Object common methods

equals

When your object fields can be `null`, implementing `Object.equals` can be a pain, because you have to check separately for `null`. Using `Objects.equal` lets you perform `equals` checks in a null-sensitive way, without risking a `NullPointerException`.

```
Objects.equal("a", "a"); // returns true
Objects.equal(null, "a"); // returns false
Objects.equal("a", null); // returns false
Objects.equal(null, null); // returns true
```

Note: The newly introduced `Objects` class in JDK 7 provides the equivalent `Objects.equals` method.

hashCode

Hashing all the fields of an `Object` should be simpler. Guava's `Objects.hashCode(Object...)` creates a sensible, order-sensitive hash for the specified sequence of fields. Use `Objects.hashCode(field1, field2, ..., fieldn)` instead of building the hash by hand.

Note: The newly introduced `Objects` class in JDK 7 provides the equivalent `Objects.hash(Object...)`.

toString

A good `toString` method can be invaluable in debugging, but is a pain to write. Use `MoreObjects.toStringHelper()` to easily create a useful `toString`. Some simple examples include:

```
// Returns "ClassName{x=1}"
MoreObjects.toStringHelper(this)
    .add("x", 1)
    .toString();

// Returns "MyObject{x=1}"
MoreObjects.toStringHelper("MyObject")
    .add("x", 1)
    .toString();
```

compare/compareTo

Implementing a `Comparator`, or implementing the `Comparable` interface directly, can be a pain. Consider:

```
class Person implements Comparable<Person> {
  private String lastName;
  private String firstName;
  private int zipCode;

  public int compareTo(Person other) {
    int cmp = lastName.compareTo(other.lastName);
    if (cmp != 0) {
      return cmp;
    }
    cmp = firstName.compareTo(other.firstName);
    if (cmp != 0) {
      return cmp;
    }
    return Integer.compare(zipCode, other.zipCode);
  }
}
```

This code is easily messed up, tricky to scan for bugs, and unpleasantly verbose. We should be able to do better.

For this purpose, Guava provides `ComparisonChain`.

`ComparisonChain` performs a "lazy" comparison: it only performs comparisons until it finds a nonzero result, after which it ignores further input.

```
  public int compareTo(Foo that) {
    return ComparisonChain.start()
        .compare(this.aString, that.aString)
        .compare(this.anInt, that.anInt)
        .compare(this.anEnum, that.anEnum, Ordering.natural().nullsLast())
        .result();
  }
```

This fluent idiom is much more readable, less prone to accidental typos, and smart enough not to do more work than it must. Additional comparison utilities can be found in Guava's "fluent Comparator" class `Ordering`, explained here.