## Lezione 17
# Le classi di utilità nel pacchetto java.util

Il package *java.util* contiene numerose classi di utilità generica. Vengono usate da molti altri pacchetti della libreria di Java, primo fra tutti *java.lang*. Dunque, è importante conoscere come le classi contenute in *java.util* possano essere manipolate. Lo stesso sviluppatore è inoltre libero di utilizzare queste classi all'interno delle proprie applicazioni. Dentro *java.util* si trovano strumenti di vario tipo: dalle strutture dati di utilizzo più comune agli strumenti per la gestione delle date, dalle utilità per l'analisi delle stringhe alle classi dedicate alla gestione delle operazioni temporizzate.

## 17.1 - Gestione delle date

Una delle classi di *java.util* più nota è ***Date***, utile per rappresentare una data ed un'ora.

| **Costruttori pubblici** | |
|---|---|
| `Date()` | Costruisce un oggetto *Date* che incapsula la data e l'ora correnti. |
| `Date(long t)` | Costruisce un oggetto *Date* che incapsula la data e l'ora espressi dall'argomento *t*. L'argomento è un *long* che riporta i millisecondi trascorsi dal 1 Gennaio 1970 sino alla data rappresentata. |
| **Metodi pubblici** | |
| `boolean after(Date d)` | Restituisce *true* se la data di invocazione è successiva alla data *d*. |
| `boolean before(Date d)` | Restituisce *true* se la data di invocazione è precedente alla data *d*. |
| `Object clone()` | Clona l'oggetto. *Date* implementa l'interfaccia *Cloneable*. |
| `int compareTo(Date d)` | Compara la data di invocazione con *d*. Restituisce 0 se le due date sono uguali, un valore negativo se la data di invocazione precede la data *d* o un valore positivo se la data di invocazione è successiva alla data *d*. |
| `long getTime()` | Restituisce la data rappresentata dall'oggetto sotto forma di valore *long*, che esprime i millisecondi trascorsi dal 1 Gennaio 1970 sino alla data rappresentata. |
| `void setTime(long t)` | Imposta la data rappresentata con un argomento di tipo *long*, che esprime i millisecondi trascorsi dal 1 Gennaio 1970 sino alla data rappresentata. |

**Tabella 17.1**
I membri resi disponibili dalla classe *Date*.

Si cominci da un semplicissimo esempio:

```
import java.util.*;

public class DateTest {
```

```
  public static void main(String[] args) {
    // Creo un oggetto Date con data ed ora correnti.
    Date d = new Date();
    // Chiedo la rappresentazione in stringa.
    System.out.println(d.toString());
    // Recuperlo il valore long associato.
    long l = d.getTime();
    // Calcolo il numero di millisecondi in un giorno.
    int giorno = 1000 * 60 * 60 * 24;
    // Tolgo una settimana al valore l.
    l -= (giorno * 7);
    // Reimposto la data con il nuovo valore.
    d.setTime(l);
    // Chiedo nuovamente la rappresentazione in stringa.
    System.out.println(d.toString());
  }

}
```

I commenti introdotti nel codice illustrano il dettaglio di ogni operazione effettuata. L'output mostrerà due date, una vicina a quella corrente, l'altra più vecchia di una settimana:

```
Wed Mar 05 22:37:49 CET 2003
Wed Feb 26 22:37:49 CET 2003
```

La classe astratta ***Calendar*** fornisce gli strumenti utili per una gestione molto più dettagliata ed accurata delle date. Da essa deriva ***GregorianCalendar***, che può essere liberamente usata in ogni applicazione. La seguente tabella riporta le funzionalità più significative della classe.

| **Costruttori pubblici** | |
|---|---|
| `GregorianCalendar()` | Costruisce un *GregorianCalendar* che rappresenta la data e l'ora correnti. |
| `GregorianCalendar(int year, int month, int date)` | Costruisce un *GregorianCalendar* che rappresenta la data espressa mediante gli argomenti forniti. |
| `GregorianCalendar(int year, int month, int date, int hour, int minute)` | Costruisce un *GregorianCalendar* che rappresenta la data e l'ora espressi mediante gli argomenti forniti. |
| **Metodi pubblici** | |
| `boolean after(Object o)` | Restituisce *true* se la data rappresentata è successiva alla data espressa dall'oggetto o, che deve essere istanza di *Calendar*. |
| `boolean before(Object o)` | Restituisce *true* se la data rappresentata è precedente alla data espressa dall'oggetto o, che deve essere istanza di *Calendar*. |
| `int get(int field)` | Recupera il valore di uno dei campi della data rappresentata. L'argomento specifica il campo di interesse. |

| | |
|---|---|
| `Date getTime()` | Restituisce un oggetto *Date* che rappresenta la data incapsulata dall'oggetto di invocazione. |
| `long getTimeInMillis()` | Restituisce la data rappresentata dall'oggetto sotto forma di valore *long*, che esprime i millisecondi trascorsi dal 1 Gennaio 1970 sino alla data rappresentata. |
| `void set(int field, int value)` | Imposta su *value* il valore del campo rappresentato dall'intero *field*. |
| `void setTime(Date d)` | Imposta la data rappresentata prelevando il suo valore all'argomento *d*. |
| `void setTimeInMillis(long l)` | Imposta la data rappresentata mediante un argomento di tipo *long*, che esprime i millisecondi trascorsi dal 1 Gennaio 1970 sino alla data rappresentata. |

**Tabella 17.2**
Le funzionalità più importanti della classe *GregorianCalendar*.

Inoltre, all'interno di *Calendar*, è definita una lunga serie di costanti statiche di tipo *int*, che possono essere usate per leggere ed impostare i campi di un oggetto *GregorianCalendar*. Alcune di esse identificano i singoli campi dell'oggetto:

- ***AM_PM***. Il campo che informa se l'ora espressa è prima o dopo mezzogiorno.
- ***DAY_OF_MONTH***. Il campo che riporta il giorno del mese.
- ***DAY_OF_WEEK***. Il campo che riporta il giorno della settimana.
- ***DAY_OF_YEAR***. Il campo che riporta il giorno dell'anno.
- ***HOUR***. Il campo che riporta l'ora del mattino o del pomeriggio, a seconda del contenuto del campo AM_PM.
- ***HOUR_OF_DAY***. Il campo che riporta l'ora del giorno, in un intervallo tra 0 e 23.
- ***MILLISECOND***. Il campo che riporta i millisecondi.
- ***MINUTE***. Il campo che riporta i minuti.
- ***MONTH***. Il campo che riporta il mese.
- ***SECOND***. Il campo che riporta i secondi.
- ***WEEK_OF_MONTH***. Il campo che riporta la settimana del mese.
- ***WEEK_OF_YEAR***. Il campo che riporta la settimana dell'anno.
- ***YEAR***. Il campo che riporta l'anno.

Alle costanti che identificano i singoli campi di un oggetto *Calendar* o *GregorianCalendar*, sono poi affiancate altre costanti (sempre di tipo *int*), che aiutano ad interpretare i valori degli stessi:

- ***AM*** e ***PM***. Esprimono il possibile contenuto del campo *AM_PM*.
- ***JANUARY***, ***FEBRUARY***, ***MARCH***, ***APRIL***, ***MAY***, ***JUNE***, ***JULY***, ***AUGUST***, ***SEPTEMBER***, ***OCTOBER***, ***NOVEMBER*** e ***DECEMBER***. Esprimono il possibile contenuto del campo *MONTH*.
- ***SUNDAY***, ***MONDAY***, ***TUESDAY***, ***WEDNESDAY***, ***THURSDAY***, ***FRIDAY*** e

***SATURDAY***. Esprimono il possibile contenuto del campo ***DAY_IN_WEEK***.

Tutte le costanti sinora utilizzate trovano applicazione nell'impiego dei metodi *get()* e *set()*, che permettono di leggere e modificare la data rappresentata:

```
import java.util.*;

public class CalendarTest {

  public static void main(String[] args) {
    GregorianCalendar c = new GregorianCalendar();
    System.out.print("Oggi è ");
    switch (c.get(Calendar.DAY_OF_WEEK)) {
    case Calendar.SUNDAY:
      System.out.print("Domenica ");
      break;
    case Calendar.MONDAY:
      System.out.print("Lunedì ");
      break;
    case Calendar.TUESDAY:
      System.out.print("Martedì ");
      break;
    case Calendar.WEDNESDAY:
      System.out.print("Mercoledì ");
      break;
    case Calendar.THURSDAY:
      System.out.print("Giovedì ");
      break;
    case Calendar.FRIDAY:
      System.out.print("Venerdì ");
      break;
    case Calendar.SATURDAY:
      System.out.print("Sabato ");
      break;
    }
    System.out.print(c.get(Calendar.DAY_OF_MONTH) + " ");
    switch (c.get(Calendar.MONTH)) {
    case Calendar.JANUARY:
      System.out.print("Gennaio ");
      break;
    case Calendar.FEBRUARY:
      System.out.print("Febbraio ");
      break;
    case Calendar.MARCH:
      System.out.print("Marzo ");
      break;
    case Calendar.APRIL:
      System.out.print("Aprile ");
      break;
    case Calendar.MAY:
      System.out.print("Maggio ");
      break;
    case Calendar.JUNE:
      System.out.print("Giugno ");
      break;
    case Calendar.JULY:
      System.out.print("Luglio ");
      break;
    case Calendar.AUGUST:
      System.out.print("Agosto ");
```

```
      break;
    case Calendar.SEPTEMBER:
      System.out.print("Settembre ");
      break;
    case Calendar.OCTOBER:
      System.out.print("Ottobre ");
      break;
    case Calendar.NOVEMBER:
      System.out.print("Novembre ");
      break;
    case Calendar.DECEMBER:
      System.out.print("Dicembre ");
      break;
    }
    System.out.println(c.get(Calendar.YEAR) + ".");
    System.out.print("Sono le ore ");
    System.out.print(c.get(Calendar.HOUR) + " e ");
    System.out.print(c.get(Calendar.MINUTE) + " ");
    switch (c.get(Calendar.AM_PM)) {
    case Calendar.AM:
      System.out.println("del mattino (AM).");
      break;
    case Calendar.PM:
      System.out.println("del pomeriggio (PM).");
      break;
    }
    System.out.print(
      "Senza la notazione AM/PM sono semplicemente le "
    );
    System.out.print(c.get(Calendar.HOUR_OF_DAY) + " e ");
    System.out.println(c.get(Calendar.MINUTE) + ".");
    System.out.print("Sono passati ");
    System.out.print(c.get(Calendar.DAY_OF_YEAR) + " ");
    System.out.println("giorni dall'inizio dell'anno.");
    System.out.print("Questa è la settimana dell'anno numero ");
    System.out.println(c.get(Calendar.WEEK_OF_YEAR) + ".");
    System.out.print("Questa è la settimana del mese numero ");
    System.out.println(c.get(Calendar.WEEK_OF_MONTH) + ".");
  }

}
```

L'output prodotto sarà qualcosa del tipo:

```
Oggi è Martedì 19 Luglio 2005.
Sono le ore 9 e 30 del pomeriggio (PM).
Senza la notazione AM/PM sono semplicemente le 21 e 30
Sono passati 200 giorni dall'inizio dell'anno.
Questa è la settimana dell'anno numero 29.
Questa è la settimana del mese numero 3.
```

Ecco invece un'applicazione del metodo *set()*:

```
import java.util.*;

public class CalendarTest2 {

  public static void main(String[] args) {
    GregorianCalendar c = new GregorianCalendar();
    c.set(Calendar.YEAR, 2548);
```

```
    c.set(Calendar.MONTH, Calendar.DECEMBER);
    c.set(Calendar.DAY_OF_MONTH, 25);
    System.out.print("Nel 2548 Natale cadrà di ");
    switch (c.get(Calendar.DAY_OF_WEEK)) {
    case Calendar.SUNDAY:
      System.out.println("Domenica.");
      break;
    case Calendar.MONDAY:
      System.out.println("Lunedì.");
      break;
    case Calendar.TUESDAY:
      System.out.println("Martedì.");
      break;
    case Calendar.WEDNESDAY:
      System.out.println("Mercoledì.");
      break;
    case Calendar.THURSDAY:
      System.out.println("Giovedì.");
      break;
    case Calendar.FRIDAY:
      System.out.println("Venerdì.");
      break;
    case Calendar.SATURDAY:
      System.out.println("Sabato.");
      break;
    }
  }

}
```

L'output prodotto è:

```
Nel 2548 Natale cadrà di Mercoledì.
```

### 17.2 - L'interfaccia Enumeration

L'interfaccia *Enumeration* descrive due metodi attraverso i quali è possibile enumerare (mettere in ordine ed esaminare uno alla volta) gli elementi di un qualsiasi insieme. Giacché in *java.util* trovano posto numerose classi che gestiscono insiemi di elementi, è stato coerente definire l'interfaccia in tale posizione. Spesso gli insiemi di *java.util* hanno un metodo *elements()*, che restituisce l'enumerazione dei loro elementi. Quindi, è importante conoscere il funzionamento dell'interfaccia *Enumeration*. Stabilisce due metodi:

- ***boolean hasMoreElements()***. Restituisce *true* fin quando ci sono elementi da esaminare.
- ***Object nextElement()***. Restituisce l'elemento successivo.

Grazie a questa coppia, diventa possibile "sfogliare" con gran facilità gli elementi di un insieme. La forma tipica è la seguente:

```
Enumeration e = insieme.elements();
while (e.hasMoreElements()) {
  Object element = e.nextElement();
  // Fai qualcosa con l'elemento corrente.
}
```

### 17.3 - Liste

La classe astratta ***AbstractList***, che estende ***AbstractCollection***, definisce un modello di

comportamento generico per la costruzione di liste. Sostanzialmente, le liste sono come degli array dinamici, che possono variare "al volo" le loro dimensioni, per fare spazio a più o a meno elementi. Esistono diversi tipi di liste, secondo la loro implementazione. Dentro *java.util* trovano posto due classi derivate da *AbstractList*, chiamate ***Vector*** e ***ArrayList***. Entrambe forniscono un'implementazione completa e funzionante del concetto di lista.

La classe ***Vector*** dispone dei seguenti membri:

| **Costruttori pubblici** | |
|---|---|
| `Vector()` | Costruisce una lista di tipo *Vector* inizialmente vuota. |
| **Metodi pubblici** | |
| `void add(int i, Object o)` | Aggiunge l'oggetto o alla lista, disponendolo alla posizione *i*. Se la posizione è già occupata, l'elemento corrispondente e tutti i suoi successivi verranno avanzati di un posto. |
| `boolean add(Object o)` | Aggiunge l'oggetto o in coda alla lista. |
| `void clear()` | Ripulisce la lista, eliminando tutti i suoi elementi. |
| `boolean contains(Object o)` | Restituisce *true* se la lista contiene l'oggetto *o*. |
| `Enumeration elements()` | Restituisce l'enumerazione di tutti gli elementi della lista. |
| `Object firstElement()` | Restituisce il primo elemento della lista. |
| `Object get(int i)` | Restituisce l'elemento alla posizione *i*. |
| `int indexOf(Object o)` | Restituisce l'indice dell'elemento *o*, oppure -1 se l'elemento non compare nella lista. |
| `int indexOf(Object o, int s)` | Come il precedente, solo che inizia la ricerca a partire dalla posizione *s*. |
| `boolean isEmpty()` | Restituisce *true* se la lista è vuota. |
| `Object lastElement()` | Restituisce l'ultimo elemento della lista. |
| `Object remove(int i)` | Individua l'elemento alla posizione *i*, lo rimuove dalla lista e lo restituisce al codice chiamante. Tutti gli elementi successivi saranno arretrati di una posizione. |
| `boolean remove(Object o)` | Rimuove la prima occorrenza dell'oggetto o, se ne esiste almeno una. Se la rimozione ha avuto luogo, restituisce *true*. |
| `Object set(int i, Object o)` | Sostituisce con *o* l'elemento alla posizione *i*. Restituisce l'elemento rimpiazzato. |
| `int size()` | Restituisce il numero degli elementi nella lista. |
| `Object[] toArray()` | Costruisce un array con gli stessi elementi della lista. |

**Tabella 17.3**
Le funzionalità più importanti della classe *Vector*.

Si prenda in esame il seguente esempio, basato sulle stringhe:

```
import java.util.*;

public class VectorTest {

  public static void main(String[] args) {
    Vector v = new Vector();
    v.add("Carlo");
    v.add("Marco");
    v.add("Francesca");
    v.add("Alessandro");
    System.out.println("v ha " + v.size() + " elementi.");
    System.out.println("Il primo è " + v.firstElement());
    System.out.println("L'ultimo è " + v.lastElement());
    System.out.println("Ora li sfoglio usando get():");
    for (int i = 0; i < v.size(); i++) {
      String s = (String)v.get(i);
      System.out.println("get(" + i + "): " + s);
    }
    System.out.println("Ora lo faccio con un enumeratore:");
    Enumeration e = v.elements();
    while (e.hasMoreElements()) {
      String s = (String)e.nextElement();
      System.out.println(s);
    }
  }

}
```

L'output prodotto è:

```
v ha 4 elementi.
Il primo è Carlo
L'ultimo è Alessandro
Ora li sfoglio usando get():
get(0): Carlo
get(1): Marco
get(2): Francesca
get(3): Alessandro
Ora lo faccio con un enumeratore:
Carlo
Marco
Francesca
Alessandro
```

***ArrayList***, rispetto a *Vector*, fornisce minore dinamicità e controllo, ma ha prestazioni generalmente migliori.

| **Costruttori pubblici** | |
|---|---|
| `ArrayList()` | Costruisce una lista di tipo *ArrayList* inizialmente vuota. |
| **Metodi pubblici** | |
| `void add(int i, Object o)` | Aggiunge l'oggetto *o* alla lista, disponendolo alla posizione *i*. Se la posizione è già occupata, l'elemento corrispondente e tutti i suoi successivi verranno avanzati di un posto. |
| `boolean add(Object o)` | Aggiunge l'oggetto *o* in coda alla lista. |

| | |
|---|---|
| `void clear()` | Ripulisce la lista, eliminando tutti i suoi elementi. |
| `boolean contains(Object o)` | Restituisce *true* se la lista contiene l'oggetto *o*. |
| `Object get(int i)` | Restituisce l'elemento alla posizione *i*. |
| `int indexOf(Object o)` | Restituisce l'indice dell'elemento *o*, o -1 se l'elemento non compare nella lista. |
| `boolean isEmpty()` | Restituisce *true* se la lista è vuota. |
| `Object remove(int i)` | Individua l'elemento alla posizione *i*, lo rimuove dalla lista e lo restituisce al codice chiamante. Tutti gli elementi successivi saranno arretrati di una posizione. |
| `Object set(int i, Object o)` | Sostituisce con *o* l'elemento alla posizione *i*. Restituisce l'elemento rimpiazzato. |
| `int size()` | Restituisce il numero degli elementi nella lista. |
| `Object[] toArray()` | Costruisce un array con gli stessi elementi della lista. |

**Tabella 17.4**
Le funzionalità più importanti della classe *ArrayList*.

L'utilizzo di *ArrayList* è simile a quello di *Vector*, come dimostra l'esempio:

```
import java.util.*;

public class ArrayListTest {

  public static void main(String[] args) {
    ArrayList list = new ArrayList();
    list.add("Fragola");
    list.add("Ribes");
    list.add("Amarena");
    list.add("Nocciola");
    list.add("Cioccolato");
    list.add("Eucalipto");
    list.add("Stracciatella");
    list.add("Crema");
    list.add("Ananas");
    System.out.println("list ha " + list.size() + " elementi.");
    System.out.println("Ora li sfoglio usando get():");
    for (int i = 0; i < list.size(); i++) {
      String s = (String)list.get(i);
      System.out.println("get(" + i + "): " + s);
    }
  }

}
```

In output si vedrà apparire:

```
list ha 9 elementi.
Ora li sfoglio usando get():
get(0): Fragola
get(1): Ribes
get(2): Amarena
get(3): Nocciola
```

```
get(4): Cioccolato
get(5): Eucalipto
get(6): Stracciatella
get(7): Crema
get(8): Ananas
```

Un altro tipo di lista, che può essere facilmente impiegato consultando la documentazione ufficiale, è *LinkedList*.

### 17.4 - Dizionari e mappe

Un dizionario è una raccolta di coppie *chiave-valore*. La classe astratta ***Dictionary*** definisce l'uso dei dizionari nel pacchetto *java.util*. Pertanto, tutti i tipi di dizionari che saranno esaminati funzionano più o meno alla stessa maniera. Hanno tutti i seguenti metodi:

- ***Enumeration elements()***. Restituisce l'enumerazione dei valori presenti nel dizionario.
- ***Object get(Object key)***. Restituisce il valore dell'elemento con chiave *key*, o *null* se tale elemento non fa parte dell'insieme.
- ***boolean isEmpty()***. Restituisce *true* se l'insieme è vuoto.
- ***Enumeration keys()***. Restituisce l'enumerazione delle chiavi presenti nel dizionario.
- ***Object put(Object key, Object value)***. Nel caso non esista un elemento con chiave *key*, lo crea con valore *value*. In caso contrario, ne aggiorna semplicemente il valore. In caso di aggiornamento, il vecchio valore viene restituito, altrimenti sarà ritornato *null*.
- ***Object remove(Object key)***. Rimuove dall'insieme l'elemento con chiave *key*. Se l'elemento non esiste, restituisce *null*, altrimenti ritorna il suo valore.
- ***int size()***. Restituisce il numero degli elementi presenti nell'insieme.

In *java.util* trovano posto diverse implementazioni concrete di *Dictionary*. La prima è ***Hashtable***. Ogni oggetto, in Java, ha un metodo *hashCode()* ereditato direttamente da *Object*. Compito di questo metodo è elaborare un numero *int* chiamato *codice hash*, il cui scopo è identificare univocamente il contenuto dell'oggetto. Due oggetti differenti, quindi, devono sempre avere un codice hash diverso, tranne nel caso in cui siano uguali i loro contenuti. Due stringhe uguali, ad esempio, hanno lo stesso codice hash, anche nel caso in cui siano oggetti separati in memoria. Lo si può dimostrare con uno stralcio di codice del tipo:

```
String s1 = "Ciao";
String s2 = "Ci" + "ao";
System.out.println(s1.hashCode());
System.out.println(s2.hashCode());
```

In sostanza, due oggetti hanno lo stesso codice hash solo se il loro confronto mediante il metodo *equals()* restituisce *true*.
Preso nota di questo, *Hashtable* è un dizionario che impiega i codici hash per sveltire l'identificazione delle chiavi e, di conseguenza, l'accesso ai valori. Ad ogni modo, non bisogna curarsi esplicitamente dei codici hash: la loro gestione avviene internamente agli oggetti *Hashtable*. Lo sviluppatore può tranquillamente continuare a ragionare secondo delle coppie chiave-valore. Il motivo dell'impiego dei codici hash è da ricercarsi nelle maggiori prestazioni che questi possono garantire, rispetto ai continui controlli imposti dai

metodi *equals()* dei singoli oggetti. Un esempio pratico:

```
import java.util.*;

public class HashtableTest {

  public static void main(String[] args) {
    Hashtable anniDiNascita = new Hashtable();
    anniDiNascita.put("Alessandro", new Integer(1978));
    anniDiNascita.put("Guido", new Integer(1979));
    anniDiNascita.put("Carlo", new Integer(1980));
    System.out.println(
      "Alessandro: " + anniDiNascita.get("Alessandro")
    );
    System.out.println(
      "Guido: " + anniDiNascita.get("Guido")
    );
    System.out.println(
      "Carlo: " + anniDiNascita.get("Carlo")
    );
  }

}
```

Dopo le spiegazioni date, l'output del programma è ovvio:

```
Alessandro: 1978
Guido: 1979
Carlo: 1980
```

***Properties*** è un altro tra i dizionari presenti in *java.util*. Deriva direttamente da *Hashtable*, specializzando questa classe nella gestione di coppie dove sia le chiavi sia i valori sono delle stringhe. Per facilitare l'accesso ai valori, *Properties* definisce diversi nuovi metodi:

- ***String getProperty(String key)***. Restituisce il valore dalla proprietà chiamata *key*, o *null* se tale proprietà non esiste.
- ***String getProperty(String key, String defaultValue)***. Restituisce il valore dalla proprietà chiamata *key*, o *defaultValue* se tale proprietà non esiste.
- ***list(java.io.PrintStream out)***. Elenca tutte le proprietà nel canale *out*.
- ***list(java.io.PrintWriter out)***. Elenca tutte le proprietà nel canale *out*.
- ***load(java.io.InputStream in)***. Carica dal canale di input *in* un insieme di proprietà precedentemente salvato.
- ***Enumeration propertyNames()***. Restituisce l'enumerazione dei nomi di ogni proprietà conservata nell'insieme.
- ***Object setProperty(String key, String value)***. Nel caso non esista una proprietà chiamata *key*, la crea con valore *value*. In caso contrario, ne aggiorna semplicemente il valore. In caso di aggiornamento, il vecchio valore viene restituito, altrimenti sarà ritornato *null*.
- ***void store(java.io.OutputStream out, String header)***. Salva tutte le proprietà nel canale *out*, facendole precedere dall'intestazione *header*.

Ecco un esempio:

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

```
public class PropertiesTest {

  public static void main(String[] args) {
    Properties compleanni = new Properties();
    compleanni.setProperty("Massimo", "11 Marzo");
    compleanni.setProperty("Francesca", "28 Marzo");
    compleanni.setProperty("Carlo", "16 Febbraio");
    compleanni.list(System.out);
  }

}
```

L'output prodotto è:

```
Massimo=11 Marzo
Carlo=16 Febbraio
Francesca=28 Marzo
```

La classe *Properties* è stata già citata nel corso della lezione precedente. Le proprietà dell'ambiente di esecuzione di Java possono essere recuperate tutte assieme, con il metodo *System.getProperties()*. Il seguente codice elenca tutte le proprietà dell'ambiente di esecuzione:

```
import java.util.*;

public class SystemPropertiesTest {

  public static void main(String[] args) {
    Properties p = System.getProperties();
    p.list(System.out);
  }

}
```

Nella prossima lezione sarà illustrato l'uso dei canali di Input/Output di Java. Pertanto, presto si potrà impiegare il file system, per leggere e salvare le proprietà utili alle proprie applicazioni direttamente su file di testo.

Le mappe, tutte basate sull'interfaccia ***Map***, si usano alla stessa maniera dei dizionari. Sono state aggiunte nelle più recenti versioni di Java per motivi di eleganza implementativa. Ad ogni modo chi sa usare i dizionari sa usare anche le mappe. Un tipo di mappa basato sui codici hash è ***HashMap***.

## 17.5 - StringTokenizer

La classe *StringTokenizer* fornisce un basilare scanner per l'interpretazione del testo. Tramite essa, è possibile prendere in esame una stringa, spezzandola in più parti (*token*) secondo determinate regole. *StringTokenizer* implementa l'interfaccia *Enumeration*, in modo che i singoli token in cui viene scomposta una stringa possano essere sfogliati in maniera classica e semplice. La seguente tabella riporta i membri pubblici di *StringTokenizer*.

| **Costruttori pubblici** | |
|---|---|
| `StringTokenizer(String str)` | Costruisce uno *StringTokenizer* per la stringa *str*, che come delimitatori usa i caratteri " \t\n\r\f". |

| | |
|---|---|
| `StringTokenizer(String str, String delim)` | Costruisce uno *StringTokenizer* per la stringa *str*, che come delimitatori usa i caratteri contenuti nella stringa *delim*. |
| `StringTokenizer(String str, String delim, boolean returnDelims)` | Costruisce uno *StringTokenizer* per la stringa *str*, che come delimitatori usa i caratteri contenuti nella stringa *delim*. Se *returnDelims* è *true*, i caratteri divisori verrano restituiti come token. |
| **Metodi pubblici** | |
| `int countTokens()` | Restituisce il numero dei token elaborati. |
| `boolean hasMoreElements()` | Come il successivo *hasMoreTokens()*. |
| `boolean hasMoreTokens()` | Restituisce *true* se ci sono ancora dei token da considerare. |
| `Object nextElement()` | Restituisce il token successivo, sotto forma di *Object*. |
| `String nextToken()` | Restituisce il token successivo, sotto forma di *String*. |
| `String nextToken(String delim)` | Imposta una nuova serie di caratteri delimitatori, quindi restituisce il token successivo. |

**Tabella 17.5**
I membri pubblici della classe *StringTokenizer*.

Per utilizzare uno *StringTokenizer* è necessario specificare una stringa da esaminare ed una stringa che contiene dei delimitatori. I delimitatori sono i caratteri che separano i singoli token. Ciascun carattere contenuto nella stringa dei delimitatori è un delimitatore. Ad esempio, la stringa ".,:" specifica tre differenti delimitatori: il punto, la virgola e i due punti. Se si fa ricorso al costruttore privo di argomenti, ha valore una stringa di delimitatori predefinita, costituita dalla sequenza " \t\n\r\f " (spazio, tabulatore, fine riga, ritorno a capo, avanzamento modulo).

```
StringTokenizer st = StringTokenizer("Ciao, mondo!");
```

Questo *StringTokenizer* usa l'elenco di delimitatori predefinito. Pertanto, individuerà due token: "Ciao," e "mondo!" (lo spazio è tra i delimitatori di default).

```
StringTokenizer st = StringTokenizer("Ciao, mondo!", ",");
```

In questo caso, invece, si è scelto di utilizzare la virgola come unico delimitatore. I token che ne derivano sono "Ciao" e " mondo!". In ambo i casi visti, i caratteri delimitatori vengono tagliati fuori dai token elaborati. Se non si desidera perdere i delimitatori, conviene usare la seguente forma:

```
StringTokenizer st = StringTokenizer("Ciao, mondo!", ",", true);
```

In questo caso, si ottengono tre token: "Ciao", "," e "mondo!". Ciascuna delimitatore incontrato viene restituito come singolo token.
Esplorare i token elaborati è molto semplice. Per prima cosa, *StringTokenizer* implementa

*Enumeration*, e quindi definisce i metodi *hasMoreElements()* e *nextElement()*. Perciò il seguente modello di esplorazione è sempre valido:

```
StringTokenizer st = ...;
while (st.hasMoreElements()) {
  String token = (String)st.nextElement();
  // Lavora con il token.
}
```

In alternativa, se si vuole evitare il casting, sono disponibili i metodi specializzati *hasMoreTokens()* e *nextToken()*:

```
StringTokenizer st = ...;
while (st.hasMoreTokens()) {
  String token = st.nextToken();
  // Lavora con il token.
}
```

Dopo aver esaminato una prima serie di token, è possibile cambiare repentinamente l'insieme dei delimitatori, usando il metodo:

```
String token = st.nextToken("nuovi delimitatori");
```

*StringTokenizer* acquisirà i nuovi delimitatori, e già il token restituito dalla chiamata al metodo sarà ottenuto secondo le nuove norme di separazione.

Sarà ora illustrato un esempio completo. Scopo dell'applicazione da realizzare è prendere in esame una stringa contenente delle coppie chiave-valore, organizzata alla seguente maniera:

```
nome=Mario
cognome=Rossi
impiego=Operaio
```

Ci sono due delimitatori in una sequenza come questa: l'uguale, che separa la chiave dal corrispondente valore, ed il ritorno a nuova riga, che isola le singole coppie. L'applicazione da sviluppare deve interpretare correttamente le stringhe formulate secondo questa norma, per importarne il contenuto all'interno di un oggetto *Properties*. Successivamente, per dimostrare il buon esito dell'operazione, dovrà stampare in output le proprietà raccolte. Ecco il codice:

```
import java.util.*;

public class StringTokenizerTest {

  public static void main(String[] args) {
    String demoString = "nome=Mario\r\n" +
                        "cognome=Rossi\r\n" +
                        "impiego=Operaio\r\n";
    Properties p = new Properties();
    StringTokenizer st = new StringTokenizer(demoString, "=\r\n");
    while (st.hasMoreTokens()) {
      String key = st.nextToken();
      String value = st.nextToken();
      p.put(key, value);
    }
```

```
    p.list(System.out);
  }

}
```

Una volta eseguita, l'applicazione mostra in output le proprietà elaborate, che saranno ovviamente identiche a quelle di partenza (anche se l'ordine in cui compaiono potrebbe essere differente):

```
nome=Mario
impiego=Operaio
cognome=Rossi
```

Come ora è facile intuire, *StringTokenizer* è una classe molto utile quando si devono interpretare dei dati testuali acquisiti dall'esterno di un'applicazione.

## 17.6 - Timer e TimerTask

A partire dalla versione 1.3 della piattaforma standard Java 2, il package *java.util* contiene le classi *Timer* e *TimerTask*, utilizzabili per poter eseguire un processo ad intervalli regolari nel tempo. Realizzare un processo temporizzato diventa, quindi, piuttosto semplice. Per prima cosa è necessario includere tale processo all'interno del metodo *run()* di una classe che estenda *TimerTask* (che a sua volta implementa *Runnable*). A questo punto, grazie ad un'istanza di *Timer* e al suo metodo *schedule()*, è possibile eseguire il processo ripetutamente e ciclicamente nel tempo. Il seguente esempio stampa in output la scritta "Eccomi!" ogni tre secondi:

```
import java.util.*;

public class TimerTest extends TimerTask {

  public void run() {
    System.out.println("Eccomi!");
  }

  public static void main(String[] args) {
    TimerTest esempio = new TimerTest();
    Timer timer = new Timer();
    timer.schedule(esempio, 0, 3000);
    try {
      Thread.sleep(10000);
      timer.cancel();
    } catch (InterruptedException e) {}
  }

}
```

L'esecuzione del programma dura circa dieci secondi, poiché il thread principale viene messo in pausa manualmente. In questo frangente, il thread secondario gestito dall'istanza di *Timer* stampa circa quattro scritte (la prima immediatamente, le altre ad intervalli di tre secondi). Il metodo *cancel()* annulla l'esecuzione temporizzata allo scadere dei dieci secondi d'attesa.
Altre forme di *schedule()* permettono l'avvio del processo in un momento prestabilito, oppure dopo un'attesa di *tot* millisecondi, anche senza la ripetizione ciclica. Maggiori informazioni possono essere reperite nella documentazione ufficiale.

## 17.7 - Random

La classe *Random* lavora come generatore di numeri pseudo-casuali. Ad ogni istanza di *Random* è associato un seme, vale a dire un valore numerico usato come spunto di partenza per il calcolo della sequenza pseudo-casuale. Due istanze di *Random* con il medesimo seme restituiscono sempre la stessa sequenza pseudo-casuale.

| **Costruttori pubblici** | |
|---|---|
| `Random()` | Costruisce un nuovo generatore di numeri pseudo-casuali, usando la data e l'ora correnti come seme. |
| `Random(int seed)` | Costruisce un nuovo generatore di numeri pseudo-casuali, usando il seme specificato. |
| **Metodi pubblici** | |
| `boolean nextBoolean()` | Restituisce il *boolean* successivo nella sequenza pseudo-casuale. |
| `void nextBytes(bytes[] b)` | Registra sull'argomento fornito una sequenza di *b.length* numeri pseudo-casuali, in formato *byte*. |
| `double nextDouble()` | Restituisce il *double* successivo nella sequenza pseudo-casuale, compreso tra 0 (incluso) ed 1 (escluso). |
| `float nextFloat()` | Restituisce il *float* successivo nella sequenza pseudo-casuale, compreso tra 0 (incluso) ed 1 (escluso). |
| `double nextGaussian()` | Restituisce il *double* gaussiano successivo nella sequenza pseudo-casuale, compreso tra 0 (incluso) ed 1 (escluso). |
| `int nextInt()` | Restituisce l'*int* successivo nella sequenza pseudo-casuale. |
| `int nextInt(int n)` | Restituisce l'*int* successivo nella sequenza pseudo-casuale, compreso tra 0 (incluso) ed n (escluso). |
| `long nextLong()` | Restituisce il *long* successivo nella sequenza pseudo-casuale. |
| `void setSeed(long seed)` | Reimposta il seme della sequenza. |

**Tabella 17.6**
I membri pubblici della classe *Random*.

Ecco un esempio:

```
import java.util.*;

public class RandomTest {

  public static void main(String[] args) {
    // r1 usa un seme ogni volta diverso.
    Random r1 = new Random();
    // r1 usa un seme fisso.
    Random r2 = new Random(12176);
    // Dimostrazione della sequenza di r1.
    System.out.print("r1: ");
    for (int i = 0; i < 10; i++) {
      System.out.print(r1.nextInt(100) + " ");
```

```
    }
    System.out.println();
    // Dimostrazione della sequenza di r2.
    System.out.print("r2: ");
    for (int i = 0; i < 10; i++) {
      System.out.print(r2.nextInt(100) + " ");
    }
    System.out.println();
  }

}
```

L'output prodotto sarà del tipo:

```
r1: 90 75 16 14 51 24 21 75 75 21
r2: 63 8 20 79 85 12 72 27 93 28
```

Sono state usate due istanze di *Random*. La prima (*r1*) ha seme differente ad ogni esecuzione, giacché lo stesso viene prelevato automaticamente dall'orologio del sistema. In pratica, è come se fosse:

```
Random r1 = new Random(System.currentTimeMillis());
```

La seconda istanza di *Random* (*r2*), al contrario, impiega sempre il medesimo seme. Eseguendo l'applicazione più volte, è possibile notare come la sequenza prodotta da *r2* sarà sempre ed inevitabilmente la stessa.