equals() in the class Object

The `Object` class implements a public `equals()` method that returns true iff the two objects are the same object.

That is:

`x.equals(y) == true` iff `x` and `y` are the same object

For some subclasses, this is adequate, especially for types for which the notion of an equality comparison doesn't really make practical sense.

Identity vs Equality

A deeper examination of the issue indicates there are two fundamentally distinct relationships at work, and that `Object equals()` conflates them:

identity

the relationship of being the same thing;

`x` is identical to `y` iff `x` and `y` are the same object;

in Java, this is tested by the operator `==`

equality

the relationship of having the same value;

`x` is equal to `y` iff `x` and `y`, in some useful sense, have equivalent content;

`x` and `y` may or may not be the same object;

in Java, this is tested by the `equals()` method

For many user-defined types, there are natural definitions of an equality relationship.

General Contract for equals()

The equals method implements an *equivalence relation* on non-null object references, `equals()` is:

- *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return `true`
- *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`
- *transitive*: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`

In addition:

- it is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in equals comparisons on the objects is modified.
- for any non-null reference value `x`, `x.equals(null)` should return false.

A User-defined Class

```
public class FileEntry {

   public Long   offset;   // offset of record in file
   public String record;   // record contents

   public FileEntry(long offset, String data) {

      this.offset = offset;
      this.record = data;
   }
   . . .
}
```

Here's a class that might be used in a program that accesses records from a file.

It's certainly possible we might create two different `FileEntry` objects from the same record, in which case the notion of equals is different from identity.

Standard equals() Features

We need to satisfy the general contract:

```
public class FileEntry {

   . . .

   public boolean equals(Object other) {

      // Make sure there really IS another object:
      if ( other == null ) return false;

      // Make sure it's of the correct type:
      if ( !this.getClass().equals(other.getClass()) )
         return false;
      . . .
   }
}
```

Specialized equals() Features

We need to implement a sensible definition of what equality means for this type:

```
public class FileEntry {

   . . .

   public boolean equals(Object other) {

      . . .
      // Get a reference of the appropriate type:
      FileEntry o = (FileEntry) other;

      // Perform the type-specific test for equality:
      return ( this.offset.equals(o.offset) );
   }
}
```

Complete Method

```
public class FileEntry {

   . . .

   public boolean equals(Object other) {

      // Make sure there really IS another object:
      if ( other == null ) return false;

      // Make sure it's of the correct type:
      if ( !this.getClass().equals(other.getClass()) )
         return false;

      // Get a reference of the appropriate type:
      FileEntry handle = (FileEntry) other;

      // Perform the type-specific test for equality:
      return ( this.offset.equals(handle.offset) );
   }
}
```

Issues with Overriding

Consider the following scenario:

```
public class
    prQuadtree< T extends TwoDComparable<? super T> > {
   . . .
   // calls equals() on the generic objects it stores
```

```
public interface TwoDComparable<T> {
   public long getX();
   public long getY();
}
```

The calls to `equals()` will bind to `Object equals()` because the Java compiler does not know what the actual type is going to be.

All that's known is that a `T` is-a-kind-of `TwoDcomparable<?>` and that doesn't guarantee a specialized implementation of `equals()`.

And so, the tree's search logic will be broken…

A Fix

If we add the `equals()` method to the interface that `T` must extend, all is well:

```
public interface TwoDComparable<T> {
   public long getX();
   public long getY();
   public boolean equals(Object other);
}
```

Now the compiler knows that whatever a `T` is, it must provide an `equals()` method.

And so, the tree's search logic will work…

A Debugging Hint

When in doubt, let your code talk to you:

```
public class FileEntry {
   . . .
   public boolean equals(Object other) {

      System.out.println("Call made to FileEntry.equals()");
      . . .
   }
}
```