Implementing compareTo

The `compareTo` method is the sole member of the `Comparable` interface, and is not a member of `Object`. However, it is quite similar in nature to `equals` and `hashCode`. It provides a means of fully ordering objects.

Implementing `Comparable` allows:

- calling `Collections.sort` and `Collections.binarySearch`
- calling `Arrays.sort` and `Arrays.binarySearch`
- using objects as keys in a `TreeMap`
- using objects as elements in a `TreeSet`

The `compareTo` method needs to satisfy the following conditions. These conditions have the goal of allowing objects to be fully sorted, much like the sorting of a database result set on all fields.

- anticommutation : `x.compareTo(y)` is the opposite sign of `y.compareTo(x)`
- exception symmetry : `x.compareTo(y)` throws exactly the same exceptions as `y.compareTo(x)`
- transitivity : `if x.compareTo(y)>0` and `y.compareTo(z)>0`, then `x.compareTo(z)>0` (and same for less than)
- if `x.compareTo(y)==0`, then `x.compareTo(z)` has the same sign as `y.compareTo(z)`
- consistency with `equals` is highly recommended, but not required : `x.compareTo(y)==0`, if and only if `x.equals(y)` ; consistency with `equals` is required for ensuring sorted collections (such as `TreeSet`) are well-behaved.

One can greatly increase the performance of `compareTo` by comparing first on items which are most likely to differ.

When a class extends a *concrete* `Comparable` class and adds a significant field, a correct implementation of `compareTo` cannot be constructed. The only alternative is to use composition instead of inheritance. (A similar situation holds true for `equals`. See *Effective Java* for more information.)

Compare the various types of fields as follows:

- numeric primitive : use `<` and `>`. There is an exception to this rule: `float` and `double` primitives should be compared using Float.compare(float, float) and Double.compare(double, double). This avoids problems associated with special border values. (Thanks to Roger Orr in the UK for pointing this out.)
- `boolean` primitive : use tests of the form `(x && !y)`
- `Object` : use `compareTo`. (Note that possibly-null fields present a problem : while `x.equals(null)` returns `false`, `x.compareTo(null)` will always throw a `NullPointerException`)
- type-safe enumeration : use `compareTo`, like any `Object`
- collection or array : `Comparable` does not seem to be intended for these kinds of fields. For example, `List`, `Map` and `Set` do not implement `Comparable`. As well, some collections have no definite order of iteration, so doing an element-by-element comparison cannot be meaningful in those cases.

If the task is to perform a sort of items which are stored in a relational database, then it is usually much preferred to let the database perform the sort using the ORDER BY clause, rather than in code.

An alternative to implementing `Comparable` is passing `Comparator` objects as parameters. Be aware that if a `Comparator` compares only one of several significant fields, then the `Comparator` is very likely not synchronized with `equals`.

All primitive wrapper classes implement `Comparable`. Note that `Boolean` did not implement `Comparable` until version 1.5, however.

Example

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

public final class Account implements Comparable<Account> {

  enum AccountType {CASH, MARGIN, RRSP};

   public Account (
      String aFirstName,
      String aLastName,
      int aAccountNumber,
      int aBalance,
      boolean aIsNewAccount,
      AccountType aAccountType
  ) {
      //..parameter validations elided
      fFirstName = aFirstName;
      fLastName = aLastName;
      fAccountNumber = aAccountNumber;
      fBalance = aBalance;
      fIsNewAccount = aIsNewAccount;
      fAccountType = aAccountType;
   }

  /**
  * @param aThat is a non-null Account.
  *
  * @throws NullPointerException if aThat is null.
  */
  @Override public int compareTo(Account aThat) {
    final int BEFORE = -1;
    final int EQUAL = 0;
    final int AFTER = 1;

    //this optimization is usually worthwhile, and can
    //always be added
    if (this == aThat) return EQUAL;

    //primitive numbers follow this form
```

```
    if (this.fAccountNumber < aThat.fAccountNumber) return BEFORE;
    if (this.fAccountNumber > aThat.fAccountNumber) return AFTER;

    //booleans follow this form
    if (!this.fIsNewAccount && aThat.fIsNewAccount) return BEFORE;
    if (this.fIsNewAccount && !aThat.fIsNewAccount) return AFTER;

    //objects, including type-safe enums, follow this form
    //note that null objects will throw an exception here
    int comparison = this.fAccountType.compareTo(aThat.fAccountType);
    if (comparison != EQUAL) return comparison;

    comparison = this.fLastName.compareTo(aThat.fLastName);
    if (comparison != EQUAL) return comparison;

    comparison = this.fFirstName.compareTo(aThat.fFirstName);
    if (comparison != EQUAL) return comparison;

    if (this.fBalance < aThat.fBalance) return BEFORE;
    if (this.fBalance > aThat.fBalance) return AFTER;

    //all comparisons have yielded equality
    //verify that compareTo is consistent with equals (optional)
    assert this.equals(aThat) : "compareTo inconsistent with equals.";

    return EQUAL;
  }

  /**
  * Define equality of state.
  */
  @Override public boolean equals(Object aThat) {
    if (this == aThat) return true;
    if (!(aThat instanceof Account)) return false;

    Account that = (Account)aThat;
    return
      ( this.fAccountNumber == that.fAccountNumber ) &&
      ( this.fAccountType == that.fAccountType ) &&
      ( this.fBalance == that.fBalance ) &&
      ( this.fIsNewAccount == that.fIsNewAccount ) &&
      ( this.fFirstName.equals(that.fFirstName) ) &&
      ( this.fLastName.equals(that.fLastName) )
    ;
  }

  /**
  * A class that overrides equals must also override hashCode.
  */
  @Override public int hashCode() {
    int result = HashCodeUtil.SEED;
    result = HashCodeUtil.hash( result, fAccountNumber );
    result = HashCodeUtil.hash( result, fAccountType );
```

```
    result = HashCodeUtil.hash( result, fBalance );
    result = HashCodeUtil.hash( result, fIsNewAccount );
    result = HashCodeUtil.hash( result, fFirstName );
    result = HashCodeUtil.hash( result, fLastName );
    return result;
  }

  //PRIVATE

  private String fFirstName; //non-null
  private String fLastName;  //non-null
  private int fAccountNumber;
  private int fBalance;
  private boolean fIsNewAccount;

  /**
  * Type of the account, expressed as a type-safe enumeration (non-null).
  */
  private AccountType fAccountType;

  /**
  * Exercise compareTo.
  */
  public static void main (String[] aArguments) {
    //Note the difference in behaviour in equals and compareTo, for nulls:
    String text = "blah";
    Integer number = new Integer(10);
    //x.equals(null) always returns false:
    System.out.println("false: " + text.equals(null));
    System.out.println("false: " + number.equals(null) );
    //x.compareTo(null) always throws NullPointerException:
    //System.out.println( text.compareTo(null) );
    //System.out.println( number.compareTo(null) );

    Account flaubert = new Account(
     "Gustave", "Flaubert", 1003, 0,true, AccountType.MARGIN
    );

    //all of these other versions of "flaubert" differ from the
    //original in only one field
    Account flaubert2 = new Account(
      "Guy", "Flaubert", 1003, 0, true, AccountType.MARGIN
    );
    Account flaubert3 = new Account(
      "Gustave", "de Maupassant", 1003, 0, true, AccountType.MARGIN
    );
    Account flaubert4 = new Account(
      "Gustave", "Flaubert", 2004, 0, true, AccountType.MARGIN
    );
    Account flaubert5 = new Account(
      "Gustave", "Flaubert", 1003, 1, true, AccountType.MARGIN
    );
    Account flaubert6 = new Account(
```

```
      "Gustave", "Flaubert", 1003, 0, false, AccountType.MARGIN
    );
    Account flaubert7 = new Account(
      "Gustave", "Flaubert", 1003, 0, true, AccountType.CASH
    );

    System.out.println( "0: " +  flaubert.compareTo(flaubert) );
    System.out.println( "first name +: " +  flaubert2.compareTo(flaubert) );
    //Note capital letters precede small letters
    System.out.println( "last name +: " +  flaubert3.compareTo(flaubert) );
    System.out.println( "acct number +: " +  flaubert4.compareTo(flaubert) );
    System.out.println( "balance +: " +  flaubert5.compareTo(flaubert) );
    System.out.println( "is new -: " +  flaubert6.compareTo(flaubert) );
    System.out.println( "account type -: " +  flaubert7.compareTo(flaubert) );
  }
}
```

A sample run of this class gives:

```
>java -cp . Account
false: false
false: false
0: 0
first name +: 6
last name +: 30
acct number +: 1
balance +: 1
is new -: -1
account type -: -1
```

In the older JDK 1.4, there are two differences :

- the type-safe version of the `Comparable` interface cannot be used. Instead, `Object` appears, along with a related cast operation
- `Boolean` objects must be treated differently from other wrapper classes, since `Boolean` did not implement `Comparable` until JDK 1.5.

Example:

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

public final class AccountOld implements Comparable {

  public AccountOld (
    String aFirstName,
    String aLastName,
    int aAccountNumber,
    int aBalance,
    boolean aIsNewAccount,
    AccountType aAccountType
  ) {
```

```
    //..parameter validations elided
    fFirstName = aFirstName;
    fLastName = aLastName;
    fAccountNumber = aAccountNumber;
    fBalance = aBalance;
    fIsNewAccount = aIsNewAccount;
    fAccountType = aAccountType;
  }

/**
* @param aThat is a non-null AccountOld.
*
* @throws NullPointerException if aThat is null.
* @throws ClassCastException if aThat is not an AccountOld object.
*/
public int compareTo(Object aThat) {
  final int BEFORE = -1;
  final int EQUAL = 0;
  final int AFTER = 1;

  //this optimization is usually worthwhile, and can
  //always be added
  if ( this == aThat ) return EQUAL;

  final AccountOld that = (AccountOld)aThat;

  //primitive numbers follow this form
  if (this.fAccountNumber < that.fAccountNumber) return BEFORE;
  if (this.fAccountNumber > that.fAccountNumber) return AFTER;

  //booleans follow this form
  if (!this.fIsNewAccount && that.fIsNewAccount) return BEFORE;
  if (this.fIsNewAccount && !that.fIsNewAccount) return AFTER;

  //Objects, including type-safe enums, follow this form.
  //Exception : Boolean implements Comparable in JDK 1.5, but not in 1.4
  //Note that null objects will throw an exception here.
  int comparison = this.fAccountType.compareTo(that.fAccountType);
  if ( comparison != EQUAL ) return comparison;

  comparison = this.fLastName.compareTo(that.fLastName);
  if ( comparison != EQUAL ) return comparison;

  comparison = this.fFirstName.compareTo(that.fFirstName);
  if ( comparison != EQUAL ) return comparison;

  if (this.fBalance < that.fBalance) return BEFORE;
  if (this.fBalance > that.fBalance) return AFTER;

  //all comparisons have yielded equality
  //verify that compareTo is consistent with equals (optional)
  assert this.equals(that) : "compareTo inconsistent with equals.";
```

```
    return EQUAL;
  }

  /**
  * Define equality of state.
  */
  public boolean equals(Object aThat) {
    if ( this == aThat ) return true;
    if ( !(aThat instanceof Account) ) return false;

    AccountOld that = (AccountOld)aThat;
    return
      ( this.fAccountNumber == that.fAccountNumber ) &&
      ( this.fAccountType == that.fAccountType ) &&
      ( this.fBalance == that.fBalance ) &&
      ( this.fIsNewAccount == that.fIsNewAccount ) &&
      ( this.fFirstName.equals(that.fFirstName) ) &&
      ( this.fLastName.equals(that.fLastName) );
  }

  /**
  * A class that overrides equals must also override hashCode.
  */
  public int hashCode() {
    int result = HashCodeUtil.SEED;
    result = HashCodeUtil.hash( result, fAccountNumber );
    result = HashCodeUtil.hash( result, fAccountType );
    result = HashCodeUtil.hash( result, fBalance );
    result = HashCodeUtil.hash( result, fIsNewAccount );
    result = HashCodeUtil.hash( result, fFirstName );
    result = HashCodeUtil.hash( result, fLastName );
    return result;
  }

  //PRIVATE

  private String fFirstName; //non-null
  private String fLastName;  //non-null
  private int fAccountNumber;
  private int fBalance;
  private boolean fIsNewAccount;

  /**
  * Type of the account, expressed as a type-safe enumeration (non-null).
  */
  private AccountType fAccountType;

  /**
  * Exercise compareTo.
  */
  public static void main (String[] aArguments) {
    //Note the difference in behaviour in equals and compareTo, for nulls:
    String text = "blah";
```

```
    Integer number = new Integer(10);
    //x.equals(null) always returns false:
    System.out.println("false: " + text.equals(null));
    System.out.println("false: " + number.equals(null) );
    //x.compareTo(null) always throws NullPointerException:
    //System.out.println( text.compareTo(null) );
    //System.out.println( number.compareTo(null) );

    AccountOld flaubert = new AccountOld(
      "Gustave", "Flaubert", 1003, 0, true, AccountType.MARGIN
    );

    //all of these other versions of "flaubert" differ from the
    //original in only one field
    AccountOld flaubert2 = new AccountOld(
      "Guy", "Flaubert", 1003, 0, true, AccountType.MARGIN
    );
    AccountOld flaubert3 = new AccountOld(
      "Gustave", "de Maupassant", 1003, 0, true, AccountType.MARGIN
    );
    AccountOld flaubert4 = new AccountOld(
      "Gustave", "Flaubert", 2004, 0, true, AccountType.MARGIN
    );
    AccountOld flaubert5 = new AccountOld(
      "Gustave", "Flaubert", 1003, 1, true, AccountType.MARGIN
    );
    AccountOld flaubert6 = new AccountOld(
      "Gustave", "Flaubert", 1003, 0, false, AccountType.MARGIN
    );
    AccountOld flaubert7 = new AccountOld(
      "Gustave", "Flaubert", 1003, 0, true, AccountType.CASH
    );

    System.out.println( "0: " +  flaubert.compareTo(flaubert) );
    System.out.println( "first name +: " +  flaubert2.compareTo(flaubert) );
    //Note capital letters precede small letters
    System.out.println( "last name +: " +  flaubert3.compareTo(flaubert) );
    System.out.println( "acct number +: " +  flaubert4.compareTo(flaubert) );
    System.out.println( "balance +: " +  flaubert5.compareTo(flaubert) );
    System.out.println( "is new -: " +  flaubert6.compareTo(flaubert) );
    System.out.println( "account type -: " +  flaubert7.compareTo(flaubert) );
  }
}
```

See Also :

Type-Safe Enumerations
Implementing equals
Implementing hashCode

Don't perform basic SQL tasks in code
Modernize old code

Would you use this technique?

Yes ○ No ○ Undecided ○ Vote