**Administrative**

**Today’s session**

Date conversions in format specifier

Date methods

Static members

Static mutator and accessor methods

equals and toString methods

**Session Topics**

**Date conversions in format specifier**

● The <format-specifier> may be used to format dates and times.

● Each date and time conversion code is preceded with “t”.

● Date and time conversion codes include:

| Converter | Purpose |
| --- | --- |
| Composite date | |
| tD | Format corresponding date as mm/dd/yy. |
| tF | Format corresponding date as yyyy-mm-dd. |
| tc | Format corresponding date as ddd mmm dd hh:mm:ss zzz yyyy |
| Composite time | |
| tR | Format corresponding time as 24-hour clock hh:mm. |
| tT | Format corresponding time as 24-hour clock hh:mm:ss. |
| tr | Format corresponding time as 12-hour clock hh:mm:ss am/pm. |
| Year | |
| tY | Format corresponding date as four-digit year. |
| ty | Format corresponding date as two-digit year. |
| Month | |
| tm | Format corresponding date as two-digit month. |
| tB | Format corresponding date as month name. |
| tb | Format corresponding date as three-character month name. |
| Day | |
| td | Format corresponding date as two-digit day. |
| te | Format corresponding date as one- or two-digit day. |
| tA | Format corresponding date as day-of-week name. |
| ta | Format corresponding date as three-character day-of-week name. |

● See **Date formats and methods** sample application on Blackboard.

**Date methods**

● Abstract class **Calendar** represents a calendar date.

● Class **GregorianCalendar** represents a calendar date.

● Class **LocalDateTime** represents a date without a time zone.

● Class **Date** represents a number of milliseconds since 1-Jan-1970, midnight.

● A date object created with:

✓ Default constructor **Date()** represents the current number of milliseconds since 1-Jan-1970, midnight.

✓ Constructor **Date(<number>)** represents the date <number> milliseconds since 1-Jan-1970, midnight.

● Class **Date** is mostly deprecated.

● Here are some Date and Calendar methods. See the complete list at [docs.oracle.com/javase/8/docs/api/java/util/Date.html](http://docs.oracle.com/javase/8/docs/api/java/util/Date.html) and [docs.oracle.com/javase/8/docs/api/java/util/Calendar.html](http://docs.oracle.com/javase/8/docs/api/java/util/Calendar.html)

respectively.

| Method | Purpose |
| --- | --- |
| Date and Calendar | |
| after(<test-date>) | ● Return true if date is after <test-date>.  ● Return false if date is after <test-date>. |
| before(<test-date>) | ● Return true if date is before <test-date>.  ● Return false if date is before <test-date>. |
| compareTo(<test-date>) | ● If this date is after <test-date>, return 1.  ● If the two dates are equal, return 0.  ● If this date is before <test-date>return -1. |
| Calendar | |
| add(<date-field>, <amount>) | Format corresponding date as month name. |
| getInstance() | Create a calendar object containing the current date and time. |

● See **Date formats and methods** sample application on Blackboard.

**Static members**

**Static members**

● A class member (field or method) may be declared **static**.

● A class member declared static means that there is one member *per class* while an application is running.

● A static member belongs to the class, not to any instance (object) of the class.

● Like instance members, static fields are generally declared private while static methods are declared public.

**Instance members**

● A class member not declared static is an instance member (field or method).

● A class member not declared static means that there is one member *per instance* while an application is running.

● A instance member belongs to its instance, not to the class.

**Static variable**

● A static variable belongs to its class, not to any object.

● A static variable is created when the application starts.

● No matter how many objects of the class that are created, there will only be one static variable created.

● A field should be made *static* if a value needs to be shared among all objects created from the class.

● Example of static and instance variables:

public class CounterClass

{

// Declare static variables

private static int sv1 = 1;

// there is one of these for each class

private static int sv2 = 2;

// there is one of these for each class

// Declare instance variables

private int iv1 = 33;

// there is one of these for each object

private int iv2 = 44;

// there is one of these for each object

// showCounts method

public void showCounts()

{

System.out.println("Static variable 1 is " + sv1);

System.out.println("Static variable 2 " + sv2);

System.out.println("Instance variable 1 " + iv1);

System.out.println("Instance variable 2 " + iv2);

}

// imIncrement method

public void imIncrement()

{

sv1 = sv1 + 1;

sv2 = sv2 + 1;

iv1 = iv1 + 1;

iv2 = iv2 + 1;

}

// smIncrement method

public static void smIncrement()

{

sv1 = sv1 + 1;

sv2 = sv2 + 1;

// Cannot reference instance variables within static methods

// iv1 = iv1 + 1;

// iv2 = iv2 + 1;

}

}

### Static method

● A static method belongs to its class, not to any object.

● A static method is created when the application starts.

● No matter how many objects of the class are created, there will only be one static method created.

● All objects from the class then share the static method.

● Reasons for making a method *static* include:

✓ The method doesn’t depend on an object.

✓ The method has generic code that could be shared among multiple objects.

✓ The method doesn’t directly refer to an instance variable.

### Instance and static references

● For fields and methods declared *in one class*, the following table summarizes how to reference them *in the same class*.

|  |  |  |
| --- | --- | --- |
| How fields and methods *in one class* are referenced *in the same class* | | |
| Class member referenced | In instance method | In static method |
| Instance variable | <var-name> | <obj-name>.<var-name>\* |
| Static variable | <var-name> | <var-name> |
| Instance method | <method-name> | <obj-name>.<method-name>\* |
| Static method | <method-name> | <method-name> |
| \* The object must be a local variable. | | |

● For fields and methods declared *in one class*, the following table summarizes how to reference them *in another class*.

|  |  |  |
| --- | --- | --- |
| How fields and methods *in one class* are referenced *in another class*  (assuming both classes are within the same package) | | |
| Class member referenced\* | In instance method | In static method |
| Instance variable | <obj-name>.<var-name> | <obj-name>.<var-name>\*\* |
| Static variable | <class-name>.<var-name> | <class-name>.<var-name> |
| Instance method | <obj-name>.<method-name> | <obj-name>.<method-name>\*\* |
| Static method | <class-name>.<method-name> | <class-name>.<method-name> |
| \* To be visible, each referenced member must be declared package-private (no modifier), protected, or public.  \*\* The object must be a local variable. | | |

● See **Instances and statics** sample application on Blackboard.

**Static mutator and accessor methods**

**Static mutator method**

● A **static mutator method** provides a way for the static variables of one class to be changed by other classes within an application.

● A mutator method is nearly always declared with the **public** modifier.

● There is typically one mutator method for each static variable.

● *Static setter method* example:

public static int setSongCount(int songCount)

{

Song.songCount = songCount;

}

● The class name enables the Java Virtual Machine to distinguish between a static variable and a local variable with the same name.

**Static accessor method**

● A **static accessor method** provides a way for the static variables of one class to be viewed by other classes within an application.

● An accessor method is nearly always declared with the **public** modifier.

● There is typically one accessor method for each static variable.

● *Static getter method* example:

public static double getTotalCost()

{

return totalCost;

}

**equals and toString methods**

● There are several “standard” methods included in each class including:

✓ Constructors\*

✓ Accessors/getters

✓ Mutators/setters

✓ equals\*

✓ toString\*

\* A default version of this method is automatically created by the Java Virtual Machine is one is not defined in a class.

***equals* method**

● The **equals method** tests whether one object is equal to another.

● The equals method typically compares the instance variables of one object to those of another.

● If a class does not specify an *equals* method, Java creates one that tests whether two object variables point to the same object.

***toString* method**

● The **toString method** typically lists the values stored in one or more instance variables of this object.

● If a class does not specify a *toString* method, Java creates one by default that simply lists the address of the object.

● Suppose there is a class without method toString defined and object **obj** has been instantiated from the class, then the following two statements will show the same output:

System.out.println("Class obj: " + obj);

System.out.println("Class obj: " + obj.toString());

● See **Classes – one file** sample application on Blackboard.

● See **Classes – two files** sample application on Blackboard.