Introduction à la Programmation Objet

2h, aucun document autorisé
Le barème est donné à titre indicatif et pourra être modifié

19 octobre 2020

1 Questions de cours (3 points)

Question 1 Quelles conditions doivent être respectées pour obtenir un bon nom de méthode ?

Question 2 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)
```

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

Question 3 Dans un projet logiciel industriel, qu'est-ce qui coûtera en général le plus cher : la spécification des besoins, le développement initial, le test ou la maintenance ?

2 Codage (12 points)

Dans cet exercice, nous allons trier des textes dans différentes langues, en utilisant la classe `java.util.Locale`.

Les objets de cette classe servent à désigner une région ou une langue, dans ses différentes variantes possibles. Nous nous intéressons ici aux caractéristiques *language* (la langue, par exemple l'anglais, critère principal utilisé), *country* (le pays, par exemple US), et *variant* (la variante régionale).

Un extrait de sa javadoc, ainsi que de celles de ArrayList et HashMap, se trouvent en annexe.

Problème : On veut dans cet exercice traiter des textes formatés de la manière suivante :

- Ils sont sous forme de chaîne de caractères (String).
- La première ligne désigne la langue du texte sous la forme suivante :
  - la langue, sous forme ISO 639 (2 ou 3 caractères alphanumériques) suivie d'un espace
  - puis, éventuellement, le pays sous forme ISO 3166 (2 lettres ou 3 chiffres) suivi d'un espace
  - puis, éventuellement et uniquement dans le cas où le pays est indiqué, la variante sous une forme compatible avec la propriété correspondante de `Locale` suivie d'un espace
  - et finit par un saut de ligne (`'\n'`), toujours présent.
- La suite de la chaîne de caractère est le contenu du texte.

Vous pouvez trouver de (brefs) exemples de tels textes dans la fonction main du code joint.
Dans toute la section, on supposera les textes bien formés (pas de gestion d'erreur).

Rappels de méthodes utiles :

- la méthode `static void System.out.println(String str)` permet l'affichage de str sur la sortie standard
- la méthode `String[] split(String regexp)` de la classe String permet le découpage d'une chaîne autour de chaque sous-chaîne `regexp` rencontrée. Elle prend une *expression régulière* en paramètre permettant l'utilisation de caractères spéciaux pour correspondre à divers cas. Pour cet exercice, vous n'avez pas besoin d'utiliser de caractères spéciaux ou de les échapper. (Ignorez le fait qu'il s'agit d'une expression régulière, considérez qu'elle prend une chaîne de caractère fixe en paramètre).

Le but, à travers les question de cette section, est de compléter la classe suivante :

```
1  import java.util.ArrayList;
2  import java.util.HashMap;
3  import java.util.Locale;
4
5  public class TextFilterByLanguage {
6
7      public static Locale readLang(String text) {
8          //TODO
9      }
10
11     public static Locale readLocale(String text) {
12         //TODO
13     }
14
15     public static void displayLanguage(String text){
16         //TODO
17     }
18
19     /**
20      * Filter the collection of text according to the given language
21      *
22      * @param textCollection: a collection of well-formed texts,
23      *                        where their language is given in the first line
24      * @param lang: the chosen language, as in would displayed for the user
25      * @return a list of the texts in the chosen language
26      */
27     public static ArrayList<String> filterByLang(
28                 ArrayList<String> textCollection, String lang){
29         //TODO
30     }
31
32     /**
33      *
34      * @param textCollection
35      * @return a map between languages in their display form
36                     and the texts in this langage
37      */
38     public static HashMap<String, ArrayList<String>> groupByLang(
39                 ArrayList<String> textCollection){
40         //TODO
41     }
42
43     public static void main(String[] args) {
44
45         String texEn1 = "en␣/n␣hello␣world";
46         String texEn2 = "en␣ca␣canada␣/n␣Oh␣Canada!";
47         String texFr1 = "fr␣/n␣Bonjour␣!␣/n␣Comment␣tu␣vas␣?";
48         String texEn3 = "en␣us␣/n␣Do␣you␣want␣to␣initialize␣this?";
49         ArrayList<String> texts = new ArrayList<>();
50         texts.add(texEn1);
51         texts.add(texEn2);
52         texts.add(texFr1);
```

```
53          texts.add(texEn3);
54
55          displayLanguage(texEn1);
56          displayLanguage(texEn2);
57          System.out.println(readLocale(texEn2));
58          System.out.println(readLang(texEn2));
59          System.out.println(readLocale(texEn3));
60
61          System.out.println("filtrer par : anglais");
62          for (String t : filterByLang(texts, "anglais")){
63              System.out.println(t);
64          }
65          System.out.println("textes en francais");
66          for (String t : groupByLang(texts).get("francais")){
67              System.out.println(t);
68          }
69          System.out.println("textes en anglais");
70          for (String t : groupByLang(texts).get("anglais")){
71              System.out.println(t);
72          }
73      }
74  }
```

Question 4 Écrire la méthode `public static Locale readLang(String text)` qui prend un texte dont la première ligne indique sa langue comme indiqué en introduction et renvoie la locale correspondant à son language (on ignore le pays et la variante éventuellement présents).

Question 5 Écrire la méthode `public static Locale readLocale(String text)` qui prend un texte dont la première ligne indique sa langue, et éventuellement un pays et une variante, comme indiqué en introduction et renvoie la locale correspondante.

Question 6 Écrire la méthode `displayLanguage(String text)` qui prend un texte et affiche sur la sortie standard la partie *language* de sa langue, dans la langue du système de l'utilisateur du programme.
Indice : On utilise le terme *display* pour indiquer un affichage adapté ; et par défaut les méthodes correspondantes de Locale ne prenant pas d'argument utilisent la langue du système de l'utilisateur.

Question 7 Écrire la méthode `public static ArrayList<String> filterByLang(ArrayList<String> textCollection, String lang)` qui prend une collection de textes, sous la forme indiquée en introduction, et une langue (dans la langue du système l'utilisateur du programme, telle qu'elle serait affichée par défaut, par exemple "anglais") et renvoie les textes de la collection qui sont dans la langue indiquée, en-tête comprise.

Question 8 Écrire la méthode `public static HashMap<String, ArrayList<String» groupByLang( ArrayList<String> textCollection)` qui prend un ensemble de textes, et crée un tableau associatif (*HashMap*), associant une langue (dans la langue du système l'utilisateur du programme, telle qu'elle serait affichée par défaut, par exemple "anglais") et l'ensemble des textes en cette langue, en-tête comprise.

3 Création de classe (5 points)

Dans cette section on souhaite créer un (premier prototype de) programme permettant la gestion de comptes bancaires. On va pour cela utiliser 3 classes :

- `Identite`, permettant de gérer les identités des propriétaires de compte, le code vous est fourni,
- `Compte`, permettant de gérer les comptes, que vous devrez écrire,
- `Banque`, contenant le programme principal et permettant la création d'identités bancaires et de comptes. Vous devrez en compléter le code.

Code de la classe Identite :

```
1  public class Identite {
2      private String nomProprietaire;
3      private String numId;
4
5      public Identite(String nomProprietaire, String numId) {
6          this.nomProprietaire = nomProprietaire;
7          this.numId = numId;
8      }
9
10     public String getNomProprietaire() {
11         return nomProprietaire;
12     }
13
14     public String getNumId() {
15         return numId;
16     }
17
18     @Override
19     public String toString() {
20         return "" + nomProprietaire + ", ID : " + numId;
21     }
22 }
```

Code de la classe Banque :

```
1  public class Banque {
2
3      private ArrayList<Identite> clients;
4      private ArrayList<Compte> comptes;
5
6      public Banque (){
7          //TODO
8      }
9
10     public void addClient(Identite identite){
11         this.clients.add(identite);
12     }
13
14     public void addCompte(Compte compte){
15         //TODO
16     }
17
18     public static void main(String[] args) {
19         Banque laBanque = new Banque();
20         Identite idDupont = new Identite("Mme. Dupont", "078523243283");
```

```
21          Identite idDurant = new Identite("M. Durant", "000338975493");
22          laBanque.addClient(idDupont);
23          laBanque.addClient(idDurant);
24
25          Compte epargneDupont = new Compte(idDupont, 1500);
26          Compte chequeDupont = new Compte(idDupont, 20);
27          Compte chequeDurant = new Compte(idDurant, 76.25);
28          laBanque.addCompte(epargneDupont);
29          laBanque.addCompte(chequeDupont);
30          laBanque.addCompte(chequeDurant);
31
32          chequeDupont.crediter(1345.76);
33          chequeDupont.debiter(28);
34          if (chequeDupont.debiter(100)){
35              epargneDupont.crediter(100);
36          }
37          chequeDurant.debiter(100);
38          System.out.println("fonds de Durant : " + chequeDurant.getFonds());
39
40      }
41  }
```

Question 9 Écrire une classe Compte, contenant (pour le moment, uniquement) :

- deux attributs : `proprietaire` de type `Identite` et `fonds` de type `double`,
- un constructeur, prenant deux arguments : un de type `Identite` et l'autre de type `double`.

Question 10 Écrire une méthode, pour la classe Compte, `public void crediter(double somme)` qui ajoute la somme indiquée en argument aux fonds du compte.

Question 11 Écrire une méthode, pour la classe Compte, `public boolean debiter(double somme)` qui retire la somme indiquée en argument aux fonds du compte, à condition que le fonds reste positif, et ne fait rien sinon. La méthode renvoie vrai si la somme a pu être débitée.

Question 12 Écrire une méthode, pour la classe Compte, `public double getFonds()`.

Question 13 Écrire le constructeur de la classe Banque.

Question 14 Écrire la méthode `public void addCompte(Compte compte)` de la classe Banque.

java.util

Class ArrayList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the `List` interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.)

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The `add` operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the `LinkedList` implementation.

Each `ArrayList` instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

An application can increase the capacity of an `ArrayList` instance before adding a large number of elements using the `ensureCapacity` operation. This may reduce the amount of incremental reallocation.

Note that this implementation is not synchronized. If multiple threads access an `ArrayList` instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the `Collections.synchronizedList` method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

```
List list = Collections.synchronizedList(new ArrayList(...));
```

The iterators returned by this class's `iterator` and `listIterator` methods are *fail-fast*: if the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw `ConcurrentModificationException` on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the Java Collections Framework.

Since:

1.2

See Also:

`Collection`, `List`, `LinkedList`, `Vector`, Serialized Form

Constructor Summary

Constructors

Constructor and Description
ArrayList() Constructs an empty list with an initial capacity of ten.
ArrayList(**Collection**<? extends **E**> c) Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.
ArrayList(int initialCapacity) Constructs an empty list with the specified initial capacity.

Method Summary

Methods

Modifier and Type	Method and Description
boolean	**add**(**E** e) Appends the specified element to the end of this list.
void	**add**(int index, **E** element) Inserts the specified element at the specified position in this list.
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.
boolean	**addAll**(int index, **Collection**<? extends **E**> c) Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	**clear**() Removes all of the elements from this list.
Object	**clone**() Returns a shallow copy of this ArrayList instance.
boolean	**contains**(**Object** o) Returns true if this list contains the specified element.
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.
E	**remove**(int index) Removes the element at the specified position in this list.
boolean	**remove**(**Object** o) Removes the first occurrence of the specified element from this list, if it is present.
boolean	**removeAll**(**Collection**<?> c) Removes from this list all of its elements that are contained in the specified collection.
protected void	**removeRange**(int fromIndex, int toIndex) Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
boolean	**retainAll**(**Collection**<?> c) Retains only the elements in this list that are contained in the specified collection.
E	**set**(int index, **E** element)

	Replaces the element at the specified position in this list with the specified element.
int	**size**() Returns the number of elements in this list.
<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 class java.util.AbstractList

equals, hashCode

Methods inherited from class java.util.AbstractCollection

containsAll, toString

Methods inherited from class java.lang.Object

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

Methods inherited from interface java.util.List

containsAll, equals, hashCode

compact1, compact2, compact3

java.util

Class HashMap<K,V>

java.lang.Object
    java.util.AbstractMap<K,V>
        java.util.HashMap<K,V>

Type Parameters:

`K` - the type of keys maintained by this map

`V` - the type of mapped values

All Implemented Interfaces:

`Serializable`, `Cloneable`, `Map<K,V>`

Direct Known Subclasses:

`LinkedHashMap`, `PrinterStateReasons`

```
public class HashMap<K,V>
extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable
```

Hash table based implementation of the `Map` interface. This implementation provides all of the optional map operations, and permits `null` values and the `null` key. (The `HashMap` class is roughly equivalent to `Hashtable`, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This implementation provides constant-time performance for the basic operations (`get` and `put`), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the `HashMap` instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

This class is a member of the Java Collections Framework.

Since:

1.2

See Also:

`Object.hashCode()`, `Collection`, `Map`, `TreeMap`, `Hashtable`, Serialized Form

Constructor Summary

Constructors

Constructor and Description

HashMap()
Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).

Method Summary

All Methods Instance Methods Concrete Methods

Modifier and Type	Method and Description
void	**clear**() Removes all of the mappings from this map.
boolean	**containsKey**(**Object** key) Returns true if this map contains a mapping for the specified key.
boolean	**containsValue**(**Object** value) Returns true if this map maps one or more keys to the specified value.
void	**forEach**(**BiConsumer**<? super **K**,? super **V**> action) Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
V	**get**(**Object** key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
V	**getOrDefault**(**Object** key, **V** defaultValue) Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.
boolean	**isEmpty**() Returns true if this map contains no key-value mappings.
Set<**K**>	**keySet**() Returns a **Set** view of the keys contained in this map.
V	**put**(**K** key, **V** value) Associates the specified value with the specified key in this map.
void	**putAll**(**Map**<? extends **K**,? extends **V**> m) Copies all of the mappings from the specified map to this map.
V	**putIfAbsent**(**K** key, **V** value) If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.
V	**remove**(**Object** key) Removes the mapping for the specified key from this map if present.
boolean	**remove**(**Object** key, **Object** value)

	Removes the entry for the specified key only if it is currently mapped to the specified value.
V	**replace**(K key, V value) Replaces the entry for the specified key only if it is currently mapped to some value.
boolean	**replace**(K key, V oldValue, V newValue) Replaces the entry for the specified key only if currently mapped to the specified value.
void	**replaceAll**(BiFunction<? super K,? super V,? extends V> function) Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
int	**size**() Returns the number of key-value mappings in this map.
Collection<V>	**values**() Returns a **Collection** view of the values contained in this map.

Methods inherited from class java.util.AbstractMap

equals, hashCode, toString

Methods inherited from class java.lang.Object

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

Methods inherited from interface java.util.Map

equals, hashCode

java.util

Class Locale

java.lang.Object
    java.util.Locale

All Implemented Interfaces:

Serializable, Cloneable

```
public final class Locale
extends Object
implements Cloneable, Serializable
```

A `Locale` object represents a specific geographical, political, or cultural region. An operation that requires a `Locale` to perform its task is called *locale-sensitive* and uses the `Locale` to tailor information for the user. For example, displaying a number is a locale-sensitive operation— the number should be formatted according to the customs and conventions of the user's native country, region, or culture.

The `Locale` class implements identifiers interchangeable with BCP 47 (IETF BCP 47, "Tags for Identifying Languages"), with support for the LDML (UTS#35, "Unicode Locale Data Markup Language") BCP 47-compatible extensions for locale data exchange.

A `Locale` object logically consists of the fields described below.

language

ISO 639 alpha-2 or alpha-3 language code, or registered language subtags up to 8 alpha letters (for future enhancements). When a language has both an alpha-2 code and an alpha-3 code, the alpha-2 code must be used. You can find a full list of valid language codes in the IANA Language Subtag Registry (search for "Type: language"). The language field is case insensitive, but `Locale` always canonicalizes to lower case.

Well-formed language values have the form `[a-zA-Z]{2,8}`. Note that this is not the the full BCP47 language production, since it excludes extlang. They are not needed since modern three-letter language codes replace them.

Example: "en" (English), "ja" (Japanese), "kok" (Konkani)

country (region)

ISO 3166 alpha-2 country code or UN M.49 numeric-3 area code. You can find a full list of valid country and region codes in the IANA Language Subtag Registry (search for "Type: region"). The country (region) field is case insensitive, but `Locale` always canonicalizes to upper case.

Well-formed country/region values have the form `[a-zA-Z]{2} | [0-9]{3}`

Example: "US" (United States), "FR" (France), "029" (Caribbean)

variant

Any arbitrary value used to indicate a variation of a `Locale`. Where there are two or more variant values each indicating its own semantics, these values should be ordered by importance, with most important first, separated by underscore('_'). The variant field is case sensitive.

Note: IETF BCP 47 places syntactic restrictions on variant subtags. Also BCP 47 subtags are strictly used to indicate additional variations that define a language or its dialects that are not covered by any combinations of language, script

and region subtags. You can find a full list of valid variant codes in the IANA Language Subtag Registry (search for "Type: variant").

However, the variant field in `Locale` has historically been used for any kind of variation, not just language variations. For example, some supported variants available in Java SE Runtime Environments indicate alternative cultural behaviors such as calendar type or number script. In BCP 47 this kind of information, which does not identify the language, is supported by extension subtags or private use subtags.

Well-formed variant values have the form `SUBTAG (('_'|'-') SUBTAG)*` where `SUBTAG = [0-9][0-9a-zA-Z]{3} | [0-9a-zA-Z]{5,8}`. (Note: BCP 47 only uses hyphen ('-') as a delimiter, this is more lenient).

Example: "polyton" (Polytonic Greek), "POSIX"

Note: Although BCP 47 requires field values to be registered in the IANA Language Subtag Registry, the `Locale` class does not provide any validation features. The `Builder` only checks if an individual field satisfies the syntactic requirement (is well-formed), but does not validate the value itself. See `Locale.Builder` for details.

Use of Locale

Once you've created a `Locale` you can query it for information about itself. Use `getCountry` to get the country (or region) code and `getLanguage` to get the language code. You can use `getDisplayCountry` to get the name of the country suitable for displaying to the user. Similarly, you can use `getDisplayLanguage` to get the name of the language suitable for displaying to the user. Interestingly, the `getDisplayXXX` methods are themselves locale-sensitive and have two versions: one that uses the default locale and one that uses the locale specified as an argument.

The Java Platform provides a number of classes that perform locale-sensitive operations. For example, the `NumberFormat` class formats numbers, currency, and percentages in a locale-sensitive manner. Classes such as `NumberFormat` have several convenience methods for creating a default object of that type. For example, the `NumberFormat` class provides these three convenience methods for creating a default `NumberFormat` object:

```
NumberFormat.getInstance()
NumberFormat.getCurrencyInstance()
NumberFormat.getPercentInstance()
```

Each of these methods has two variants; one with an explicit locale and one without; the latter uses the default locale:

```
NumberFormat.getInstance(myLocale)
NumberFormat.getCurrencyInstance(myLocale)
NumberFormat.getPercentInstance(myLocale)
```

A `Locale` is the mechanism for identifying the kind of object (`NumberFormat`) that you would like to get. The locale is **just** a mechanism for identifying objects, **not** a container for the objects themselves.

Since:

1.1

See Also:

`Locale.Builder`, `ResourceBundle`, `Format`, `NumberFormat`, `Collator`, Serialized Form

Constructor Summary

Constructors

Constructor and Description
`Locale``(String language)` Construct a locale from a language code.
`Locale``(String language, String country)` Construct a locale from language and country.
`Locale``(String language, String country, String variant)` Construct a locale from language, country and variant.

Method Summary

Methods

Modifier and Type	Method and Description
Object	**clone**() Overrides Cloneable.
boolean	**equals**(**Object** obj) Returns true if this Locale is equal to another object.
static **Locale**	**forLanguageTag**(**String** languageTag) Returns a locale for the specified IETF BCP 47 language tag string.
static **Locale**[]	**getAvailableLocales**() Returns an array of all installed locales.
String	**getCountry**() Returns the country/region code for this locale, which should either be the empty string, an uppercase ISO 3166 2-letter code, or a UN M.49 3-digit code.
static **Locale**	**getDefault**() Gets the current value of the default locale for this instance of the Java Virtual Machine.
static **Locale**	**getDefault**(**Locale.Category** category) Gets the current value of the default locale for the specified Category for this instance of the Java Virtual Machine.
String	**getDisplayCountry**() Returns a name for the locale's country that is appropriate for display to the user.
String	**getDisplayCountry**(**Locale** inLocale) Returns a name for the locale's country that is appropriate for display to the user.
String	**getDisplayLanguage**() Returns a name for the locale's language that is appropriate for display to the user.
String	**getDisplayLanguage**(**Locale** inLocale) Returns a name for the locale's language that is appropriate for display to the user.
String	**getDisplayVariant**() Returns a name for the locale's variant code that is appropriate for display to the user.
String	**getDisplayVariant**(**Locale** inLocale) Returns a name for the locale's variant code that is appropriate for display to the user.
String	**getLanguage**() Returns the language code of this Locale.
String	**getScript**() Returns the script for this locale, which should either be the empty string or an ISO 15924 4-letter script code.
String	**getVariant**() Returns the variant code for this locale.
int	**hashCode**() Override hashCode.
static void	**setDefault**(**Locale.Category** category, **Locale** newLocale) Sets the default locale for the specified Category for this instance of the Java Virtual Machine.
static void	**setDefault**(**Locale** newLocale) Sets the default locale for this instance of the Java Virtual Machine.
String	**toLanguageTag**() Returns a well-formed IETF BCP 47 language tag representing this locale.
String	**toString**() Returns a string representation of this Locale object, consisting of language, country, variant, script, and extensions as below:

Methods inherited from class java.lang.Object

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

Constructor Detail

Locale

```
public Locale(String language,
      String country,
      String variant)
```

Construct a locale from language, country and variant. This constructor normalizes the language value to lowercase and the country value to uppercase.

Note:

- ISO 639 is not a stable standard; some of the language codes it defines (specifically "iw", "ji", and "in") have changed. This constructor accepts both the old codes ("iw", "ji", and "in") and the new codes ("he", "yi", and "id"), but all other API on Locale will return only the OLD codes.
- For backward compatibility reasons, this constructor does not make any syntactic checks on the input.
- The two cases ("ja", "JP", "JP") and ("th", "TH", "TH") are handled specially, see Special Cases for more information.

Parameters:

`language` - An ISO 639 alpha-2 or alpha-3 language code, or a language subtag up to 8 characters in length. See the `Locale` class description about valid language values.

`country` - An ISO 3166 alpha-2 country code or a UN M.49 numeric-3 area code. See the `Locale` class description about valid country values.

`variant` - Any arbitrary value used to indicate a variation of a `Locale`. See the `Locale` class description for the details.

Throws:

`NullPointerException` - thrown if any argument is null.

Locale

```
public Locale(String language,
      String country)
```

Construct a locale from language and country. This constructor normalizes the language value to lowercase and the country value to uppercase.

Note:

- ISO 639 is not a stable standard; some of the language codes it defines (specifically "iw", "ji", and "in") have changed. This constructor accepts both the old codes ("iw", "ji", and "in") and the new codes ("he", "yi", and "id"), but all other API on Locale will return only the OLD codes.
- For backward compatibility reasons, this constructor does not make any syntactic checks on the input.

Parameters:

`language` - An ISO 639 alpha-2 or alpha-3 language code, or a language subtag up to 8 characters in length. See the `Locale` class description about valid language values.

`country` - An ISO 3166 alpha-2 country code or a UN M.49 numeric-3 area code. See the `Locale` class description about valid country values.

Throws:

`NullPointerException` - thrown if either argument is null.

Locale

```
public Locale(String language)
```

Construct a locale from a language code. This constructor normalizes the language value to lowercase.

Note:

- ISO 639 is not a stable standard; some of the language codes it defines (specifically "iw", "ji", and "in") have changed. This constructor accepts both the old codes ("iw", "ji", and "in") and the new codes ("he", "yi", and "id"), but all other API on Locale will return only the OLD codes.
- For backward compatibility reasons, this constructor does not make any syntactic checks on the input.

Parameters:

`language` - An ISO 639 alpha-2 or alpha-3 language code, or a language subtag up to 8 characters in length. See the `Locale` class description about valid language values.

Throws:

`NullPointerException` - thrown if argument is null.

Since:

1.4

Method Detail

getDefault

```
public static Locale getDefault()
```

Gets the current value of the default locale for this instance of the Java Virtual Machine.

The Java Virtual Machine sets the default locale during startup based on the host environment. It is used by many locale-sensitive methods if no locale is explicitly specified. It can be changed using the `setDefault` method.

Returns:

the default locale for this instance of the Java Virtual Machine

getDefault

```
public static Locale getDefault(Locale.Category category)
```

Gets the current value of the default locale for the specified Category for this instance of the Java Virtual Machine.

The Java Virtual Machine sets the default locale during startup based on the host environment. It is used by many locale-sensitive methods if no locale is explicitly specified. It can be changed using the setDefault(Locale.Category, Locale) method.

Parameters:

`category` - - the specified category to get the default locale

Returns:

the default locale for the specified Category for this instance of the Java Virtual Machine

Throws:

`NullPointerException` - - if category is null

Since:

1.7

See Also:

`setDefault(Locale.Category, Locale)`

setDefault

```
public static void setDefault(Locale newLocale)
```

Sets the default locale for this instance of the Java Virtual Machine. This does not affect the host locale.

If there is a security manager, its `checkPermission` method is called with a `PropertyPermission("user.language", "write")` permission before the default locale is changed.

The Java Virtual Machine sets the default locale during startup based on the host environment. It is used by many locale-sensitive methods if no locale is explicitly specified.

Since changing the default locale may affect many different areas of functionality, this method should only be used if the caller is prepared to reinitialize locale-sensitive code running within the same Java Virtual Machine.

By setting the default locale with this method, all of the default locales for each Category are also set to the specified default locale.

Parameters:

`newLocale` - the new default locale

Throws:

`SecurityException` - if a security manager exists and its `checkPermission` method doesn't allow the operation.

`NullPointerException` - if `newLocale` is null

See Also:

`SecurityManager.checkPermission(java.security.Permission)`, `PropertyPermission`

setDefault

```
public static void setDefault(Locale.Category category,
              Locale newLocale)
```

Sets the default locale for the specified Category for this instance of the Java Virtual Machine. This does not affect the host locale.

If there is a security manager, its checkPermission method is called with a PropertyPermission("user.language", "write") permission before the default locale is changed.

The Java Virtual Machine sets the default locale during startup based on the host environment. It is used by many locale-sensitive methods if no locale is explicitly specified.

Since changing the default locale may affect many different areas of functionality, this method should only be used if the caller is prepared to reinitialize locale-sensitive code running within the same Java Virtual Machine.

Parameters:

`category` - - the specified category to set the default locale

`newLocale` - - the new default locale

Throws:

`SecurityException` - - if a security manager exists and its checkPermission method doesn't allow the operation.

`NullPointerException` - - if category and/or newLocale is null

Since:

1.7

See Also:

`SecurityManager.checkPermission(java.security.Permission)`, `PropertyPermission`, `getDefault(Locale.Category)`

getAvailableLocales

```
public static Locale[] getAvailableLocales()
```

Returns an array of all installed locales. The returned array represents the union of locales supported by the Java runtime environment and by installed `LocaleServiceProvider` implementations. It must contain at least a `Locale`

instance equal to `Locale.US`.

Returns:

An array of installed locales.

getLanguage

```
public String getLanguage()
```

Returns the language code of this Locale.

Note: ISO 639 is not a stable standard— some languages' codes have changed. Locale's constructor recognizes both the new and the old codes for the languages whose codes have changed, but this function always returns the old code. If you want to check for a specific language whose code has changed, don't do

```
if (locale.getLanguage().equals("he")) // BAD!
   ...
```

Instead, do

```
if (locale.getLanguage().equals(new Locale("he").getLanguage()))
   ...
```

Returns:

The language code, or the empty string if none is defined.

See Also:

`getDisplayLanguage()`

getScript

```
public String getScript()
```

Returns the script for this locale, which should either be the empty string or an ISO 15924 4-letter script code. The first letter is uppercase and the rest are lowercase, for example, 'Latn', 'Cyrl'.

Returns:

The script code, or the empty string if none is defined.

Since:

1.7

See Also:

`getDisplayScript()`

getCountry

```
public String getCountry()
```

Returns the country/region code for this locale, which should either be the empty string, an uppercase ISO 3166 2-letter code, or a UN M.49 3-digit code.

Returns:

The country/region code, or the empty string if none is defined.

See Also:

`getDisplayCountry()`

getVariant

```
public String getVariant()
```

Returns the variant code for this locale.

Returns:

The variant code, or the empty string if none is defined.

See Also:

`getDisplayVariant()`

toString

```
public final String toString()
```

Returns a string representation of this `Locale` object, consisting of language, country, variant, script, and extensions as below:

language + "_" + country + "_" + (variant + "_#" | "#") + script + "-" + extensions

Language is always lower case, country is always upper case, script is always title case, and extensions are always lower case. Extensions and private use subtags will be in canonical order as explained in `toLanguageTag()`.

When the locale has neither script nor extensions, the result is the same as in Java 6 and prior.

If both the language and country fields are missing, this function will return the empty string, even if the variant, script, or extensions field is present (you can't have a locale with just a variant, the variant must accompany a well-formed language or country code).

If script or extensions are present and variant is missing, no underscore is added before the "#".

This behavior is designed to support debugging and to be compatible with previous uses of `toString` that expected language, country, and variant fields only. To represent a Locale as a String for interchange purposes, use `toLanguageTag()`.

Examples:

- `en`
- `de_DE`
- `_GB`
- `en_US_WIN`
- `de__POSIX`
- `zh_CN_#Hans`
- `zh_TW_#Hant-x-java`
- `th_TH_TH_#u-nu-thai`

Overrides:

`toString` in class `Object`

Returns:

A string representation of the Locale, for debugging.

See Also:

`getDisplayName()`, `toLanguageTag()`

toLanguageTag

```
public String toLanguageTag()
```

Returns a well-formed IETF BCP 47 language tag representing this locale.

If this `Locale` has a language, country, or variant that does not satisfy the IETF BCP 47 language tag syntax requirements, this method handles these fields as described below:

Language: If language is empty, or not well-formed (for example "a" or "e2"), it will be emitted as "und" (Undetermined).

Country: If country is not well-formed (for example "12" or "USA"), it will be omitted.

Variant: If variant **is** well-formed, each sub-segment (delimited by '-' or '_') is emitted as a subtag. Otherwise:

- if all sub-segments match `[0-9a-zA-Z]{1,8}` (for example "WIN" or "Oracle_JDK_Standard_Edition"), the first ill-formed sub-segment and all following will be appended to the private use subtag. The first appended subtag will be "lvariant", followed by the sub-segments in order, separated by hyphen. For example, "x-lvariant-WIN", "Oracle-x-lvariant-JDK-Standard-Edition".
- if any sub-segment does not match `[0-9a-zA-Z]{1,8}`, the variant will be truncated and the problematic sub-segment and all following sub-segments will be omitted. If the remainder is non-empty, it will be emitted as a private use subtag as above (even if the remainder turns out to be well-formed). For example, "Solaris_isjustthecoolestthing" is emitted as "x-lvariant-Solaris", not as "solaris".

Special Conversions: Java supports some old locale representations, including deprecated ISO language codes, for compatibility. This method performs the following conversions:

- Deprecated ISO language codes "iw", "ji", and "in" are converted to "he", "yi", and "id", respectively.
- A locale with language "no", country "NO", and variant "NY", representing Norwegian Nynorsk (Norway), is converted to a language tag "nn-NO".

Note: Although the language tag created by this method is well-formed (satisfies the syntax requirements defined by the IETF BCP 47 specification), it is not necessarily a valid BCP 47 language tag. For example,

```
new Locale("xx", "YY").toLanguageTag();
```

will return "xx-YY", but the language subtag "xx" and the region subtag "YY" are invalid because they are not registered in the IANA Language Subtag Registry.

Returns:

a BCP47 language tag representing the locale

Since:

1.7

See Also:

`forLanguageTag(String)`

forLanguageTag

```
public static Locale forLanguageTag(String languageTag)
```

Returns a locale for the specified IETF BCP 47 language tag string.

If the specified language tag contains any ill-formed subtags, the first such subtag and all following subtags are ignored. Compare to `Locale.Builder.setLanguageTag(java.lang.String)` which throws an exception in this case.

The following **conversions** are performed:

- The language code "und" is mapped to language "".
- The language codes "he", "yi", and "id" are mapped to "iw", "ji", and "in" respectively. (This is the same canonicalization that's done in Locale's constructors.)
- The portion of a private use subtag prefixed by "lvariant", if any, is removed and appended to the variant field in the result locale (without case normalization). If it is then empty, the private use subtag is discarded:

```
Locale loc;
loc = Locale.forLanguageTag("en-US-x-lvariant-POSIX");
loc.getVariant(); // returns "POSIX"
loc.getExtension('x'); // returns null

loc = Locale.forLanguageTag("de-POSIX-x-URP-lvariant-Abc-Def");
loc.getVariant(); // returns "POSIX_Abc_Def"
loc.getExtension('x'); // returns "urp"
```

- When the languageTag argument contains an extlang subtag, the first such subtag is used as the language, and the primary language subtag and other extlang subtags are ignored:

```
Locale.forLanguageTag("ar-aao").getLanguage(); // returns "aao"
Locale.forLanguageTag("en-abc-def-us").toString(); // returns "abc_US"
```

- Case is normalized except for variant tags, which are left unchanged. Language is normalized to lower case, script to title case, country to upper case, and extensions to lower case.
- If, after processing, the locale would exactly match either ja_JP_JP or th_TH_TH with no extensions, the appropriate extensions are added as though the constructor had been called:

```
Locale.forLanguageTag("ja-JP-x-lvariant-JP").toLanguageTag();
// returns "ja-JP-u-ca-japanese-x-lvariant-JP"
Locale.forLanguageTag("th-TH-x-lvariant-TH").toLanguageTag();
// returns "th-TH-u-nu-thai-x-lvariant-TH"
```

This implements the 'Language-Tag' production of BCP47, and so supports grandfathered (regular and irregular) as well as private use language tags. Stand alone private use tags are represented as empty language and extension 'x-whatever', and grandfathered tags are converted to their canonical replacements where they exist.

Grandfathered tags with canonical replacements are as follows:

grandfathered tag	modern replacement
art-lojban	jbo
i-ami	ami
i-bnn	bnn
i-hak	hak
i-klingon	tlh
i-lux	lb
i-navajo	nv
i-pwn	pwn
i-tao	tao
i-tay	tay
i-tsu	tsu
no-bok	nb
no-nyn	nn
sgn-BE-FR	sfb
sgn-BE-NL	vgt
sgn-CH-DE	sgg
zh-guoyu	cmn
zh-hakka	hak
zh-min-nan	nan
zh-xiang	hsn

Grandfathered tags with no modern replacement will be converted as follows:

grandfathered tag	converts to
cel-gaulish	xtg-x-cel-gaulish
en-GB-oed	en-GB-x-oed
i-default	en-x-i-default
i-enochian	und-x-i-enochian
i-mingo	see-x-i-mingo
zh-min	nan-x-zh-min

For a list of all grandfathered tags, see the IANA Language Subtag Registry (search for "Type: grandfathered").

Note: there is no guarantee that `toLanguageTag` and `forLanguageTag` will round-trip.

Parameters:

`languageTag` - the language tag

Returns:

The locale that best represents the language tag.

Throws:

`NullPointerException` - if `languageTag` is `null`

Since:

1.7

See Also:

`toLanguageTag()`, `Locale.Builder.setLanguageTag(String)`

getDisplayLanguage

```
public final String getDisplayLanguage()
```

Returns a name for the locale's language that is appropriate for display to the user. If possible, the name returned will be localized for the default locale. For example, if the locale is fr_FR and the default locale is en_US, getDisplayLanguage() will return "French"; if the locale is en_US and the default locale is fr_FR, getDisplayLanguage() will return "anglais". If the name returned cannot be localized for the default locale, (say, we don't have a Japanese name for Croatian), this function falls back on the English name, and uses the ISO code as a last-resort value. If the locale doesn't specify a language, this function returns the empty string.

getDisplayLanguage

```
public String getDisplayLanguage(Locale inLocale)
```

Returns a name for the locale's language that is appropriate for display to the user. If possible, the name returned will be localized according to inLocale. For example, if the locale is fr_FR and inLocale is en_US, getDisplayLanguage() will return "French"; if the locale is en_US and inLocale is fr_FR, getDisplayLanguage() will return "anglais". If the name returned cannot be localized according to inLocale, (say, we don't have a Japanese name for Croatian), this function falls back on the English name, and finally on the ISO code as a last-resort value. If the locale doesn't specify a language, this function returns the empty string.

Throws:

`NullPointerException` - if `inLocale` is `null`

getDisplayCountry

```
public final String getDisplayCountry()
```

Returns a name for the locale's country that is appropriate for display to the user. If possible, the name returned will be localized for the default locale. For example, if the locale is fr_FR and the default locale is en_US, getDisplayCountry() will return "France"; if the locale is en_US and the default locale is fr_FR, getDisplayCountry() will return "Etats-Unis". If the name returned cannot be localized for the default locale, (say, we don't have a Japanese name for Croatia), this function falls back on the English name, and uses the ISO code as a last-resort value. If the locale doesn't specify a country, this function returns the empty string.

getDisplayCountry

```
public String getDisplayCountry(Locale inLocale)
```

Returns a name for the locale's country that is appropriate for display to the user. If possible, the name returned will be localized according to inLocale. For example, if the locale is fr_FR and inLocale is en_US, getDisplayCountry() will return "France"; if the locale is en_US and inLocale is fr_FR, getDisplayCountry() will return "Etats-Unis". If the name returned cannot be localized according to inLocale. (say, we don't have a Japanese name for Croatia), this function falls back on the English name, and finally on the ISO code as a last-resort value. If the locale doesn't specify a country, this function returns the empty string.

Throws:

`NullPointerException` - if `inLocale` is `null`

getDisplayVariant

```
public final String getDisplayVariant()
```

Returns a name for the locale's variant code that is appropriate for display to the user. If possible, the name will be localized for the default locale. If the locale doesn't specify a variant code, this function returns the empty string.

getDisplayVariant

```
public String getDisplayVariant(Locale inLocale)
```

Returns a name for the locale's variant code that is appropriate for display to the user. If possible, the name will be localized for inLocale. If the locale doesn't specify a variant code, this function returns the empty string.

Throws:

`NullPointerException` - if `inLocale` is `null`

clone

```
public Object clone()
```

Overrides Cloneable.

Overrides:

`clone` in class `Object`

Returns:

a clone of this instance.

See Also:

`Cloneable`

hashCode

```
public int hashCode()
```

Override hashCode. Since Locales are often used in hashtables, caches the value for speed.

Overrides:

`hashCode` in class `Object`

Returns:

a hash code value for this object.

See Also:

`Object.equals(java.lang.Object)`, `System.identityHashCode(java.lang.Object)`

equals

```
public boolean equals(Object obj)
```

Returns true if this Locale is equal to another object. A Locale is deemed equal to another Locale with identical language, script, country, variant and extensions, and unequal to all other objects.

Overrides:

`equals` in class `Object`

Parameters:

`obj` - the reference object with which to compare.

Returns:

true if this Locale is equal to the specified object.

See Also:

`Object.hashCode()`, `HashMap`