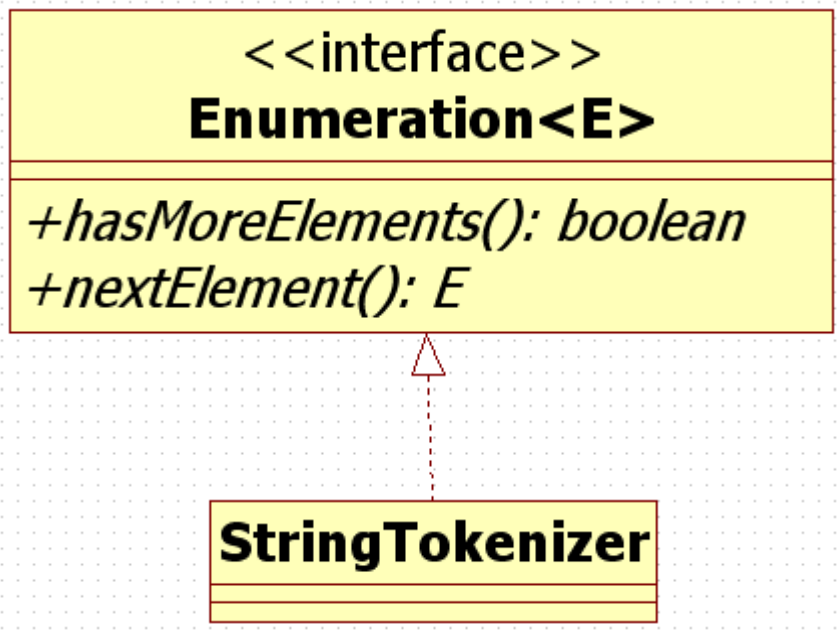
Class Library
java.util Package

Bok, Jong Soon
javaexpert@nate.com
https://github.com/swacademy/JavaSE

`Enumeration<E>` interface

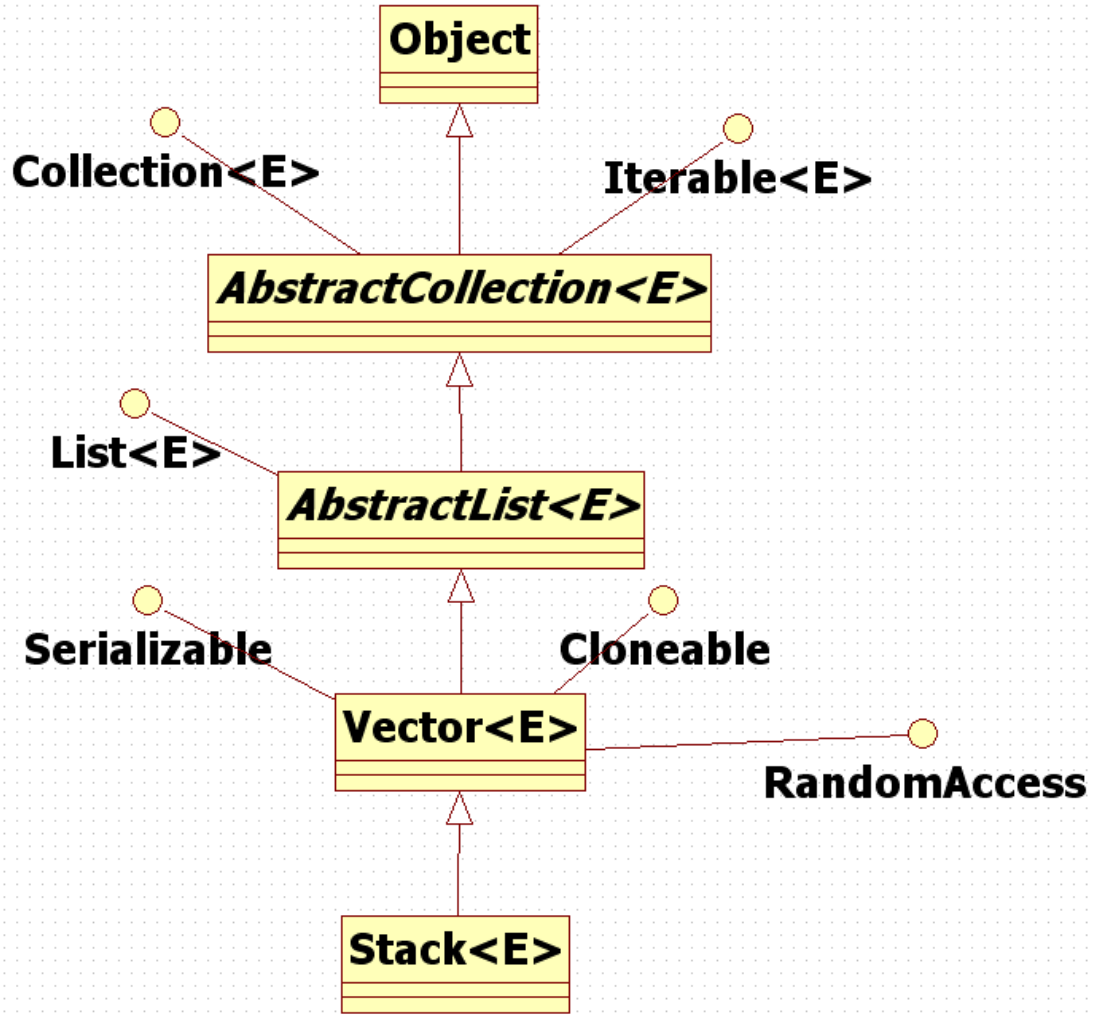
- An object that implements the `Enumeration` interface generates a series of elements, one at a time.
- Methods are provided to enumerate through the elements of a vector, the keys of a hashtable, and the values in a hashtable.



`Stack<E>` class

- Represents a last-in-first-out (*LIFO*) stack of objects.
- It extends class `Vector` with five operations that allow a vector to be treated as a stack.
- *push* and *pop* operations.
- When a stack is first created, it contains no items.

Stack<E> (Cont.)

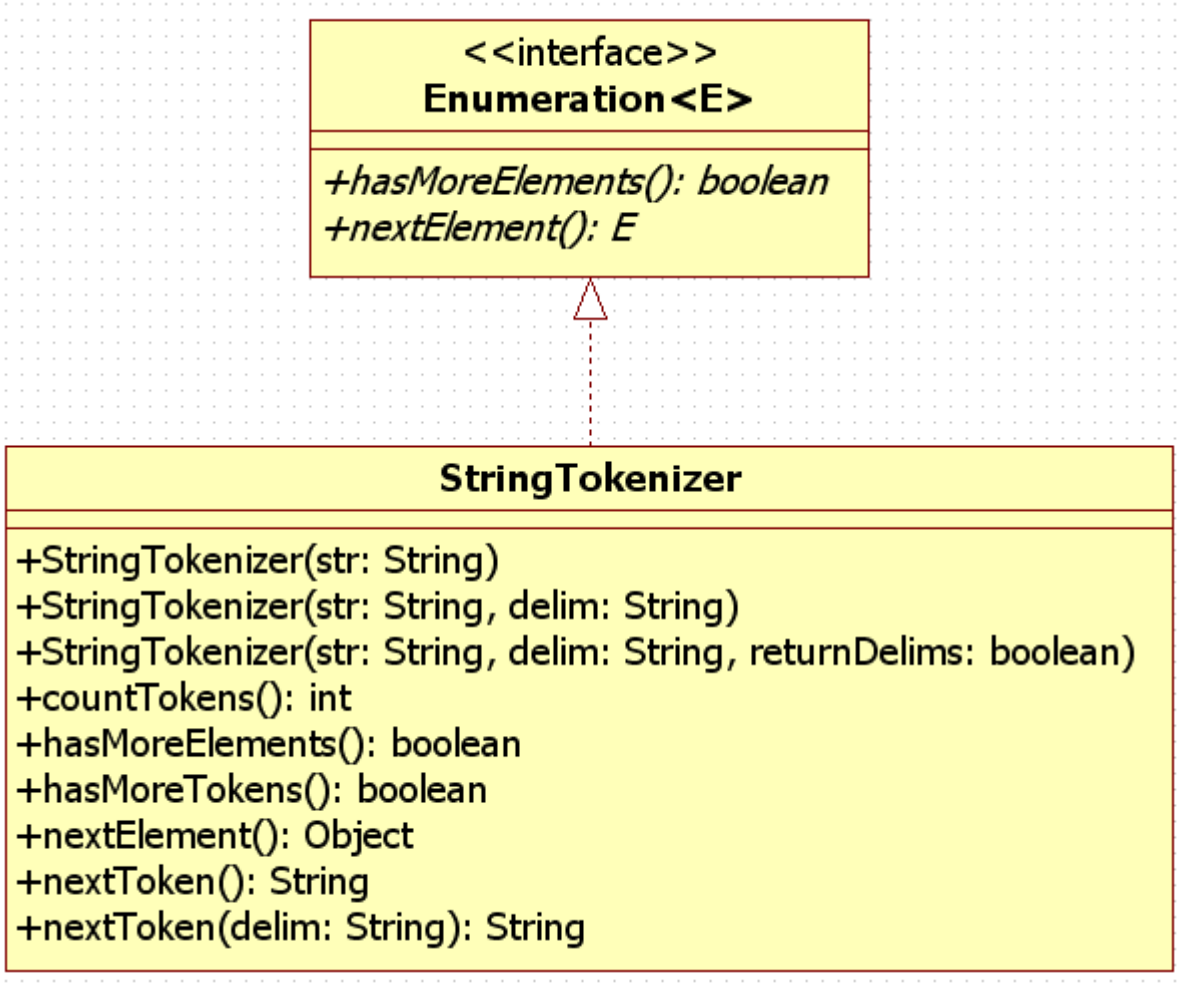


Stack<E>

+Stack()
+empty(): boolean
+peek(): E
+pop(): E
+push(item: E): E
+search(o: Object): int

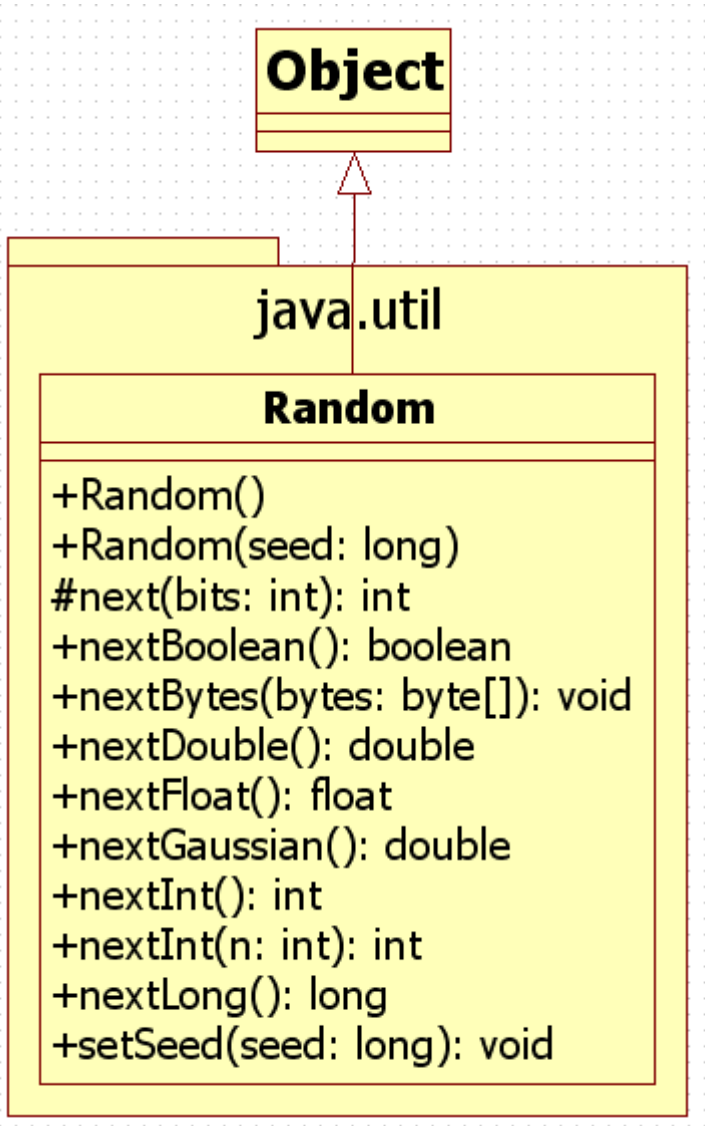
`StringTokenizer` class

- Allows an application to break a string into tokens.



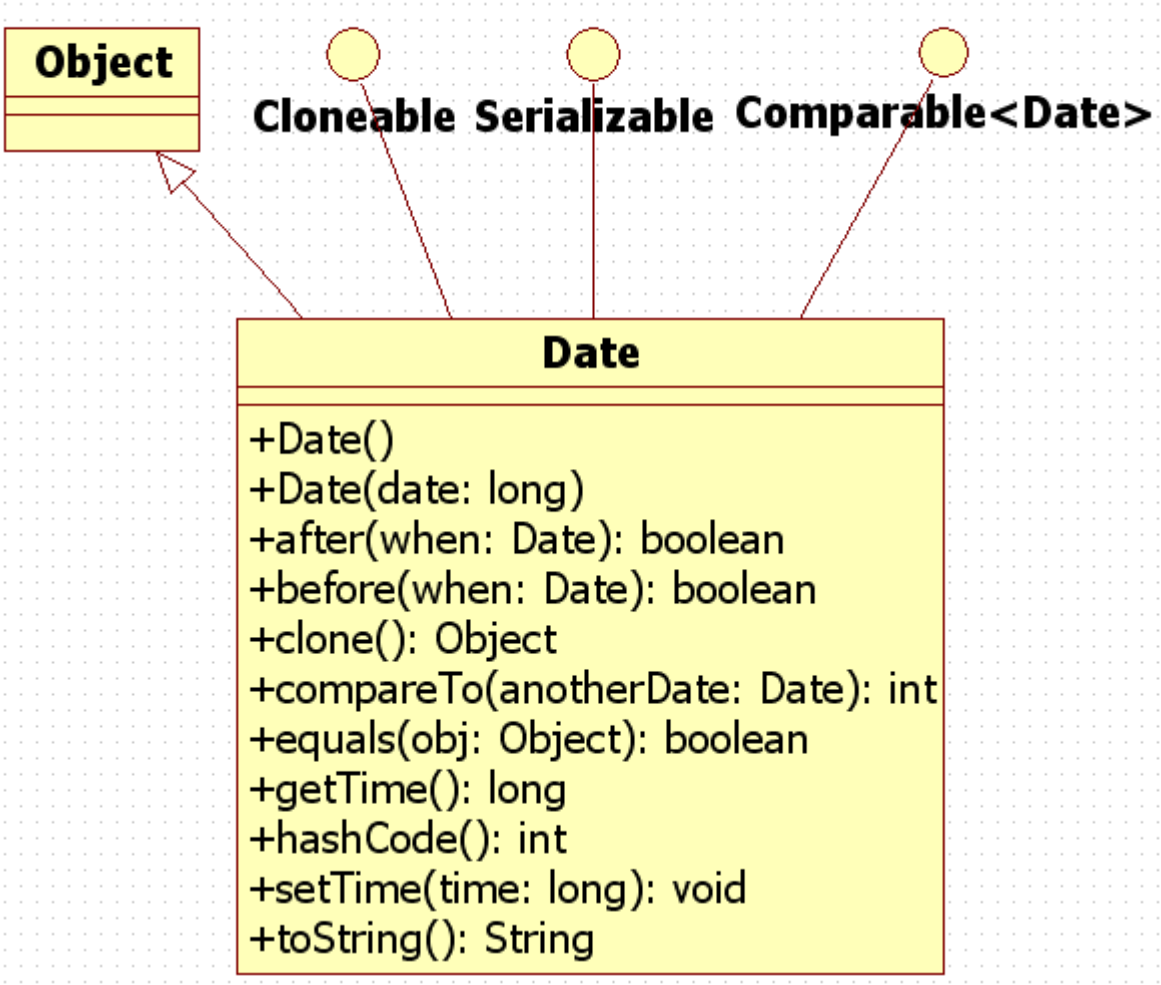
`Random` class

- Is used to generate a stream of pseudorandom numbers.



`Date` class

- Represents a specific instant in time, with millisecond precision.



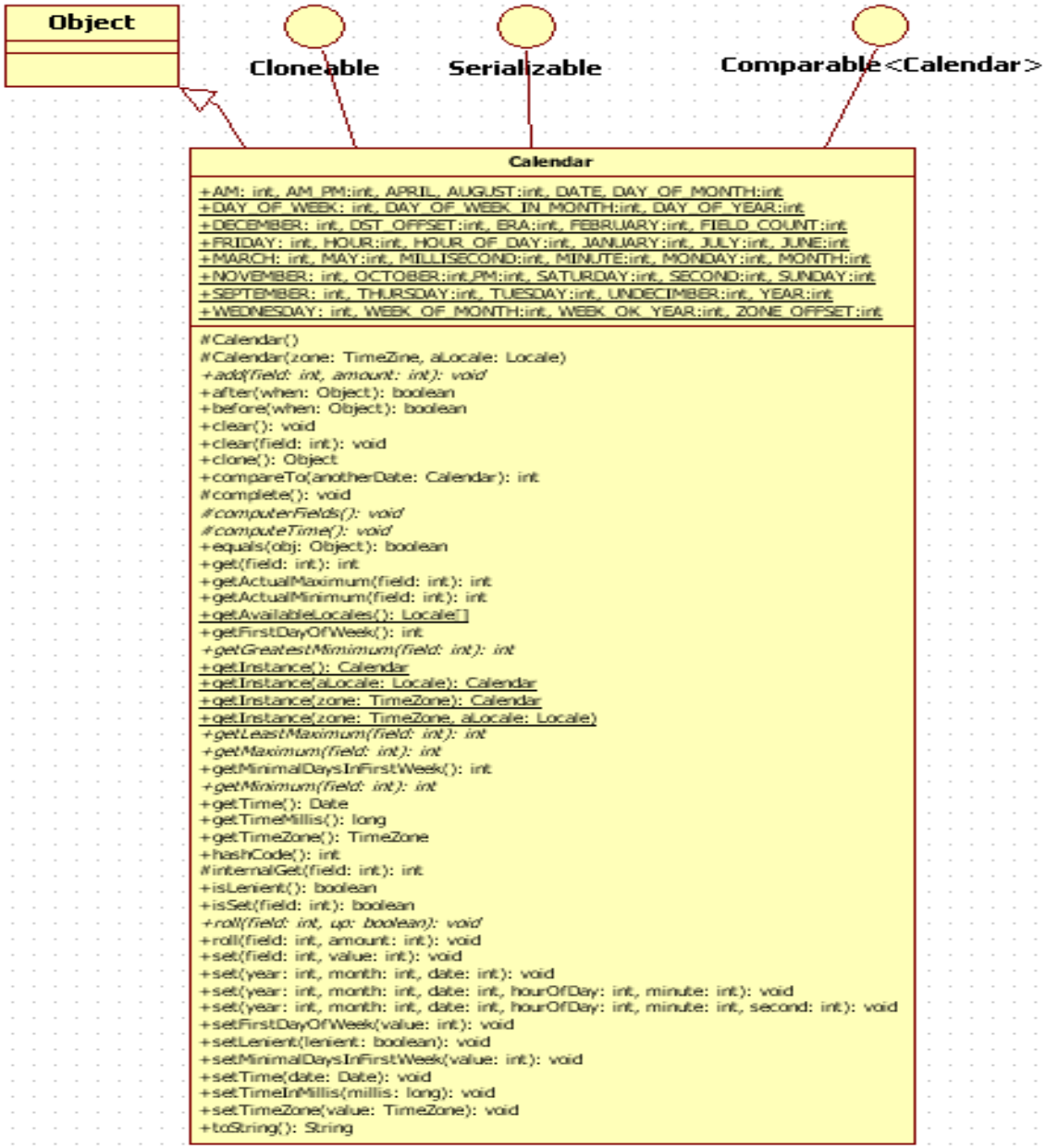
`Date` class (Cont.)

- Allowed the interpretation of dates as year, month, day, hour, minute, and second values.
- Also allowed the formatting and parsing of date strings.
- As of JDK 1.1, the `Calendar` class should be used to convert between dates and time fields and the `DateFormat` class should be used to format and parse date strings.
- The corresponding methods in `Date` are deprecated.

`Date` class (Cont.)

- A year is represented by the integer y – 1990
- A month is represented by an integer from 0 to 11
- A date (day of month) is represented by an integer from 1 to 31
- An hour is represented by an integer from 0 to 23
- A minute is represented by an integer from 0 to 59
- A second is represented by an integer from 0 to 61

`Calendar` class



`Calendar` class (Cont.)

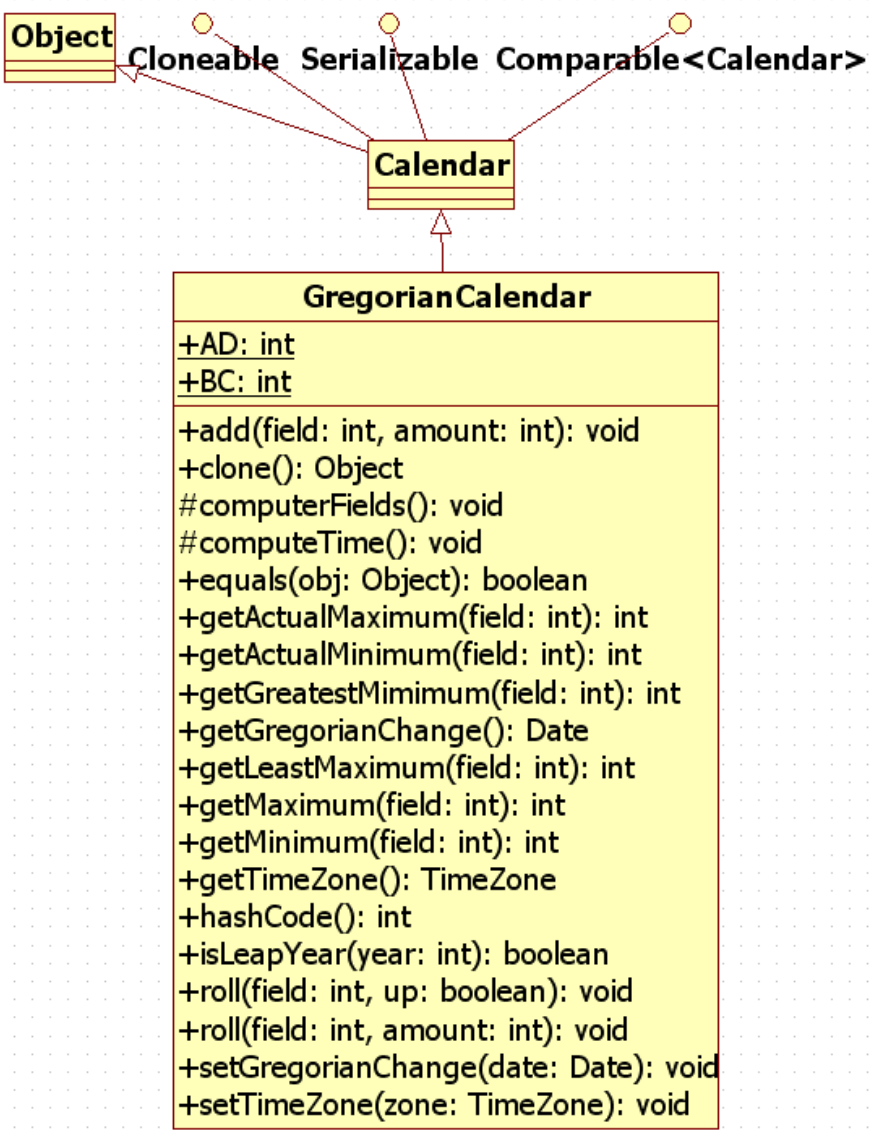
- `Calendar` is an abstract base class for converting between a `Date` object and a set of integer fields such as `YEAR`, `MONTH`, `DAY`, `HOUR`, and so on.
- Subclasses of `Calendar` interpret a `Date` according to the rules of a specific calendar system. The platform provides one concrete subclass of `Calendar`: `GregorianCalendar`.
- `Calendar`'s `getInstance()` method returns a `Calendar` object whose time fields have been initialized with the current date and time

`Calendar` class (Cont.)

- `static int DATE`
- `static int DAY_OF_WEEK`
- `static int DAY_OF_WEEK_IN_MONTH`
- `static int DAY_OF_YEAR`
- `static int MONTH`
- `static int HOUR`
- `static int HOUR_OF_DAY`
- `static int AM_PM`

`GregorianCalendar` class

- Is a concrete subclass of **`Calendar`** and provides the standard calendar used by most of the world.



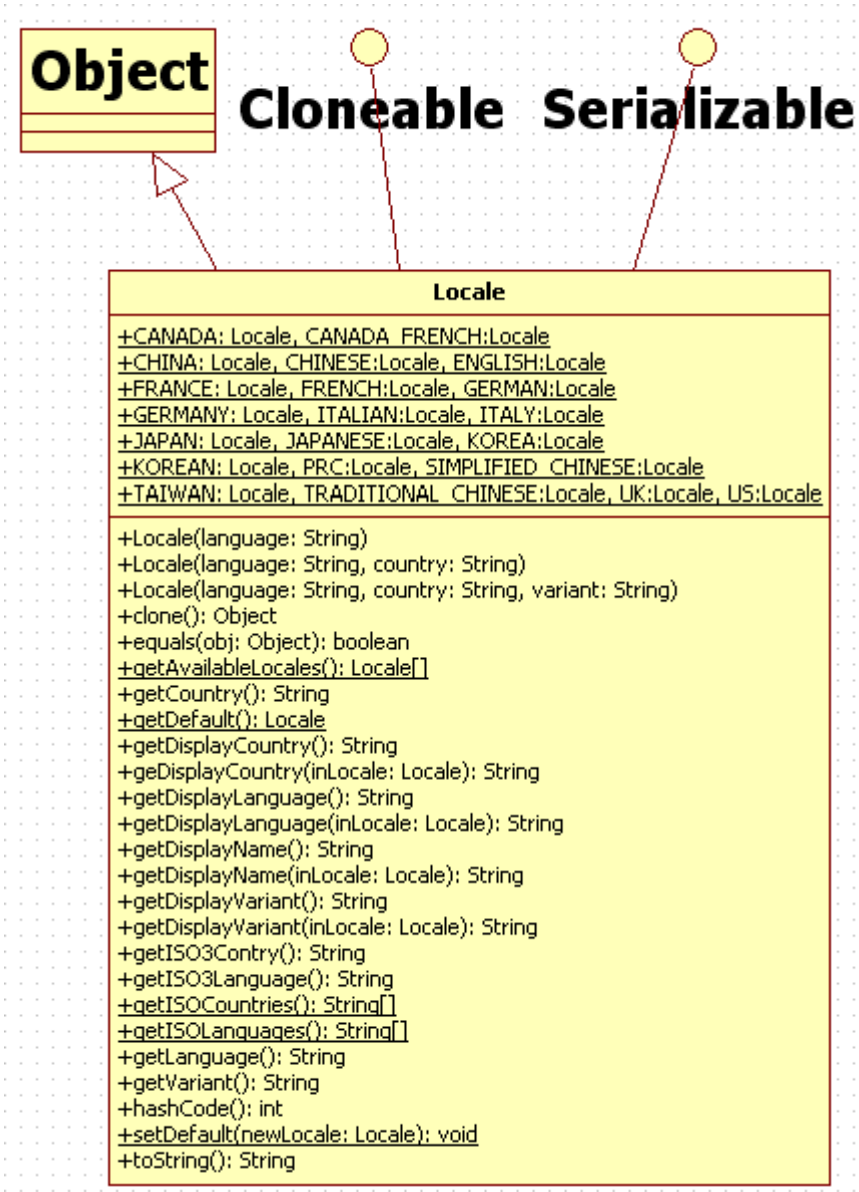
`GregorianCalendar` class (Cont.)

- The standard (Gregorian) calendar has 2 eras, BC and AD.
- This implementation handles a single discontinuity, which corresponds by default to the date the Gregorian calendar was instituted (October 15, 1582 in some countries, later in others).

`Locale` class

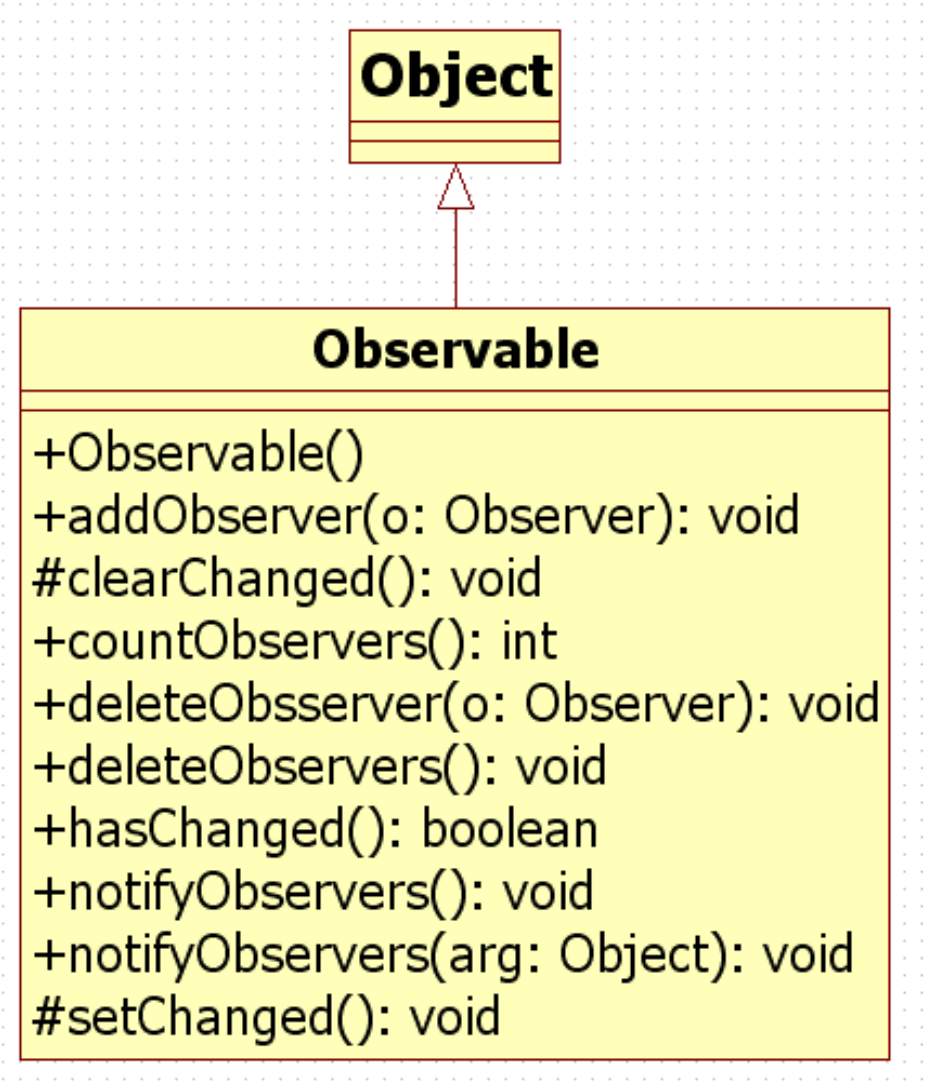
- A `Locale` object represents a specific geographical, political, or cultural region.
- The language argument is a valid ISO Language Code. These codes are the lower-case, two-letter codes as defined by ISO-639. You can find a full list of these codes at a number of sites, such as: http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt
- The country argument is a valid ISO Country Code. These codes are the upper-case, two-letter codes as defined by ISO-3166. You can find a full list of these codes at a number of sites, such as: http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html

Locale class (Cont.)



`Observable` class

- Represents an observable object.



`Observable` class (Cont.)

- It can be subclassed to represent an object that the application wants to have observed.
- An observer may be any object that implements interface `Observer`.
- After an observable instance changes, an application calling the `Observable`'s `notifyObservers()` method causes all of its observers to be notified of the change by a call to their `update()` method.

Observable class (Cont.)

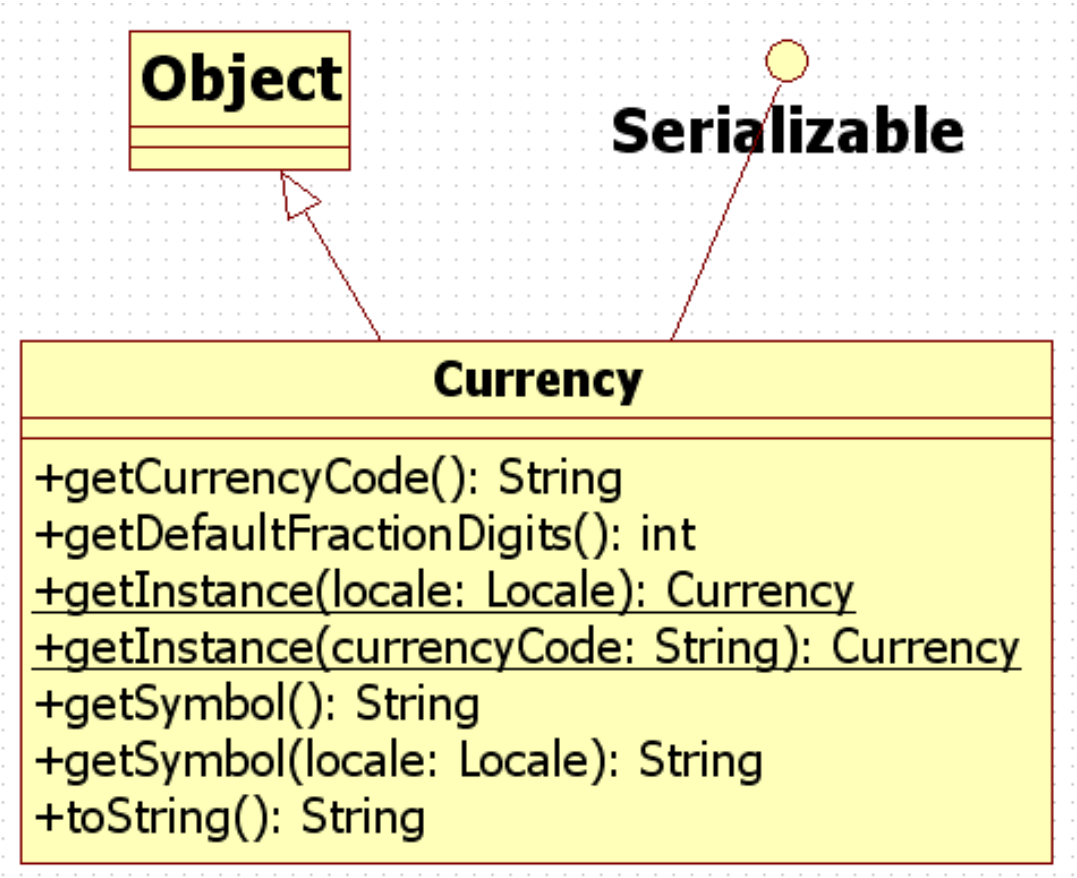
```java
import java.util.*;
import java.util.Observer;
public class Admin implements Observer {
    public void update(Observable arg0, Object arg1) {
        // TODO 자동 생성된 메소드 스텁
        System.out.println("update() 호출, " + arg1 +
                "님이 5번째 회원으로 가입했습니다");
        System.out.println("가입 일시 : " + new Date());
    }
}
```

```java
public class Member extends java.util.Observable {
    int count = 0;
    void register(String id){
        count++;
        if( count == 5){
            setChanged();
            notifyObservers(id);
        }else{
            try{
                Thread.sleep(1000);
                System.out.println("count = " + count);
            }catch(Exception e){}
        }
    }
}
```

```java
public class ObserverDemo {
    public static void main(String[] args) {
        // TODO 자동 생성된 메소드 스텁
        Member member = new Member();
        Admin admin = new Admin();
        member.addObserver(admin);
        for(int i=0;i<10;i++){
            member.register("id_" + i);
        }
    }
}
/*
count = 1
count = 2
count = 3
count = 4
update() 호출, id_4님이 5번째 회원으로 가입했습니다
가입 일시 : Mon Sep 25 23:46:42 KST 2006
count = 6
count = 7
count = 8
count = 9
count = 10
*/
```

`Currency` class

- Represents a currency.



`Currency` class (Cont.)

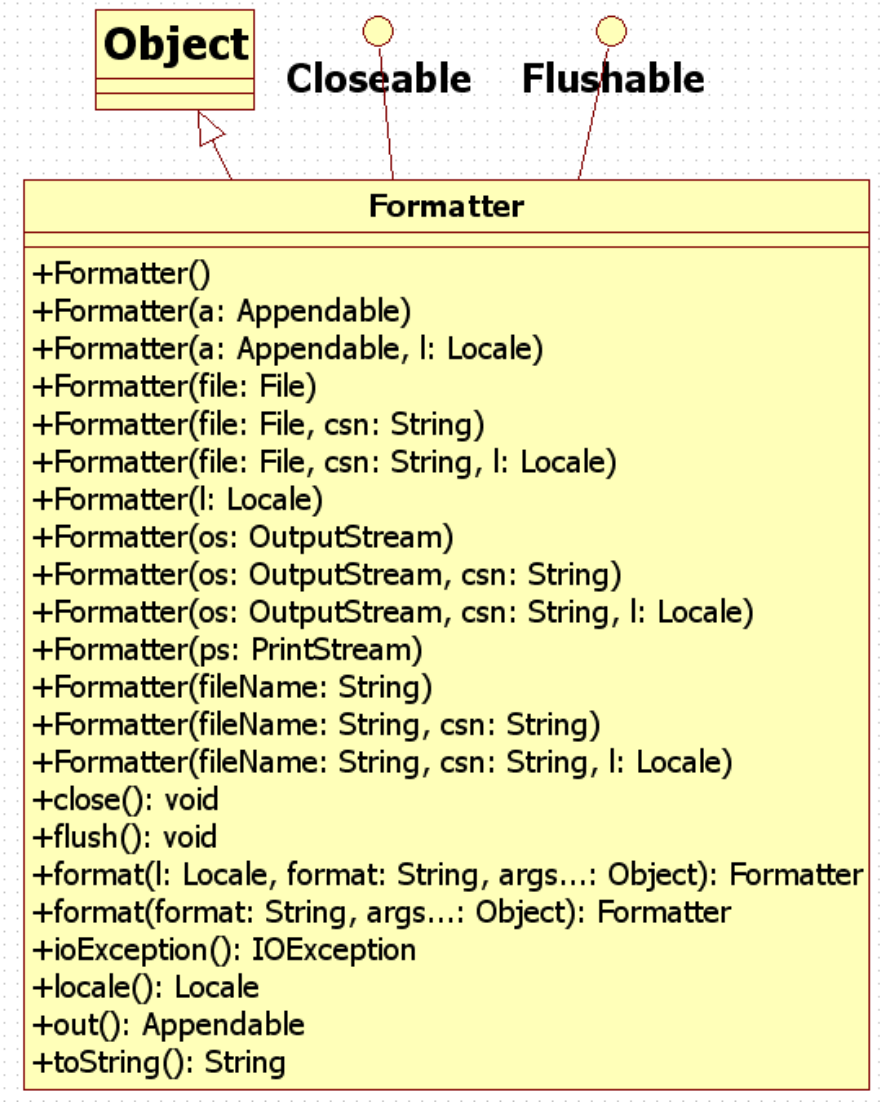
- Currencies are identified by their ISO 4217 currency codes. See the ISO 4217 maintenance agency for more information, including a table of currency codes.
- The class is designed so that there's never more than one `Currency` instance for any given currency. Therefore, there's no public constructor. You obtain a `Currency` instance using the `getInstance()` methods.

Currency class (Cont.)

```
1  public class CurrencyDemo {
2      public static void main(String[] args) {
3          // TODO 자동 생성된 메소드 스텁
4          java.util.Currency c;
5          c = java.util.Currency.getInstance(java.util.Locale.US);
6          System.out.println(c.getSymbol());
7          System.out.println(c.getDefaultFractionDigits());
8      }
9  }
10 /*
11 USD
12 2
13  */
```

`Formatter` class

- An interpreter for printf-style format strings.
- Provides support for layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output.



`Formatter` class (Cont.)

- Formatted printing for the Java language is heavily inspired by ***C-language's printf***.
- Java formatting is more strict than ***C-language***'s
- The format strings are thus intended to be recognizable to ***C-language*** programmers but not necessarily completely compatible with those in ***C-language***.

`Formatter` class (Cont.)

- Explicit argument indices may be used to re-order output.

```
fmt.format("%4$2s %3$2s %2$2s %1$2s",
"a", "b", "c", "d")  → " d c b a"
```

- Optional locale as the first argument can be used to get locale-specific formatting of numbers.

```
fmt.format(Locale.FRANCE, "PI
= %+10.4f", Math.PI); → "e = +3,1416"
```

`Formatter` class (Cont.)

- Writes a formatted string to System.out.

  `System.out.format("Local time: %tT", Calendar.getInstance());` → `"Local time: 13:34:18"`

- Writes formatted output to System.err.

  `System.err.printf("Unable to open file '%1$s': %2$s", fileName, exception.getMessage());` → `"Unable to open file 'food': No such file or directory"`

`Formatter` class (Cont.)

- Format a string containing a date.

```
import java.util.*;
Calendar c = new GregorianCalendar(1995,
MAY, 23);
String s = String.format("Duke's
Birthday: %1$tm %1$te,%1$tY", c);
```

→ `s == "Duke's Birthday: May 23, 1995"`

`Formatter` class (Cont.)

- The format specifiers for general, character, and numeric types have the following

  `%[argument_index$][flags][width][.precision]conversion`

- The optional *argument_index* is a decimal integer indicating the position of the argument in the argument list. The first argument is referenced by "1$", the second by "2$", etc.
- The optional *flags* is a set of characters that modify the output format. The set of valid flags depends on the conversion.

`Formatter` class (Cont.)

- The optional ***width*** is a non-negative decimal integer indicating the minimum number of characters to be written to the output.
- The optional ***precision*** is a non-negative decimal integer usually used to restrict the number of characters. The specific behavior depends on the conversion.
- The required ***conversion*** is a character indicating how the argument should be formatted. The set of valid conversions for a given argument depends on the argument's data type

`Formatter` class (Cont.)

Conversion	Argument Category	Description
‘b’, ‘B’	general	If args is null, “false”. Otherwise “true”
‘h’, ‘H’	general	If args is null, “null”. Otherwise Integer.toHexString()
‘s’, ‘S’	general	If args is null, “null”. Otherwise args.toString()
‘c’, ‘C’	character	A Unicode character.
‘d’	integral	A decimal integer.
‘o’	integral	An octal integer
‘x’, ‘X’	integral	An hexadecimal integer

`Formatter` class (Cont.)

Conversion	Argument Category	Description
‘e’, ‘E’	Floating point	A decimal number in computerized scientific notation
‘f’	Floating point	A decimal number
‘g’,’G’	Floating point	Computerized scientific notation or decimal format
‘a’, ‘A’	Floating point	A nexadecimal floating-point number
‘t’, ‘T’	Date/time	Prefix for date and time conversion characters.
‘%’	percent	A literal ‘%’ (‘\u0025’)
‘n’	Line separator	The platform-specific line separator

`Formatter` class (Cont.)

Conversion	Description
‘H’	Hour of the day for the 24-hour clock. i.e. 00 - 23.
‘I’	Hour for the 12-hour clock. i.e. 01 - 12.
‘k’	Hour of the day for the 24-hour clock, i.e. 0 - 23.
‘M’	Minute within the hour formatted as two digits with a leading zero as necessary, i.e. 00 - 59.
‘S’	Seconds within the minute, formatted as two digits with a leading zero as necessary, i.e. 00 – 60.
‘L’	Millisecond within the second formatted as three digits with leading zeros as necessary, i.e. 000 - 999.

`Formatter` class (Cont.)

Conversion	Description
‘N’	Nanosecond within the second, formatted as nine digits with leading zeros as necessary, i.e. 000000000 - 999999999.
‘p’	Locale-specific morning or afternoon marker in lower case, e.g."am" or "pm". Use of the conversion prefix 'T' forces this output to upper case.
‘z’	RFC 822 style numeric time zone offset from GMT, e.g. -0800.
‘Z’	A string representing the abbreviation for the time zone.
‘s’	Seconds since the beginning of the epoch starting at 1 January 1970 00:00:00 UTC, i.e. Long.MIN_VALUE/1000 to Long.MAX_VALUE/1000.
‘Q’	Milliseconds since the beginning of the epoch starting at 1 January 1970 00:00:00 UTC, i.e. Long.MIN_VALUE to Long.MAX_VALUE.

`Formatter` class (Cont.)

Conversion	Description
‘B’	Locale-specific full month name, e.g. "January", "February".
‘b’, ‘h’	Locale-specific abbreviated month name, e.g. "Jan", "Feb".
‘A’	Locale-specific full name of the day of the week, e.g. "Sunday", "Monday"
‘a’	Locale-specific short name of the day of the week, e.g. "Sun", "Mon"
‘C’	Four-digit year divided by 100, formatted as two digits with leading zero as necessary, i.e. 00 - 99
‘Y’	Year, formatted as at least four digits with leading zeros as necessary, e.g. 0092 equals 92 CE for the Gregorian calendar.

`Formatter` class (Cont.)

Conversion	Description
‘y’	Last two digits of the year, formatted with leading zeros as necessary, i.e. 00 - 99.
‘j’	Day of year, formatted as three digits with leading zeros as necessary, e.g. 001 - 366 for the Gregorian calendar.
‘m’	Month, formatted as two digits with leading zeros as necessary, i.e. 01 - 13.
‘d’	Day of month, formatted as two digits with leading zeros as necessary, i.e. 01 - 31
‘e’	Day of month, formatted as two digits, i.e. 1 - 31.

`Formatter` class (Cont.)

Conversion	Description
‘R’	Time formatted for the 24-hour clock as "%tH:%tM"
‘T’	Time formatted for the 24-hour clock as "%tH:%tM:%tS".
‘r’	Time formatted for the 12-hour clock as "%tI:%tM:%tS %Tp".
‘D’	Date formatted as "%tm/%td/%ty".
‘F’	ISO 8601 complete date formatted as "%tY-%tm-%td".
‘c’	Date and time formatted as "%ta %tb %td %tT %tZ %tY", e.g. "Sun Jul 20 16:17:00 EDT 1969".

Formatter class (Cont.)

```
1 import java.util.Formatter;
2 import java.util.Calendar;
3 public class FormatterDemo {
4     public static void main(String[] args) {
5         // TODO 자동 생성된 메소드 스텁
6         Formatter fmt = new Formatter();
7         Calendar cal = Calendar.getInstance();
8         System.out.println(fmt.format("%tr", cal));
9         fmt = new Formatter();
10        System.out.println(fmt.format("%tc", cal));
11        fmt = new Formatter();
12        System.out.println(fmt.format("%tl : %tM", cal, cal));
13        fmt = new Formatter();
14        System.out.println(fmt.format("%tB %tb %tm", cal, cal, cal));
15    }
16 }
17 /*
18 12:34:36 오전
19 화 9월 26 00:34:36 KST 2006
20 12 : 34
21 9월 9월 09
22 */
```

```
22 /*
23 11:09:48 PM
24 Tue Sep 26 23:09:48 KST 2006
25 11 : 09
26 September Sep 09
27 */
```

Formatter class (Cont.)

```
2  public class FormatterDemo1 {
3      public static void main(String[] args) {
4          // TODO 자동 생성된 메소드 스텁
5          java.util.Formatter fmt = new java.util.Formatter();
6          String name = "Sujan";
7          String str = fmt.format("안녕하세요. %s님 %tD는 저의 생일입니다", name,
8                  new java.util.Date()).toString();
9          System.out.println(str);
10     }
11 }
12 /*
13 안녕하세요. Sujan님 09/26/06는 저의 생일입니다
14 */
```

```
1  import java.util.Formatter;
2  public class FormatterDemo3 {
3      public static void main(String[] args) throws Exception {
4          // TODO 자동 생성된 메소드 스텁
5          Formatter fmt = new Formatter("output.txt");
6          fmt.format("%f", 123456789.01234);
7          fmt.flush();
8      }
9  }
```