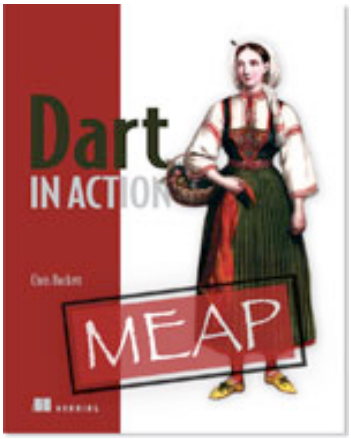
MANNING PUBLICATIONS



Dart in Action
By Chris Buckett

Overloading happens when you provide a new implementation customized to your particular class, which allows you to overload the common operators such as > (greater than) and < (less than). In this article, based on chapter 8 of Dart in Action, author Chris Buckett explains how this function helps you create truly self-documenting code by customizing the meaning built into the standard operators.

Operator Overloading

When Alice logs into a timesheet app, the system retrieves the `Role`s that represent the way Alice might use the timesheet system. For example, Alice might be a timesheet user, meaning that she enters her own time into the system. She might also be a timesheet reporter, meaning that she can produce reports based upon other people's timesheets. Finally, she could be a timesheet administrator, meaning that she can also edit any timesheet in the system.

Each of these three roles encompasses all the ability of the previous role, such that the timesheet app needs to know only the role with the greatest access level in order to function correctly. If Alice has `TIMESHEET_ADMIN` role, then she also has the abilities of the `TIMESHEET_REPORTER` and `TIMESHEET_USER` roles. You can order these roles by access level value, as shown in figure 1.

Figure 1 Example of the levels of access that Alice could have in the timesheet app

Overloading comparison operators

There is a natural order to these roles: one is greater or lesser than the other. To test whether two roles relate to each other, you can write code that compares each role's `accessLevel` values, which works adequately. It would

aid readability, however, if you could compare each of the role instances with each other directly, using the greater-than (>) and less-than (<) operators, as shown in listing 1.

Listing 1 Ways to compare roles

```
var adminRole = new Role("TIMESHEET_ADMIN",3);              #A
var reporterRole = new Role("TIMESHEET_REPORTER", 2);       #A
var userRole = new Role("TIMESHEET_USER", 1);               #A

if (adminRole.accessLevel > reporterRole.accessLevel) {     #B
  print("Admin role is greater than Reporter role");        #B
}                                                           #B
                                                            #B
if (userRole.accessLevel < adminRole.accessLevel) {         #B
  print("User role is less than Admin role");               #B
}                                                           #B

if (adminRole > reporterRole) {                             #C
  print("Admin role is greater than Reporter role");        #C
}                                                           #C

if (userRole < adminRole) {                                 #C
  print("User role is less than Admin role");               #C
}                                                           #C
```

#A Creating the three role instances
#B You could use the `.accessLevel` property to compare each role using < and > operators.
#C But you get better readability when you can compare the roles directly.

Fortunately, Dart allows this functionality with operator overloading, which means that you can take the standard operators and let your own classes provide meaning to them. In this instance, you want to provide meaning to the greater-than and less-than operator in the context of the `Role` class. Dart provides the ability to do this by providing our own version of operator implementation. The `operator` keyword lets Dart know that your class is defining an operator implementation, as shown in listing 2.

Listing 2 Providing your own implementation of < and > with the operator keyword

```
class Role {
  final String name;
  final int _accessLevel;                              #A

  const Role(this.name, this._accessLevel);

  bool operator >(Role other) {                        #B
    return this._accessLevel > other._accessLevel;     #B
  }                                                    #B

  bool operator <(Role other) {                        #C
    return this._accessLevel < other._accessLevel;     #C
  }                                                    #C

}
```

#A You can now hide the `_accessLevel` by making it private.
#B The `operator` keyword is paired with the operator you want to overload to provide a new function.

When you overload an operator, provide a method containing your implementation of the operator. The `operator`'s method usually takes a single parameter containing another instance of same class. Table 1 shows some common comparison operators that you can overload.

Table 1 Common comparison operators

Operator method	Description
`bool operator >(var other) {…}`	This instance is greater than the other.
`bool operator <(var other) {…}`	This instance is less than the other.

`bool operator >=(var other) {…}`	This instance is greater than or equal to the other.
`bool operator <=(var other) {…}`	This instance is less than or equal to the other.
`bool operator equals(var other) {…}` `bool operator ==(var other) {…}`	This instance is equal to the other. Note that there are two different versions of this method. At the time of this writing, the language spec defines the word `equals` as the operator, but the implementations are currently using a double equal sign == to represent the equals operator.

Surprising use for operator overloading

When overloading operators, the `other` value *should* be the same class, but there is no actual requirement that it *must be* the same class. This situation provides for some interesting, if slightly unorthodox syntax. For example, to add a role to a user, you *could* overload the `Users +` operator, allowing you to write the code shown in listing 3.

Listing 3 Overloading the addition operator to add Roles to a User

```
class User {
  List roles;

  User() {
    roles = new List();
  }

  operator +(Role newRole) {                #A
    this.roles.add(newRole);                #A
  }                                         #A
}

main() {
  User alice = new User();
  Role adminUser = new Role("TIMESHEET_ADMIN", 3);

  alice + adminUser;                        #B
  print(alice.roles.length);                #C
}
```

#A Override the + operator
#B Using the + operator
#C The role has been "added" to the user.

WARNING It is good practice to overload operators only when it would be unsurprising to the reader to do so. The previous example would be more readable if it had provided an `add(Role)` method instead. Developers do not like surprises.

Overloading indexer operators

When you deal with lists and maps, you use the indexer operators to write `[]=` and read `[]` a value in an instance of a class, such as:

```
myList[1] = "Some value";      #A
var myValue = myList[1];       #B
```

#A Using the [ ]= operator to write a value by index
#B Reading a value with the [ ] operator

The `[]` operator allows us to read a value by index; `[]=` allows us to write a value by index; and you can overload these in your classes to provide indexer access to underlying values. The `[]` operator method takes a single index parameter and returns a value, and the `[]=` takes both an index parameter and a value parameter that should be applied to that index item. Imagine a `User` class that could have only exactly two roles. You could use an indexer

to allow reading and writing to those two roles. Listing 4 uses indexers to access the underlying `_role1` and `_role2` properties.

Listing 4 Overloading the indexer operators

```
class User {
  Role _role1;
  Role _role2

  User() {
    roles = new List();
  }

  operator []=(int index, Role role) {                    #A
    if (index == 1) {
      _role1 = role;
    }
    else if (index == 2) {
      _role2 = role;
    }
    else throw new IndexOutOfRangeException();
  }

  Role operator [](int index) {                            #B
    if (index == 1) {
      return _role1;
    }
    else if (index == 2) {
      return _role2;
    }
    else throw new IndexOutOfRangeException();
  }
}

main() {
  User alice = new User();
  alice[1] = new Role("TIMESHEET_ADMIN", 3);              #C
  alice[2] = new Role("TIMESHEET_USER", 1);               #C
  var roleIndex1 = alice[1];                              #D
}
```

#A Overriding the [ ] = write indexer
#B Overriding the [ ] read indexer
#C Using the write indexer to set roles by index
#D Using the read indexer to read a role by index

A common reason to use indexers is to have a class to implement a `Map` interface so that properties on the class can be read as though they were part of a map, but they actually formed real properties. This method allows tools such as the JSON parser, which understands maps and lists, to convert your class into a JSON representation. When data is in a JSON format, it can be sent back and forth over the web. You can make your `Role` class implement a `Map` and convert it to JSON using the code shown in listing 5. Although the code has "snipped" some of the boilerplate methods required by the `Map` interface, you must provide all of them. Listing 5 also makes use of some of the other patterns you have seen in this chapter, such as returning list literals and returning typed and untyped generic collections.

Listing 5 Allowing your own class to implement Map allows it to be converted to JSON

```
class Role implements Map {                               #A

  String name;
  int _accessLevel;

  Role(this.name, this._accessLevel) {}

  //Map methods
  bool containsKey(String key) {                          #B
    return key == "name" || key == "accessLevel";         #B
  }                                                       #B
```

```
  operator[](String key) {
    if (key == "name") return this.name;                    #C
    if (key == "accessLevel") return this._accessLevel;     #C
    return null;
  }

  void operator[]=(String key, var value) {
    if (key == "name") this.name = key;                     #D
    if (key == "accessLevel") this._accessLevel = value;    #D
  }

  Collection<String> getKeys() {
    return ["name","accessLevel"];                          #E
  }

  Collection getValues() {
    return [this.name, this._accessLevel];                  #F
  }

  //…snip other map methods…

}
```

#A Implementing the `Map` interface
#B Returning true if the key is `name` or `accessLevel`
#C Overloading the `[]` operator to allow reading properties
#D Overloading the `[]=` operator to allow writing properties
#E Returning a typed collection of `String` key names created as a literal list
#F Returning an untyped collection of values, created as a literal list

Now that you have implemented map in our Role class, we can use the `JSON.stringify()` method (defined in the `dart:json` library) to convert an instance of a role into a string, as in the following snippet:

```
Role adminRole = new Role("TIMESHEET_ADMIN",3);
var roleString = JSON.stringify(adminRole);
```

You can use this serialized string to send the `Role` data over the web.

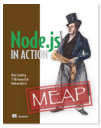
REMEMBER

- Use the `operator` keyword in conjunction with the operator symbol to provide a new method in your class to overload the operator.
- Ensure that you overload operators only where it would aid readability of the code to do so.
- Indexer operators can be overloaded to allow map-like access to properties of your class.
- The `dart:json` library can convert classes that implement the `Map` interface into JSON strings.

Summary

We looked at operator overloading, which allows you to aid readability when using your classes by providing your own versions of common operator symbols, such as > (greater than) and < (less than). The culmination of operator overloading was to use the indexer operators `[]` and `[]=` to provide your own implementation of the `Map` interface, which allows your class to be converted to JSON by the built-in JSON library.

Here are some other Manning titles you might be interested in:



Node.js in Action
Mike Cantelon, TJ Holowaychuk and Nathan Rajlich

Third-Party JavaScript
Ben Vinegar and Anton Kovalyov

HTML5 in Action
Robert Crowther, Joe Lennon, and Ash Blue

Last updated: November 7, 2012