":"+calNewYork.get(Calendar.SECOND));
     System.out.println();
    }
}
```

Écrivez un programme Java pour obtenir la date et l'heure complètes actuelles

```
import java.util.*;
public class Exercise6 {
   public static void main(String[] args)
    {
   Calendar now = Calendar.getInstance();
    System.out.println();
    System.out.println("Current full date and time is : " + (now.get(Calendar.MONTH) + 1) + "-
"
        + now.get(Calendar.DATE) + "-" + now.get(Calendar.YEAR) + " "
        + now.get(Calendar.HOUR_OF_DAY) + ":" + now.get(Calendar.MINUTE) + ":"
        + now.get(Calendar.SECOND) + "." + now.get(Calendar.MILLISECOND));
System.out.println();
    }
}
```

Écrivez un programme Java pour obtenir le dernier jour du mois en cours.

```
import java.util.*;
public class Exercise7 {
   public static void main(String[] args)
    {
    //Gets a calendar using the default time zone and locale.
        Calendar calendar = Calendar.getInstance();
   System.out.println();
   System.out.println(calendar.getActualMaximum(Calendar.DAY_OF_MONTH));
   System.out.println();
   }
}
```

Écrivez un programme Java pour obtenir l'heure locale actuelle.

```
import java.time.*;
public class Exercise14 {
   public static void main(String[] args)
    {
        LocalTime time = LocalTime.now();
        System.out.println();
        // in hour, minutes, seconds, nano seconds
        System.out.println("Local time now : " + time);
        System.out.println();
    }
}
```

Écrivez un programme Java pour ajouter quelques heures à l'heure actuelle.

```
import java.time.*;
public class Exercise15 {
   public static void main(String[] args)
    {
      LocalTime time = LocalTime.now();
      // adding four hours
      LocalTime newTime = time.plusHours(4);
      System.out.println();
      System.out.println("Time after 2 hours : " + newTime);
      System.out.println();
    }
}
```

Écrivez un programme Java pour obtenir une date après 2 semaines.

```
import java.util.*;
public class Exercise16 {
   public static void main(String[] args)
    {
     //two weeks
     int noOfDays = 14;
     Calendar cal = Calendar.getInstance();
     Date cdate = cal.getTime();
     cal.add(Calendar.DAY_OF_YEAR, noOfDays);
     Date date = cal.getTime();
     System.out.println("\nCurrent Date: " + cdate+"\n");
     System.out.println("Day after two weeks: " + date+"\n");
    }
}
```

Écrivez un programme Java pour obtenir une date avant et après 1 an par rapport à la date actuelle.

```
import java.util.*;
public class Exercise17 {
   public static void main(String[] args)
    {
      Calendar cal = Calendar.getInstance();
      Date cdate = cal.getTime();
      // get next year
      cal.add(Calendar.YEAR, 1);
      Date nyear = cal.getTime();
      //get previous year
      cal.add(Calendar.YEAR, -2);
      Date pyear = cal.getTime();
      System.out.println("\nCurrent Date : " + cdate);
      System.out.println("\nDate before 1 year : " + pyear);
      System.out.println("\nDate after 1 year  : " + nyear+"\n");
    }
}
```

Écrivez un programme Java pour vérifier qu'une année est bissextile ou non.

```
public class Exercise18 {
   public static void main(String[] args)
    {
     //year to leap year or not
       int year = 2016;
       System.out.println();
       if((year % 400 == 0) || ((year % 4 == 0) && (year % 100 != 0)))
               System.out.println("Year " + year + " is a leap year");
       else
               System.out.println("Year " + year + " is not a leap year");
               System.out.println();
    }
}
```

Écrivez un programme Java pour obtenir l'année et les mois entre deux dates.

```
import java.time.*;
public class Exercise19 {
   public static void main(String[] args)
    {
     LocalDate today = LocalDate.now();
     LocalDate userday = LocalDate.of(2015, Month.MAY, 15);
     Period diff = Period.between(userday, today);
     System.out.println("\nDifference between "+ userday +" and "+ today +": "
     + diff.getYears() +" Year(s) and "+ diff.getMonths() +" Month()s\n");
    }
}
```

Écrivez un programme Java pour obtenir l'horodatage actuel.

```
import java.time.*;
public class Exercise20 {
   public static void main(String[] args)
    {
    Instant timestamp = Instant.now();
    System.out.println("\nCurrent Timestamp: " + timestamp+"\n");
    }
}
```

Écrivez un programme Java pour obtenir l'heure actuelle dans tous les fuseaux horaires disponibles.

```
import java.time.*;
public class Exercise21 {
   public static void main(String[] args)
    {
    ZoneId.SHORT_IDS.keySet().
    stream().forEach(
    zoneKey ->System.out.println(" "+ ZoneId.of( ZoneId.SHORT_IDS.get( zoneKey ) ) +": "+
LocalDateTime.now(ZoneId.of(ZoneId.SHORT_IDS.get( zoneKey ) ) ) ) );
    }
}
```

Écrivez un programme Java pour obtenir les dates 10 jours avant et après aujourd'hui.

```
import java.time.*;
public class Exercise22 {
   public static void main(String[] args)
    {
     LocalDate today = LocalDate.now();
     System.out.println("\nCurrent Date: "+today);
     System.out.println("10 days before today will be "+today.plusDays(-10));
     System.out.println("10 days after today will be "+today.plusDays(10)+"\n");
   }
}
```

Écrivez un programme Java pour obtenir les mois restants dans l'année.

```
import java.time.*;
import java.time.temporal.TemporalAdjusters;
public class Exercise23 {
   public static void main(String[] args)
    {
```

```
      LocalDate today = LocalDate.now();
      LocalDate lastDayOfYear = today.with(TemporalAdjusters.lastDayOfYear());
      Period period = today.until(lastDayOfYear);
      System.out.println();
      System.out.println("Months remaining in the year: "+period.getMonths());
      System.out.println();
    }
}
```

Écrivez un programme Java pour afficher les dates dans les formats suivants.

LocalDate=2016-09-16
16::Sep::2016
LocalDateTime=2016-09-16T11:46:01.455
16::Sep::2016 11::46::01
Instant=2016-09-16T06:16:01.456Z
Default format after parsing = 2014-04-27T21:39:48

```
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class DateParseFormatExercise24 {

	public static void main(String[] args) {
		
		//Format examples
		LocalDate date = LocalDate.now();
		//default format
		System.out.println("\nDefault format of LocalDate="+date);
		//specific format
		System.out.println(date.format(DateTimeFormatter.ofPattern("d::MMM::uuuu")));

		LocalDateTime dateTime = LocalDateTime.now();
		//default format
		System.out.println("Default format of LocalDateTime="+dateTime);
```

```
		//specific format
		System.out.println(dateTime.format(DateTimeFormatter.ofPattern("d::MMM::uuuu HH::mm::ss")));
			Instant timestamp = Instant.now();
		//default format
		System.out.println("Default format of Instant="+timestamp);
		
		//Parse examples
		LocalDateTime dt = LocalDateTime.parse("27::Apr::2014 21::39::48",
				DateTimeFormatter.ofPattern("d::MMM::uuuu HH::mm::ss"));
		System.out.println("Default format after parsing = "+dt+"\n");
	}

}
```

Écrivez un programme Java pour afficher les informations de date et d'heure avant quelques heures et minutes à partir de la date et de l'heure actuelles

```
import java.time.*;

public class DateParseFormatExample28 {

	public static void main(String[] args) {
	
     // Before 7 houres and 30 minutes
      LocalDateTime dateTime = LocalDateTime.now().minusHours(5).minusMinutes(30);
       System.out.println("\nCurrent Date and Time:   " + LocalDateTime.now());
       System.out.println("Before 7 hours and 30 minutes: " + dateTime+"\n");
	}

}
```

Écrivez un programme Java pour convertir une chaîne en date.

```
import java.time.*;
import java.util.*;
import java.time.format.DateTimeFormatter;

public class MainEx29 {
  public static void main(String[] args) {
   String string = "May 1, 2016";
   DateTimeFormatter formatter = DateTimeFormatter.ofPattern("MMMM d, yyyy", Locale.ENGLISH);
   LocalDate date = LocalDate.parse(string, formatter);
   System.out.println();
   System.out.println(date);
   System.out.println();
  }
}
```

Ecrire un programme Java pour calculer la différence entre deux dates (année, mois, jours)

```
import java.time.*;
import java.util.*;

public class Exercise1 {
   public static void main(String[] args)
    {
        LocalDate pdate = LocalDate.of(2012, 01, 01);
        LocalDate now = LocalDate.now();

        Period diff = Period.between(pdate, now);

     System.out.printf("\nDifference is %d years, %d months and %d days old\n\n",
                    diff.getYears(), diff.getMonths(), diff.getDays());
```

```
  }
}
```

Écrivez un programme Java pour calculer la différence entre deux dates (heures, minutes, milli, secondes et nano).

```
import java.time.*;
import java.util.*;

public class Exercise31 {
   public static void main(String[] args)
    {
     LocalDateTime dateTime = LocalDateTime.of(2016, 9, 16, 0, 0);
     LocalDateTime dateTime2 = LocalDateTime.now();
     int diffInNano = java.time.Duration.between(dateTime, dateTime2).getNano();
     long diffInSeconds = java.time.Duration.between(dateTime, dateTime2).getSeconds();
     long diffInMilli = java.time.Duration.between(dateTime, dateTime2).toMillis();
     long diffInMinutes = java.time.Duration.between(dateTime, dateTime2).toMinutes();
     long diffInHours = java.time.Duration.between(dateTime, dateTime2).toHours();
     System.out.printf("\nDifference is %d Hours, %d Minutes, %d Milli, %d Seconds and %d
Nano\n\n",
                  diffInHours, diffInMinutes, diffInMilli, diffInSeconds, diffInNano );

  }
}
```

Écrivez un programme Java pour extraire la date, l'heure de la chaîne de date.

```
import java.util.*;
import java.text.*;
```

```
public class Exercise35 {
  public static void main(String[] args)
   {
    try {
      String originalString = "2016-07-14 09:00:02";
      Date date = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").parse(originalString);
      String newstr = new SimpleDateFormat("MM/dd/yyyy, H:mm:ss").format(date);
       System.out.println("\n"+newstr+"\n");
      }
    catch (ParseException e) {
    //Handle exception here
     e.printStackTrace();
     }
    }
}
```

Écrivez un programme Java pour convertir un horodatage Unix en date en Java.

```
import java.util.*;
import java.text.*;

public class Exercise36 {
  public static void main(String[] args)
   {
   //Unix seconds
   long unix_seconds = 1372339860;
   //convert seconds to milliseconds
   Date date = new Date(unix_seconds*1000L);
   // format of the date
   SimpleDateFormat jdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss z");
   jdf.setTimeZone(TimeZone.getTimeZone("GMT-4"));
   String java_date = jdf.format(date);
   System.out.println("\n"+java_date+"\n");
   }
}
```

Écrivez un programme Java pour calculer la différence entre deux dates en jours.

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.Period;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.temporal.ChronoUnit;
import java.util.Calendar;
import java.util.concurrent.TimeUnit;

public class Main {

    public static void main(String[] args) {

        System.out.println("\nBefore JDK 8:");

        Calendar cal1 = Calendar.getInstance();
        cal1.set(2019, 0, 1);
        Calendar cal2 = Calendar.getInstance();
        cal2.set(2020, 2, 1);

        System.out.println("\nDate/Calendar case: " + cal1.getTime() + " <-> " +
cal2.getTime());

        long inMs = Math.abs(cal1.getTimeInMillis() - cal2.getTimeInMillis());
        long inDays = Math.abs(TimeUnit.DAYS.convert(inMs, TimeUnit.MILLISECONDS));

        System.out.println("Difference in milliseconds is: " + inMs);
        System.out.println("Difference in days is: " + inDays);

        System.out.println("\nStarting with JDK 8:");

        LocalDate ld1 = LocalDate.of(2019, 1, 1);
        LocalDate ld2 = LocalDate.of(2020, 3, 1);

        System.out.println("\nLocalDate case: " + ld1 + " <-> " + ld2);
```

```
        long between_In_Days = Math.abs(ChronoUnit.DAYS.between(ld1, ld2));
        long between_In_Months = Math.abs(ChronoUnit.MONTHS.between(ld1, ld2));
        long between_In_Years = Math.abs(ChronoUnit.YEARS.between(ld1, ld2));
        long until_In_Days = Math.abs(ld1.until(ld2, ChronoUnit.DAYS));
        long until_In_Months = Math.abs(ld1.until(ld2, ChronoUnit.MONTHS));
        long until_In_Years = Math.abs(ld1.until(ld2, ChronoUnit.YEARS));
        Period period = ld1.until(ld2);
        System.out.println("Difference as Period: "
                + period.getYears() + "y" + period.getMonths() + "m" + period.getDays() +
"d");
        System.out.println("Difference in days is via between(): " + between_In_Days);
        System.out.println("Difference in months is via between(): " + between_In_Months);
        System.out.println("Difference in years is via between(): " + between_In_Years);
        System.out.println("Difference in days is via until(): " + until_In_Days);
        System.out.println("Difference in months is via until(): " + until_In_Months);
        System.out.println("Difference in years is via until(): " + until_In_Years);

        LocalDateTime ldt1 = LocalDateTime.of(2019, 1, 1, 22, 15, 15);
        LocalDateTime ldt2 = LocalDateTime.of(2020, 1, 1, 23, 15, 15);

        System.out.println("\nLocalDateTime case: " + ldt1 + " <-> " + ldt2);

        long betweenInMinutesWithoutZone = Math.abs(ChronoUnit.MINUTES.between(ldt1, ldt2));
        long untilInMinutesWithoutZone = Math.abs(ldt1.until(ldt2, ChronoUnit.HOURS));
        System.out.println("Difference in minutes without zone: " +
betweenInMinutesWithoutZone);
        System.out.println("Difference in hours without zone: " + untilInMinutesWithoutZone);

        System.out.println("\nZonedDateTime case:");

        ZonedDateTime zdt1 = ldt1.atZone(ZoneId.of("Europe/Bucharest"));
        ZonedDateTime zdt2 =
zdt1.withZoneSameInstant(ZoneId.of("Australia/Perth")).plusHours(1);
        ZonedDateTime zdt3 = ldt2.atZone(ZoneId.of("Australia/Perth"));

        long betweenInMinutesWithZone12 = Math.abs(ChronoUnit.MINUTES.between(zdt1, zdt2));
        long untilInHoursWithZone12 = Math.abs(zdt1.until(zdt2, ChronoUnit.HOURS));
        long betweenInMinutesWithZone13 = Math.abs(ChronoUnit.MINUTES.between(zdt1, zdt3));
        long untilInHoursWithZone13 = Math.abs(zdt1.until(zdt3, ChronoUnit.HOURS));
```

```
        System.out.println("Europe/Bucharest: " + zdt1 + " <-> Australia/Perth: " + zdt2);
        System.out.println("Difference in minutes with zone (same instant): " +
betweenInMinutesWithZone12);
        System.out.println("Difference in hours with zone (same instant): " +
untilInHoursWithZone12);

        System.out.println("\nEurope/Bucharest: " + zdt1 + " <-> Australia/Perth: " + zdt3);
        System.out.println("Difference in minutes with zone (not same instant): " +
betweenInMinutesWithZone13);
        System.out.println("Difference in hours with zone: " + untilInHoursWithZone13);
    }
}
```

Écrivez un programme Java pour imprimer aaaa-MM-jj, HH:mm:ss, aaaa-MM-jj HH:mm:ss, E MMM aaaa HH:mm:ss.SSSZ et HH:mm:ss,Z, modèle de format pour la date et l'heure

```
import java.text.SimpleDateFormat;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.OffsetDateTime;
import java.time.ZonedDateTime;
import java.time.OffsetTime;
import java.time.format.DateTimeFormatter;
import java.util.Date;
public class Main {
public static void main(String[] args) {
              String result;
       //yyyy-MM-dd
       LocalDate localDate = LocalDate.now();
```

```
        DateTimeFormatter formatterLocalDate = DateTimeFormatter.ofPattern("yyyy-MM-dd");
        result = formatterLocalDate.format(localDate);
        System.out.println("\nyyyy-MM-dd: " + result);
        // HH:mm:ss
        LocalTime localTime = LocalTime.now();
        DateTimeFormatter formatterLocalTime = DateTimeFormatter.ofPattern("HH:mm:ss");
        result = formatterLocalTime.format(localTime);
                        System.out.println("\nHH:mm:ss: " + result);
        // yyyy-MM-dd HH:mm:ss
        LocalDateTime localDateTime = LocalDateTime.now();
        DateTimeFormatter formatterLocalDateTime =
                DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        result = formatterLocalDateTime.format(localDateTime);
                        System.out.println("\nyyyy-MM-dd HH:mm:ss: " + result);
        // E MMM yyyy HH:mm:ss.SSSZ
        ZonedDateTime zonedDateTime = ZonedDateTime.now();
        DateTimeFormatter formatterZonedDateTime =
                DateTimeFormatter.ofPattern("E MMM yyyy HH:mm:ss.SSSZ");
        result = formatterZonedDateTime.format(zonedDateTime);
                        System.out.println("\nE MMM yyyy HH:mm:ss.SSSZ: " + result);
        // HH:mm:ss,Z
        OffsetTime offsetTime = OffsetTime.now();
        DateTimeFormatter formatterOffsetTime =
                DateTimeFormatter.ofPattern("HH:mm:ss,Z");
        result = formatterOffsetTime.format(offsetTime);
                        System.out.println("\nHH:mm:ss,Z: " + result);
       }
}
```

Écrivez un programme Java pour afficher la date et l'heure locales combinées dans un seul objet.

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
```

```
import java.time.format.DateTimeFormatter;

public class Main {
    public static void main(String[] args) {
        LocalDate local_Dt = LocalDate.now();
        String localDateAsString = local_Dt
                .format(DateTimeFormatter.ofPattern("yyyy-MMM-dd"));
        System.out.println("Local Date: " + localDateAsString);
        LocalTime local_Time = LocalTime.now();
        String localTimeAsString = local_Time
                .format(DateTimeFormatter.ofPattern("hh:mm:ss a"));
        System.out.println("Local Time: " + localTimeAsString);
        LocalDateTime localDateTime = LocalDateTime.of(local_Dt, local_Time);
        String localDateTimeAsString = localDateTime
                .format(DateTimeFormatter.ofPattern("yyyy-MMM-dd hh:mm:ss a"));
        System.out.println("\nCombine local Date Time: " + localDateTimeAsString);
    }
}
```

Écrivez un programme Java pour définir une période de temps en utilisant des valeurs basées sur la date (Period) et une durée en utilisant des valeurs basées sur le temps (Duration).

```
import java.time.LocalDate;
import java.time.Period;

public class Main {

    public static void main(String[] args) {

        Period fromDays = Period.ofDays(120);
        System.out.println("Period from days: " + fromDays);

```

```
        Period periodFromUnits = Period.of(2000, 11, 24);
        System.out.println("Period from units: " + periodFromUnits);

        LocalDate localDate = LocalDate.now();
        Period periodFromLocalDate = Period.of(localDate.getYear(),
                localDate.getMonthValue(), localDate.getDayOfMonth());
        System.out.println("Period from LocalDate: " + periodFromLocalDate);

        Period periodFromString = Period.parse("P2019Y2M25D");
        System.out.println("Period from String: " + periodFromString);

        LocalDate startLocalDate = LocalDate.of(2018, 3, 12);
        LocalDate endLocalDate = LocalDate.of(2019, 7, 20);
        Period periodBetween = Period.between(startLocalDate, endLocalDate);
        System.out.println("\nBetween " + startLocalDate + " and "
                + endLocalDate + " there are " + periodBetween.getYears() + " year(s)");
        System.out.println("Between " + startLocalDate + " and "
                + endLocalDate + " there are " + periodBetween.getMonths() + " month(s)");
        System.out.println("Between " + startLocalDate + " and "
                + endLocalDate + " there are " + periodBetween.getDays() + " days(s)");

        System.out.println("Expressed as y:m:d: " + periodToYMD(periodBetween));

        System.out.println(startLocalDate + " is after "
                + endLocalDate + " ? " + periodBetween.isNegative());

        Period periodBetweenPlus1Year = periodBetween.plusYears(1L);
        System.out.println("\n" + periodBetween + " has " + periodBetween.getYears() + "
year,"
                + " after adding one year it has " + periodBetweenPlus1Year.getYears());

        Period p1 = Period.ofDays(5);
        Period p2 = Period.ofDays(20);
        Period p1p2 = p1.plus(p2);
        System.out.println(p1 + "+" + p2 + "=" + p1p2);
    }

    private static String periodToYMD(Period period) {

        if (period == null) {
            // or throw IllegalArgumentException
```

```
            return "";
        }

        StringBuilder sb = new StringBuilder();
        sb.append(period.getYears())
                .append("y:")
                .append(period.getMonths())
                .append("m:")
                .append(period.getDays())
                .append("d");

        return sb.toString();
    }
}
```