```
  // mapping: "index" id -> Thrift-compatible data object
  public function evaluate($id) { ... }
}

class FQLTable {
  // a static list of contained fields:
  // mapping: () -> ('books' => 'FQLUserBooks', 'pic' ->'FQLUserPic', ...)
  public function get_fields() { ... }
}
```

The FQLField and FQLTable objects constitute this new method for accessing data. FQLField contains the data-specific logic transforming the index of the "row" (e.g., user ID) plus the viewer information (user and app_id) into our internal stack data calls. On top of that, we ensure privacy evaluation is built right in with the required can_see method. When processing a request, we create in memory one such FQLTable object for each named table ('user') and one FQLField object for each named field (one for 'books', one for 'pic', etc.). Each FQLField object mapped to by one FQLTable tends to use the same data accessor underneath (in the following case, user_get_info), though it is not necessary—it's just a convenient interface.
Example 6-18 shows an example of the typical string field for the user table.

EXAMPLE 6-18. Mapping a core data library to an FQL field definition

```
// base object for any simple FQL field in the user table.
class FQLStringUserField extends FQLField {

  public function __construct($user, $app_id, $table, $name) { ... }

  public function evaluate($id) {
    // call into internal function
    $info = user_get_info($id);
    if ($info && isset($info[$this->name])) {
      return $info[$this->name];
    }
    return null;
  }

  public function can_see($id) {
    // call into internal function
    return can_see($id, $user, $table, $name);
  }
}

// simple string data field
class FQLUserBooks extends FQLStringUserField { }

// simple string data field
class FQLUserPic extends FQLStringUserField { }
```