

Fundamental Java Methods

These methods are frequently needed in Java classes. You can find a discussion of each one in any Java book, such as *Big Java* or *Thinking in Java*.

You should practice until you can write each of these methods without any effort.

T = the *name* of a *type* or *class*.

String getName() boolean isOn()	accessor method for a String attribute <i>name</i> accessor for a boolean attribute (<i>on</i>) begins with "is" not "get"
void setName(String name) void setOn(boolean on)	mutator ("setter") method: set the value of <i>name</i> . A mutator method can be trivial like: <pre>public void setName(String name) { this.name = name; }</pre> A mutator can perform <i>data validation</i> and convert a <i>synthetic attribute</i> into actual attributes. For example, to require that a Person's name is not null or empty: <pre>public void setName(String name) { if (name == null) throw new IllegalArgumentException("cannot be null"); if (name.isEmpty()) throw new IllegalArgumentException(...); this.name = name; }</pre>
String toString()	Return a string representation of the object suitable for printing. <i>This method does <u>not print anything</u> on System.out.</i> Example: <pre>public String toString() { return name+" "+id; }</pre>
@Override boolean equals(Object obj)	Test if two objects are equal in <i>value</i> . The parameter for this method should always be Object , not the class of this object. Use @Override (optional) to avoid typing errors. There is a 4-part pattern for writing equals: <pre>public boolean equals(Object obj) { // (1) verify that obj is not null if (obj == null) return false; // (2) test that obj is the same class as "this" object if (obj.getClass() != this.getClass()) return false; // (3) cast obj to this class's type Person other = (Person) obj; // (4) compare whatever values determine "equal" if (name.equalsIgnoreCase(other.name)) return true; return false; }</pre>
int hashCode()	hashCode() is used by HashSet, HashMap, and a few other collections to decide where to store the object in a collection.

	<p>If two objects are "equal", then the hashCode should be same. If <code>a.equals(b)</code> then <code>a.hashCode() == b.hashCode()</code>.</p> <p>Because hashCode determines which "bin" an object is put within a set or map, it should be something that normally doesn't change.</p> <p>See textbook for how to choose a good hash code.</p>
<pre>int compareTo(T obj)</pre>	<p>Defines an <i>ordering</i> of objects. Used for sort and binary search methods in <code>java.util.Arrays</code> and <code>java.util.Collections</code>. The semantics of this method are defined by the <i>Comparable</i> interface. <i>See example below.</i></p> <p><code>a.compareTo(b) < 0</code> if a is "less than" or "before" b <code>a.compareTo(b) = 0</code> if a has same order as b <code>a.compareTo(b) > 0</code> if a is "greater than" or "after" b</p> <p>Any positive or negative value can be returned by compareTo. Only the <u>sign</u> of the return value is important (+, -, or 0). To see this, try some Strings: <code>"ant".compareTo("dog")</code></p> <p>Be careful of null parameter values, esp. if the collection you want to sort may contain nulls.</p>
<pre>Object clone() == or == T clone()</pre>	<p>Make an identical copy of an object. If you implement this, then declare that the class implements <i>Cloneable</i>. Otherwise, calling <code>clone()</code> will throw <code>CloneNotSupportedException</code>. Usually <code>clone</code> performs a <i>deep copy</i>, <i>Horstmann 7.4</i>.</p>

Sorting and Comparable

The `compareTo` method is used for sorting and searching. Your class must declare that it has a `compareTo()` method by implementing the `Comparable` interface.

If your class has a `compareTo` method, then include this in your Java class:

```
/** Person objects can be sorted using compareTo */
public class Person implements Comparable<Person> {
    /** order Person objects by name. */
    public int compareTo(Person other) {
        if ( other == null ) return -1;

        // this calls compareTo of the String class, ignoring case of letters
        int comp = this.name.compareToIgnoreCase( other.name );
        return comp;
    }
}
```

If your class already has a `compareTo` but you want to order objects some other way, implement a *java.util.Comparator* which has a single method `compare(a, b)`. Its like an externalized `compareTo`.

Example

```
public class Person implements Cloneable, Comparable<Person> {
    private String name;
    private Date birthday;

    /** constructor initializes the attributes using parameters */
    public Person(String name, Date birthday) {
        this.name = name;
        this.birthday = new Date( birthday ); // copy the parameter value
                                              // because Date is mutable
    }

    /** accessor method for name (immutable) returns the name */
    public String getName( ) {
        return name;
    }

    /** accessor for birthday */
    public Date getBirthday( ) {
        return this.birthday; // or: return (Date) (birthday.clone())
    }

    /** Change the person's birthday.
     * @param birthday is birthday to assign to this person
     */
    public void setBirthday( Date birthday ) {
        // don't allow birthday to be null.
        if ( birthday == null )
            throw new IllegalArgumentException("must be born");
        this.birthday = new Date( birthday ); // because Date is mutable
    }

    /** two persons are equal if name *and* birthday are same */
    public boolean equals( Object obj ) {
        if ( obj == null ) return false;
        if ( this == obj ) return true; // this test is optional
        if ( this.getClass() != obj.getClass() ) return false;
        // cast obj to Person so we can get its attributes
        Person other = (Person) obj;
        // now test equality any way to want.
        return this.name.equals( other.name )
            && this.birthday.equals( other.birthday );
    }

    /** hashCode should be consistent with equals */
    public int hashCode( ) {
        int hash = 0;
        if (name != null) hash += name.hashCode(); // String hashCode
        if (birthday != null) hash += 37 * birthday.hashCode( );
        return hash;
    }

    /** compare people by name. Used for sorting. */
    public int compareTo( Person other ) {
        if ( other == null ) return -1;
        // this uses compareToIgnoreCase of the String class
        // this assumes that a Person's name is never null.
        return name.compareToIgnoreCase( other.name );
    }
}
```

```
/** clone makes a deep copy of an object.
 * It returns Object for compatibility with superclass,
 * but it is also legal to declare return type as Person.
 * @return a copy of this Person as a new object
 */
public Object clone( ) {
    Person clone = (Person)super.clone( ); // clone parent type first
    clone.name = name; // String is immutable, so sharing is OK
    clone.birthday = (Date)birthday.clone(); // clone mutable attribute
    return clone;
}
```

Exercises

1) Write a **toString** that returns the Person's name, a space, and birth date (but not time of day). To create a nicely formatted String, use `String.format()`. The format codes are given in the Javadoc for the `Formatter` class. `%s` formats a String; `%tF` and `%tD` are formats for a date. So you could use (try this in BlueJ to see the result):

```
String.format( "%s %tF", name, birthday )
```

2) `Date` objects are *mutable* (can be changed). Since `getBirthday()` returns a *reference* to the Person's birthday, we can use it to surreptitiously change a person's birthday!!

```
// Nok is born on 1 Jan 2000 ("Jan" = month 0)
Person nok = new Person("Nok", new Date(100, 0, 1) );
System.out.println("Nok = " + nok);
// get Nok's birthday.
Date date = nok.getBirthday( );
System.out.printf( "Nok was born on %tF\n", date);

// change the date object
date.setMonth( Calendar.JUNE );
date.setYear( 99 ); // this means 1999

// Did Nok's birthday change?
System.out.println( "Nok = " + nok);
System.out.printf( "Nok was born on %tF\n", nok.getBirthday() );
```

If protecting an object's attributes is important, `getBirthday()` should return a *copy* of the birthday using `birthday.clone()`. The downside of returning a copy is that it creates a new object each time.

3) Create an array of Person objects and sort them using `Arrays.sort()`.

```
Person [ ] people = new Person [4];
people[0] = new Person( "Nok", new Date(100, 3, 1) );
people[1] = new Person( "Maew", new Date(99, 1, 1) );
people[2] = new Person( "Ling", new Date(101, 2, 2) );
// two persons named "Nok" to test compare by birthday
people[3] = new Person( "Nok", new Date(100, 2, 15) );

System.out.println("Before sorting:");
// classic "for" loop over the array
for(int k=0; k<people.length; k++)
    System.out.println( people[k] );
java.util.Arrays.sort( people ); // sort using compareTo
System.out.println("\n\nAfter sorting:");
// a "for-each" loop that iterates over the same array
for( Person p : people ) System.out.println( p );
```

4) (Custom sorting) We also want to sort people by birthday using only the month and day!

But `Person` *already* has a `compareTo` method that orders `Person` objects by name.

No problem! `Arrays.sort` has another form like this:

```
Array.sort( T [] array, Comparator<T> comparator );
```

A **Comparator** is an object that compares two *other* objects -- for example, to compare two **Person**. **Comparator** is an interface in **java.util**. To write a **Comparator** you create a new class with a single method named **compare**. The **compare** method compares 2 parameters and returns an integer, similar to the way **compareTo** does, except **compare** uses parameters instead of "this". To write a **Comparator** you must implement this method:

```
compare( Person p1, Person p2 )
```

The **Comparator.compare** method returns a result of the comparison like this:

	< 0	if p1 has order "before" p2
compare(Person p1, Person p2)	= 0	if p1 and p2 have same order
	> 0	if p1 has order "after" p2

(a) Write a **BirthdayComparator** class that implements **Comparator<Person>** and write the **compare** method to order the objects by month and day of birthday.

```
import java.util.Comparator;

public class BirthdayComparator implements Comparator<Person> {
    public int compare( Person person1, Person person2 ) {
        //TODO check for person1 == null or person2 == null
        Date date1 = person1.getBirthday();
        Date date2 = person2.getBirthday();
        // compare months first.  if same then compare day.
        int comp = date1.getMonth() - date2.getMonth();
        if (comp == 0) comp = date1.getDate() - date2.getDate();
        return comp;
    }
}
```

(b) Test your **BirthdayComparator** by creating an instance of it and sort an array of **Person**.

```
Comparator<Person> comp = new BirthdayComparator( );
Arrays.sort( people, comp );
// print the array
System.out.println("People sorted by birthday");
for(Person p : people ) System.out.println( p );
```