

---

## Chapter 9

# Some Useful Tools

---

### What is in This Chapter ?

There are many useful classes in JAVA. We take a look here at some of the commonly used ones. It is important to know some of these available classes so that we don't end up re-creating a method that already exists in JAVA. We begin with a discussion of the **String** and **StringBuilder** classes and describe how to use some of their methods. We also discuss how **enumerated types** and the **Character** class can simplify our code. Lastly, we discuss the **Date** and **Calendar** classes and how they can be used to represent time and date information. Obviously, there are many more classes in JAVA and you should refer to the API to get more information.



## 9.1 The string Class

**Strings** are one of the most commonly used concepts in all programming languages. They are used to represent text characters and are fundamental in allowing a user to interact with the program. In JAVA, Strings are actually objects, not primitives and any text between double quotes represents a *literal* String in our programs:

```
String name = "Stan Dupp";
String empty = "";
```

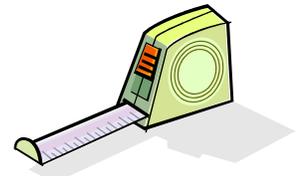
However, since Strings are also objects, we can create one by using one of many available constructors. Here are two examples:

```
String nothing = new String();           // makes an empty String
String copy = new String(name);         // makes copy of the name String
```



A **String** has a *length* corresponding to the number of characters in the String. We can ask a **String** for its length by using the **length()** method:

```
String name = "Stan Dupp";
String empty = "";
name.length();    // returns 9
empty.length();   // returns 0
```



This length remains unchanged for the string at all times. That is, once a string has been created we *cannot change the size* of the string, nor can we *append* to the string.

Even though we cannot append to a String, we can still make use of the **+** operator to join two of them together. Recall, for example, the use of the **+** operator within the **toString()** method for the **Person** class:

```
int age = 4;
String first = "Kara";
String last = "Lot";
System.out.println(age + " year old Person named " + first + " " + last);
```

Here, we are actually combining 5 **String** objects to form a new **String** object containing the result ... the original 5 String objects remain unaltered.

Each character in a **String** is assigned an imaginary integer index that represents its order in the sequence. The first character in the **String** has an index of 0, the second character has an index of 1, and so on. We can access any character from a **String** by using the **charAt()** method which requires us to specify the index of the character that we want to get:

```
String name = "Hank Urchif";
name.charAt(0);           // returns character 'H'
name.charAt(1);           // returns character 'a'
name.charAt(name.length() - 1); // returns character 'f'
name.charAt(name.length()); // causes StringIndexOutOfBoundsException
name.charAt(100);        // causes StringIndexOutOfBoundsException
```



There are also some methods in the **String** class that allow us to extract a sequence (or range) of characters from the String. The **substring(s,e)** method does just that. It takes two parameters **s** and **e**, where **s** specifies the starting character index and **e** specifies one more than the ending character index:

```
String name = "Hank Urchif";
name.substring(0, 4);           // returns character "Hank"
name.substring(5, 11);          // returns character "Urchif"
name.substring(1, name.length()); // returns character "ank Urchif"
name.substring(3, 6);           // returns character "k U"
```

In all cases above, the resulting **String** is a new object, the original **name** object remaining unchanged.



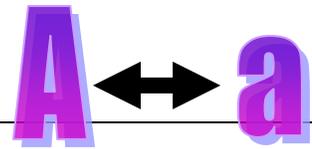
There is also a very useful method for eliminating unwanted leading and trailing characters (e.g., spaces, tabs, newlines and carriage returns). This can be useful when writing programs that get String input (e.g., name, address, email etc..) from the user through text fields on windows. The **trim()** method returns a new **String** object that represents the original string object but with no leading and trailing space, tab, newline or carriage return characters.

```
String s1 = "  I need a shave  ";
String s2 = "          ";
s1.trim(); // returns "I need a shave"
s2.trim(); // returns empty string ""
```



Also, sometimes when getting input from the user we would like to force the information to be formatted as either uppercase or lowercase characters. Two useful methods called **toUpperCase()** and **toLowerCase()** will generate a copy of the string but with all alphabetic characters converted to uppercase or lowercase, respectively. The methods only affect the alphabetic characters ... all other characters remain the same.

```
String s = "Tea For 2!";
s.toUpperCase(); // returns "TEA FOR 2!"
s.toLowerCase(); // returns "tea for 2!"
```



A final important topic that we will discuss regarding strings is that of comparing strings with one another. String comparison is a fundamental tool used in many programs. For example, whenever we want to search for a person's name in a list, we must compare the name of the person (i.e., a **String**) with all of the names in a list of some sort.

JAVA has two useful methods for comparing Strings. The **equals(s)** method compares one **String** with another **String, s**, and then returns **true** if the two strings have the exact same characters in them and **false** otherwise. A similar comparison method called **equalsIgnoreCase(s)** is used to compare the two strings but in a way such that lowercase and uppercase characters are considered equal.

```
String apple1 = "apple";
String apple2 = "APPLE";
String apple3 = "apples";
String orange = "orange";

apple1.equals(orange); // returns false
apple1.equals(apple2); // returns false
apple1.equals(apple3); // returns false
apple1.equals(apple2.toLowerCase()); // returns true
apple1.equalsIgnoreCase(apple2); // returns true
```



In regards to sorting strings, the **compareTo(s)** method will compare one string with another (i.e., parameter **s**) and return information about their respective alphabetical ordering. The method returns an integer which is:

- negative if the first string is *alphabetically before s*
- positive if the first string is *alphabetically after s*, or
- zero if the first string equals **s**

```
String apple = "Apple";
String orange = "Orange";
String banana = "Banana";

banana.compareTo(orange); // returns -13, Banana comes before Orange
banana.compareTo(apple); // returns 1, Banana comes after Apple
apple.compareTo("Apple"); // returns 0, Apple equals Apple
"Zebra".compareTo("apple"); // returns -7, uppercase chars are before lower!
"apple".compareTo("Apple"); // returns 32, lowercase chars are after upper!
```



You may notice, in the last two cases, that uppercase characters always come alphabetically before lowercase characters. You should always take this into account when sorting data.

To avoid sorting problems, it may be best to use `toUpperCase()` on each **String** before comparing them:

```
if (s1.toUpperCase().compareTo(s2.toUpperCase()) < 0)
    // s1 comes first
else
    // s2 comes first
```

Another very useful method in the **String** class is the `split()` method because it allows you to break up a **String** into individual substrings (called *tokens*) based on some separation criteria. For example, we can extract

- words from a sentence, one by one
- fields from a database or text file, separated by commas or other chars

The term **delimiter** is used to indicate the character(s) that separate the tokens (i.e., individual words or data elements).

Consider for example, the following String data which has been read in from a file:

```
"Mark,Lanthier,47,M,false"
```

Perhaps this is data for a particular person and we want to extract the information from the string in a way that we can use it. If we consider the comma ',' character as the only delimiter, then we can use the `split` method to obtain an array of Strings which we can then parse one by one to extract the needed data:

```
String s1 = "Mark,Lanthier,47,M,false";

String[] tokens = s1.split(",");
for(int i=0; i<tokens.length; i++)
    System.out.println(tokens[i]);
```



The code above will produce the following output:

```
Mark
Lanthier
47
M
false
```

Each token is an individual String that can be used afterwards. If, for example, we wanted to have just the 3<sup>rd</sup> piece of data (i.e., 47) and use it in a math expression, we could split the string and access just that piece of data, converting it to an integer as necessary ...

```
String s1 = "Mark,Lanthier,47,M,false";
String[] tokens;
int age;

tokens = s1.split(",");
age = Integer.parseInt(tokens[2]);
if (age > 21) ...
```

The `,` parameter to the `split()` method above indicates that the `,` character is the delimiter. If we had the following String, however, we may want to include the `:` character as a delimiter as well:

```
"Mark,Lanthier:47:M,false"
```

We cannot simply use the parameter string `,:` because that will only consider consecutive comma colon characters as delimiters (i.e., a 2-char delimiter). We want to allow the comma OR the colon to be delimiters, but not necessarily together. To accomplish this, the expression in the string becomes more complex. We basically have to indicate that we want all non-alphanumeric characters to be part of the tokens and everything else to be delimiters. So the following code would do what we want:

```
String s1 = "Mark,Lanthier:47:'M',false";

String[] tokens = s1.split("[^a-zA-Z0-9]");
for(int i=0; i<tokens.length; i++)
    System.out.println(tokens[i]);
```

Notice the square brackets `[]` in the parameter string. This indicates that we are about to list a sequence of characters to be the delimiters. The `^` character negates the list of characters to indicate that we are about to list all the non-delimiter characters (i.e., the token characters). Then we list the alphanumeric ranges `a-z`, `A-Z` and `0-9` to indicate that any alphanumeric character is part of a token, while everything else is to be considered a delimiter.

The parameter string is considered to be a **regular expression** (not discussed here) and can be quite complex. You may look in JAVA's API for more information. In some cases, the token strings will be of size 0. For example, consider the following code:

```
String s1 = "Mark, Lanthier , 47 ,,, M , false";

String[] tokens = s1.split("[, ]"); // comma or space delimiter
for(int i=0; i<tokens.length; i++)
    System.out.println(tokens[i]);
```

The following output would be obtained ...

```
Mark
Lanthier
47
M
false
```

Notice that there are many spaces in between. These spaces are empty strings. We should check for the empty strings in our code:

```
String s1 = "Mark, Lanthier , 47 ,,, M , false";

String[] tokens = s1.split("[, ]"); // comma or space delimiter
for(int i=0; i<tokens.length; i++)
    if (tokens[i].length() > 0)
        System.out.println(tokens[i]);
```

Then we obtain the output as before:

```
Mark
Lanthier
47
M
false
```

## Supplemental Information (StringTokenizers)

There is another (perhaps simpler) way of extracting tokens from a **String** through use of the **StringTokenizer** class (imported from the **java.util** package). However, for some reason, the JAVA guys “suggest” that you use the **split()** method instead.

```
String s = "Mark, Lanthier , 47 ,,, M , false";

StringTokenizer tokens = new StringTokenizer(s, ", ");
System.out.println("The string has " + tokens.countTokens() + " tokens");

while(tokens.hasMoreTokens()) {
    System.out.println(tokens.nextToken());
}
```

This code will produce the same result as above, but with an extra line of output indicating the number of tokens in total, which is 5 in this example.

Interestingly, the **Scanner** class that we used for getting keyboard input can also be used to get tokens from a **String**. The list of delimiters however is actually a pattern sequence, not a list of separate delimiter characters. That means, whatever is listed as the delimiter string must match exactly (i.e., in the example below, a single comma must be followed by a single space character):

```
String sentence = "Banks, Rob, 34, Ottawa, 12.67";
Scanner s = new Scanner(sentence).useDelimiter(", ");
System.out.println(s.next());
System.out.println(s.next());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.nextFloat());
s.close();
```

Notice that the **Scanner** should be closed, we did not do this earlier but it is common practice.

## 9.2 Enumerated Types

Consider writing a program that evaluates certain code depending on the day of the week:

```
String dayOfWeek = ...;

if (dayOfWeek.equals("SATURDAY") || dayOfWeek.equals("SUNDAY"))
    // do something ...
else
    // do something else ...
```

Each time we make a **String** comparison using the **equals()** method, this takes time because JAVA needs to compare the characters from the strings. However, comparing integers is fast. Alternatively, we could define some integer constants for the days of the week ... store our day of the week as an integer ... and then and simply compare the constants using **==** as follows:

```
public static final int MONDAY = 1;
public static final int TUESDAY = 2;
public static final int WEDNESDAY = 3;
public static final int THURSDAY = 4;
public static final int FRIDAY = 5;
public static final int SATURDAY = 6;
public static final int SUNDAY = 7;

...
...
int dayOfWeek = ...;

if ((dayOfWeek == SATURDAY) || (dayOfWeek == SUNDAY))
    // do something ...
else
    // do something else ...
```



This comparison is very fast since it merely compares two integers as opposed to a sequence of characters. We can use this more efficient strategy whenever we have a set of fixed constant values such as the days of the week, planets in the solar system, choices on a menu, command-line flags, etc.. However, rather than clutter up our code with these constants, we can define them in their own publicly accessible class as follows ...

```
public class Day {
    public static final int MONDAY = 1;
    public static final int TUESDAY = 2;
    public static final int WEDNESDAY = 3;
    public static final int THURSDAY = 4;
    public static final int FRIDAY = 5;
    public static final int SATURDAY = 6;
    public static final int SUNDAY = 7;
}
```

Then we can make use of this class in our code:

```
...
int dayOfWeek = ...;

if ((dayOfWeek == Day.SATURDAY) || (dayOfWeek == Day.SUNDAY))
    // do something ...
else
    // do something else ...
```

Notice the manner in which we access these constants in the **Day** class by specifying the **Day** class name and the dot operator to go into that class to get the requested constant value.

JAVA provides a useful keyword called **enum** that can be used for this exact situation. The **enum** keyword can be used in place of the **class** keyword to define a set of constant symbols such as the days of the week. It makes the code much simpler:

```
public enum Day {  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY  
}
```

I'm sure you will agree that this is much shorter than the class definition. In fact, we can define as many of these enumerated types as we would like ... each in their own file:

```
public enum Gender {  
    MALE, FEMALE  
}
```

```
public enum Size {  
    TINY, SMALL, MEDIUM, LARGE, HUGE  
}
```

```
public enum Direction {  
    NORTH, NORTH_EAST, EAST, SOUTH_EAST, SOUTH,  
    SOUTH_WEST, WEST, NORTH_WEST  
}
```

These would be saved, compiled and then used in our programs just as we would use any other class or interface definition. Once the type is defined, we can even define variables with these types, assigning them one of the values in the list specified in the **enum** declaration:

```
Gender    gender;  
Size      orderSize;  
Day       payday;  
Direction homePosition;  
  
gender = Gender.MALE;  
orderSize = Size.MEDIUM;  
payday = Day.FRIDAY;  
homePosition = Direction.NORTH_WEST;
```

Notice that the variable type matches the **enum** type. Also notice that when assigning a value to the variable, we must again specify the **enum** type followed by the dot **.** operator and then by one of the fixed constant values.

If we wanted to, we could have defined the **enum** types right inside of our class (but then they may not be declared **public** and would only be useable within that class's code). It would however, allow us to reduce the number of classes that we write. To use the **enum** types in your own class definition, you should place them directly under the class definition (or above it). You are not allowed to place the declarations in the executable (i.e., running) part of your program (i.e., in the main method, nor in other functions or procedures). Note that when you display an **enum** type value (as in the code above), the symbol value is printed out just as it appears in the type declaration.

```
public class EnumTestProgram {

    enum Gender {MALE, FEMALE};
    enum Size {TINY, SMALL, MEDIUM, LARGE, HUGE};
    enum Day {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY};
    enum Direction {NORTH, NORTH_EAST, EAST, SOUTH_EAST,
                   SOUTH, SOUTH_WEST, WEST, NORTH_WEST};

    public static void main(String args[]) {
        Gender    gender = Gender.MALE;
        Size      orderSize = Size.MEDIUM;
        Day       payday = Day.FRIDAY;
        Direction homePosition = Direction.NORTH_WEST;

        System.out.println(gender);           // displays MALE
        System.out.println(orderSize);       // displays MEDIUM
        System.out.println(payday);         // displays FRIDAY
        System.out.println(homePosition);   // displays NORTH_WEST
    }
}
```

## 9.3 The StringBuilder & Character Classes

Strings cannot be changed once created. Instead, when we try to manipulate them, we always get back a "brand new" **String** object. This is not normally a problem in most cases when programming, however, sometimes we would like to be able to *modify* a **String** by inserting/removing characters. For example, when we open a file in a text editor or word processor, we usually append, cut and insert text "on the fly". It would be memory-inefficient and time-inefficient to continually create new strings and copy over characters from an old string to a new one.



The **StringBuilder** class in JAVA is useful for such a purpose. You may think of it simply as a **String** that can be modified, although it is not a **String**. The **StringBuilder** methods run a little slower than their **String**-equivalent methods, so if you plan to create strings that will not need to change, use **String** objects instead.

Here are two constructors for the **StringBuilder** class:

```
new StringBuilder();  
new StringBuilder(s); // s is a String
```

The first creates a **StringBuilder** with no characters to begin with and the second creates one with the characters equal to the ones in the given **String s**.

As with **Strings**, the **length()** method can be used to return the number of characters in the **StringBuilder** as follows:

```
StringBuilder sb1, sb2;  
  
sb1 = new StringBuilder();  
sb2 = new StringBuilder("hello there");  
sb1.length(); // returns 0  
sb2.length(); // returns 11
```

Unlike **Strings**, you can actually modify the length of the **StringBuilder** to any particular length by using a **setLength(int newLength)** method. If the **newLength** is less than the current length, the characters at the end of the **StringBuilder** are truncated. If the size is greater, **null** characters are used to fill in the extra places at the end as follows:

```
StringBuilder sb;  
  
sb = new StringBuilder("hello there");  
sb.setLength(9);  
System.out.println(sb); // displays "hello the"
```

As with **Strings**, the **charAt(int index)** method is used to access particular characters based on their **index** position (which starts at position 0). Unlike **Strings** though, a **setCharAt(int index, char c)** method is also available which allows you to change the character at the given **index** to become the specified character **c**.

Here is how these methods work ...

```
StringBuilder name;  
  
name = new StringBuilder("Chip Electronic");  
name.charAt(3); // returns 'p'  
name.setCharAt(4, '+');  
System.out.println(name); // displays "Chip+Electronic"
```

However, a more commonly used method in the **StringBuilder** class is the **append(Object x)** method which allows you to append a bunch of characters to the end of the **StringBuilder**. If **x** is a **String** object, the entire string is appended to the end. If **x** is any other object, JAVA will call the **toString()** method for that object and append the resulting **String** to the end of the **StringBuilder**:

```
StringBuilder sb = new StringBuilder();

sb.append("People: \n");
sb.append(new Person("Holly Day", 15, 5.6, 'F', false));
sb.append("\n");
sb.append(new Person("Hank Urchiff", 89, 5.4, 'M', true));
sb.append("\n");

System.out.println(sb);
```

Here would be the output :

```
People:
15 year old non-retired person named Holly Day
89 year old retired person named Hank Urchiff
```

The resulting output may differ, of course, depending on the **Person** class's **toString()** method. Similar methods also exist for appending an **int**, **long**, **float**, **double**, **boolean** or **char** as follows:

```
append(int x), append(long x), append(float x),
append(double x), append(boolean x), append(char x)
```

The final two methods that we will mention allow you to remove characters from the **StringBuilder**. The **deleteCharAt(int index)** method will remove the character at the given **index** while the **delete(int start, int end)** method will delete all the characters within the indices ranging from **start** to **end-1** as follows:

```
StringBuilder sb;

sb = new StringBuilder("Miles Pergalon");
sb.delete(3,11); // changes sb to "Millon"
sb.deleteCharAt(sb.length()-1); // changes sb to "Millo"
sb.deleteCharAt(sb.length()-1); // changes sb to "Mill"
```

Sometimes, it is useful to use a **StringBuilder** to go through a **String** and make changes to it. For example, consider using a **StringBuilder** to remove all the non-alphabetic characters from a **String** as follows (of course the result would have to be a new **String**, since the original cannot be modified) ...

```

String          original, result = "";
StringBuilder   sb;
Character       c;

original = "Hello, my 1st name ... is Mark !!";
sb = new StringBuilder();
for (int i=0; i<original.length(); i++) {
    c = original.charAt(i);
    if (Character.isLetter(c))
        sb.append(c);
}
result = new String(sb);
System.out.println(result);

```

Notice a couple of things from this code. First, the **StringBuilder** object is used as a temporary object for creating the result string but is no longer useful after the method has completed. We use one of the **String** class' constructors to create the new **String** ... passing in the **StringBuilder** object. Second, we are checking for non-alphabetic characters by using **Character.isLetter()**. Here, **isLetter()** is a **static** function in the **Character** class that determines whether or not the given character is alphabetic or not.



**Side note:** **Character** is a class in JAVA known as a **wrapper class** because it is an **object wrapper** for the **char** primitive. Essentially, the class can be used to “convert” (i.e., wrap up) a **char** into an object that can then be used as a regular object. There is a wrapper class for each of the primitives in JAVA (i.e., **Integer**, **Long**, **Float**, **Double**, **Character**, **Boolean**, **Short** and **Byte**). Since JAVA 1.5, primitives are automatically wrapped into objects, and so we need not worry about this.

There are other useful methods in the **Character** class. Here are just a few:

```

Character.isLetter(c)           // checks if c is a letter in the alphabet
Character.isDigit(c)           // checks if c is a digit (i.e., '0' - '9')
Character.isLetterOrDigit(c)   // ... this one is obvious ...
Character.isWhiteSpace(c)      // checks if c is the space character
Character.isLowerCase(c)       // checks if c is lowercase (e.g., 'a')
Character.isUpperCase(c)       // checks if c is uppercase (e.g., 'A')
Character.toLowerCase(c)       // returns lowercase equivalent of c
Character.toUpperCase(c)       // returns uppercase equivalent of c

```

Here are some examples of how they are used:

```

Character.isLetter('A')         // returns true
Character.isDigit('6')         // returns true
Character.isLetterOrDigit('@') // returns false
Character.isWhiteSpace(' ')    // returns true
Character.isLowerCase('a')     // returns true
Character.isUpperCase('A')     // returns true
Character.toLowerCase('B')     // returns 'b'
Character.toUpperCase('b')     // returns 'B'

```

Note that none of these methods require you to make an instance of a **Character** object. They are all **static**/class methods that take a **char** as a parameter (**int** in some cases) and return another primitive.

## 9.4 The Date and Calendar Classes

It is often necessary to use dates and times when programming. Let us take a look at the **Date** class provided in the **java.util** package. The **Date** class allows us to make data objects that incorporate time as well. The **java.util.Date** class is used to represent BOTH date and time. Dates are stored simply as a number, which happens to be the number of milliseconds since January 1, 1970, 00:00:00 GMT.



New dates are created with a call to a constructor as follows:

```
Date    today = new Date();
```

The result is an object that represents the current date and time and it looks something like this when displayed (of course it will vary depending on the day you run your code):

```
Wed Nov 18 14:10:17 EDT 2015
```

Notice that it shows the **day**, **month**, **day-of-month**, **hours**, **minutes**, **seconds**, **timezone** and **year** of the **Date** object. This is default behavior for this class. There are only three other useful methods in the **Date** class:

- **getTime()** - Returns a **long** representing this time in milliseconds.
- **after(Date d)** - Returns whether or not a date comes after the given date **d**.
- **before(Date d)** - Returns whether or not a date comes before the given date **d**.



Most other methods have been **deprecated** (which means they should not be used anymore).

In the class **Date** itself, there is no easy way to create a specific date (e.g., Feb. 13, 1998). Instead, we must use a different class to do this. In the current version of JAVA, **Calendar** objects are used to represent dates, instead of **Date** objects. **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.



Although this **Calendar** class has many useful constants and methods (as you will soon see), we cannot make instances of it (i.e., we cannot say **new Calendar()**). Instead, the more specific kind of calendar called a **GregorianCalendar** is used.

The **java.util.GregorianCalendar** class is used to query and manipulate dates. Here are some of the available constructors ...

```
new GregorianCalendar()           // today's date
new GregorianCalendar(1999, 11, 31) // year, month, day
new GregorianCalendar(1968, 0, 8, 11, 55) // year, month, day, hours, mins
```

Notice that:

- the year is specified as 4-digits (e.g., 1968)
- months are specified from 0 to 11 (January being 0)
- days must be from 1 to 31
- hours and minutes are at the end of the constructor

Calendars do not display well. Here is what you would see if you tried displaying a **GregorianCalendar**:

```
java.util.GregorianCalendar[time=1178909251343,areFieldsSet=true,
areAllFieldsSet=true,lenient=true,zone=sun.util.calendar.ZoneInfo[id=
"America/New_York",offset=-18000000,dstSavings=3600000,useDaylight=true,
transitions=235,lastRule=java.util.SimpleTimeZone[id=America/New_York,
offset=-18000000,dstSavings=3600000,useDaylight=true,startYear=0,
startMode=3,startMonth=3,startDay=1,startDayOfWeek=1,startTime=7200000,
startTimeMode=0,endMode=2,endMonth=9,endDay=-1,endDayOfWeek=1,endTime=
7200000,endTimeMode=0]],firstDayOfWeek=1,minimalDaysInFirstWeek=1,ERA=1,
YEAR=2007,MONTH=4,WEEK_OF_YEAR=19,WEEK_OF_MONTH=2,DAY_OF_MONTH=11,
DAY_OF_YEAR=131,DAY_OF_WEEK=6,DAY_OF_WEEK_IN_MONTH=2,AM_PM=1,HOUR=2,
HOUR_OF_DAY=14,MINUTE=47,SECOND=31,MILLISECOND=343,ZONE_OFFSET=
-18000000,DST_OFFSET=3600000]
```



Obviously, this is not pleasant. To display a **Calendar** in a friendlier manner, we must use the **getTime()** method, which actually returns a **Date** object (... not very intuitive ... I know). Consider these examples:

```
System.out.println(new GregorianCalendar().getTime()); // today
System.out.println(new GregorianCalendar(1999,11,31).getTime());
System.out.println(new GregorianCalendar(1968,0,8,11,55).getTime());
```

Here is the output (which of course varies with the current date):

```
Wed Nov 18 14:10:40 EDT 2015
Fri Dec 31 00:00:00 EST 1999
Mon Jan 08 11:55:00 EST 1968
```



The **isLeapYear(int year)** method returns whether or not the given year is a leap year for this calendar:

```
new GregorianCalendar().isLeapYear(2016); // returns true
new GregorianCalendar().isLeapYear(2015); // returns false
```



There are many other methods that we can use to query or alter the date which are inherited from the **Calendar** class.

For example, the **get(int field)** method is used along with some **static** constants to access information about the particular calendar date. For example, at the time of updating these notes the date was:

```
Tue Nov 17 23:44:10 EDT 2015
```

Consider the results (shown to the right) of each **get** method call in the code below. You should use **import java.util.Calendar** at the top of your code so that you can use these constants:

```
Calendar today = Calendar.getInstance();

today.get(Calendar.YEAR);           // 2015
today.get(Calendar.MONTH);         // 10
today.get(Calendar.DAY_OF_MONTH);  // 17
today.get(Calendar.DAY_OF_WEEK);   // 3
today.get(Calendar.DAY_OF_WEEK_IN_MONTH); // 3
today.get(Calendar.DAY_OF_YEAR);   // 321
today.get(Calendar.WEEK_OF_MONTH); // 3
today.get(Calendar.WEEK_OF_YEAR);  // 47
today.get(Calendar.HOUR);          // 11
today.get(Calendar.AM_PM);         // 1
today.get(Calendar.HOUR_OF_DAY);   // 23
today.get(Calendar.MINUTE);        // 44
today.get(Calendar.SECOND);        // 10
```

The value returned from the **get(int field)** method can be compared with other **Calendar** constants. For example,

```
if (aCalendar.get(Calendar.MONTH) == Calendar.APRIL) {...}
if (aCalendar.get(Calendar.DAY_OF_WEEK) == Calendar.SATURDAY) {...}
```

Here are some of the useful constants:

Calendar.SUNDAY Calendar.MONDAY Calendar.TUESDAY Calendar.WEDNESDAY Calendar.THURSDAY Calendar.FRIDAY Calendar.SATURDAY	Calendar.JANUARY Calendar.FEBRUARY Calendar.MARCH Calendar.APRIL Calendar.MAY Calendar.JUNE Calendar.AM	Calendar.JULY Calendar.AUGUST Calendar.SEPTEMBER Calendar.OCTOBER Calendar.NOVEMBER Calendar.DECEMBER Calendar.PM	
---	---	---	---

There is also a **set(int field, int value)** method that can be used to set the values for certain date fields:

```
aCalendar.set(Calendar.MONTH, Calendar.JANUARY);
aCalendar.set(Calendar.YEAR, 1999);
aCalendar.set(Calendar.AM_PM, Calendar.AM);
```

Other set methods allow the date and time to be changed ...

```
aCalendar.set(1999, Calendar.AUGUST, 15);
aCalendar.set(1999, Calendar.AUGUST, 15, 6, 45);
```



We can also format dates when we want to print them nicely. There is a **SimpleDateFormat** class (in the **java.text** package) that formats a **Date** object using one of many predefined formats. It does this by generating a **String** representation of the date. The constructor takes a **String** which indicates the desired format:

```
new SimpleDateFormat("MMM dd,yyyy");
```

The parameter in the method is a format string that specifies “how you want the date to look” when it is printed. By using different characters in the format string, you get different output for the date. The **format(Date d)** method in the **SimpleDateFormat** class is then used to actually do the work by applying the format to the given date. Here is an example:

```
import java.text.SimpleDateFormat;
// ...

SimpleDateFormat dateFormatter = new SimpleDateFormat("MMM dd,yyyy");
Date today = new Date();
String result = dateFormatter.format(today);

System.out.println(result);
```

Here is the result (which would vary, depending on the date):

```
Nov 18, 2015
```

Here are examples of format Strings and their effect on the date April 30th 2001 at 12:08 PM:

Format String	Resulting output
without formatting	Tue Apr 10 15:07:52 EDT 2001
"yyyy/MM/dd"	2001/04/30
"yy/MM/dd"	01/04/30
"MM/dd"	04/30
"MMM dd,yyyy"	Apr 30, 2001
"MMMM dd,yyyy"	April 30, 2001
"EEE. MMMM dd,yyyy"	Mon. April 30, 2001
"EEEE, MMMM dd,yyyy"	Monday, April 30, 2001
"h:mm a"	12:08 PM
"MMMM dd, yyyy (hh:mm a)"	April 30, 2001 (12:08PM)

For additional formatting information, check out the JAVA API specification. Here is a simple example that creates two dates. One representing today, the other representing a future date:

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

public class DateTestProgram {
    public static void main (String args[]) {

        Calendar today = Calendar.getInstance();
        Calendar future;
        int difference;
```

```

// Display Information about today's date and time
System.out.println("Here is today:");
System.out.println(today.getTime());
System.out.println(today.get(Calendar.YEAR));
System.out.println(today.get(Calendar.MONTH));
System.out.println(today.get(Calendar.DAY_OF_MONTH));

// Display Information about a future day's date and time
future = Calendar.getInstance();
future.set(2025, Calendar.MARCH, 5);
System.out.println("Here is the future:");
System.out.println(future.getTime());
System.out.println(future.get(Calendar.YEAR));
System.out.println(future.get(Calendar.MONTH));
System.out.println(future.get(Calendar.DAY_OF_MONTH));

// Test the formatting
Date aDate = new Date();
System.out.println(aDate);
System.out.println(new SimpleDateFormat("yyyy/MM/dd").format(aDate));
System.out.println(new SimpleDateFormat("yy/MM/dd").format(aDate));
System.out.println(new SimpleDateFormat("MM/dd").format(aDate));
System.out.println(new SimpleDateFormat("MMM dd, yyyy").format(aDate));
System.out.println(new SimpleDateFormat("MMMM dd, yyyy").format(aDate));
}
}

```

Here is the output from running this code on May 28<sup>th</sup>, 2015:

```

Here is today:
Thu May 28 13:54:36 EDT 2015
2015
4
28
Here is the future:
Wed Mar 05 13:55:30 EST 2025
2025
2
5
Thu May 28 13:54:36 EDT 2015
2015/05/28
15/05/28
05/28
May 28,2015
May 28,2015

```

Notice that the months start at **0**, and so May is month **4**.

Although we can create and display simple dates, we have not done any manipulation at all. For instance, we may want to know how many working days there are between two dates. There are many more functions in the **Calendar** and **Date** classes, but we will not discuss them any further here. You would have to look at the API for the **Date**, **Calendar**, **GregorianCalendar** and **SimpleDateFormat** classes.

## Supplemental Information (Formatting Dates with Strings)

We can also use the **String.format()** method to format dates and times. There are many flags that can be used (see the API for details) but here are some commonly used ones for displaying dates and times:

```
Date aDate = new Date();

System.out.println(String.format("%tc", aDate));
System.out.println(String.format("%tF", aDate));
System.out.println(String.format("%tR", aDate));
System.out.println(String.format("%tr", aDate));
System.out.println(String.format("%tD", aDate));
```

Here was the output when it was ran on May 28, 2015 at 1:57pm:

```
Thu May 28 13:57:26 EDT 2015
2015-05-28
13:57
01:57:26 PM
05/28/15
```