Objects vs. Data Structures

Originally published by Matt Carroll on January 16th 2016 ★ 26,601 reads

Matt Carroll

Start Writing Log in

.tech Never get a boring domain. Get .Tech!

Objects vs. Data Structures by @mattcarroll

In the world of Java development, there are objects and there are Objects. The former is more important.

Confused yet?

Object-oriented programming (OOP) introduced the world to the concept of objects (little “o”). OOP objects are encapsulation boundaries around state that provide some kind of public behavior. The concept of objects is not limited to any specific language. In fact, we see objects represented in class-based languages like Java/C#/Ruby/Python, as well as prototype-based languages like JavaScript.

Java introduced a Java Class called “Object” (with a big “O”). Java Objects are instances of the Object class (including all subclasses). These Objects are language constructs, not conceptual constructs.

Thus, the question is begged: do Java Objects qualify as OOP objects? Well, it depends on circumstance. This article looks at the specific circumstances around the use of data structure

Objects vs. OOP objects.

Data Structures

Consider a data structure representing a person that includes a first name, last name, and phone number. How might that data structure look in various procedural languages?

Person data structure in C:

```
struct Person {
char firstName[20];
char lastName[20];
char phoneNumber[10];
};
```

Person data structure in Pascal:

```
type
Person = record
firstName : string;
lastName : string;
phoneNumber : string;
end
```

In Java, that same data structure might look like this:

```
public class Person {
public String firstName;
public String lastName;
public String phoneNumber;
}
```

The Java data structure is “technically” different than the C and Pascal versions, because the Java data structure is a Class instead of a struct or a record. But is the Java Person functionally different than the C struct or the Pascal record? No. Its functionally the exact same thing. All 3 data structures provide 3 string fields that can be read or written.

The important point is that the Java Person Object isn’t an “object” at all, its a data structure. The Person Object exists to organize some data into a single entity that can be passed around and managed as a whole—just like a C struct and a Pascal record.

But what if the Java Person Object looked like this:

```
public class Person {
private String mFirstName;
private String mLastName;
private String mPhoneNumber;
```

```
public String getFirstName() {
return mFirstName;
}
```

```
public void setFirstName(String firstName) {
mFirstName = firstName;
}
```

```
public String getLastName() {
return mLastName;
}
```

```
public void setLastName(String lastName) {
mLastName = lastName;
}
```

```
public String getPhoneNumber() {
return mPhoneNumber;
}
```

```
public void setPhoneNumber(String phoneNumber) {
mPhoneNumber = phoneNumber;
}
}
```

Now is the Java Object a real object? Now it has private data and public methods. It must be an OOP object, right?

Even with public getters and setters, Person is still a data structure. Has its purpose changed? Has its behavior changed? No. The Person Object would still be used in the same manner and fashion as its public-property predecessor. The getter/setter version of Person is still 100% a data structure.

Ok, what if we start adding some behavior to Person, like this:

```
public class Person {
```

```
//... same getters/setters as before, except one:
```

```
public void setPhoneNumber(String phoneNumber) throws FormatException {
validatePhoneNumber(phoneNumber);
mPhoneNumber = phoneNumber;
}
```

```
private void validatePhoneNumber(String phoneNumber) throws FormatException {
// Do validation here to ensure we have a legit phone number.
// Throw an exception if its invalid.
}
```

```
}
```

Now do we have a real OOP object? This latest version of Person now includes validation behavior, and it even uses a private method to implement the validation.

Even this latest incarnation of Person isn't completely an OOP object. It's more of a Franken-object. The phone number validation behavior really is behavior—its a service—which is what OOP objects are supposed to be. But notice that we still have all of these methods that exist to read and write state. There really isn't any state hiding here, and there isn't much behavior at all. Person still remains primarily a data structure that exposes its entire state, does almost nothing, and will be used as a concrete type throughout the code base.

No matter how much lipstick you put on this pig, Person is a data structure, not an object.

OOP Objects

What does an OOP object look like? It looks like a service. An OOP object is a construct that does things—it behaves and acts.

A Person data structure has a first name, last name, and phone number. A Person object walks, runs, jumps, and speaks. A Person object does things.

Here are some OOP object examples:

```
public interface PhoneNumberValidator {
```

```
boolean validate(String phoneNumber);
```

```
}
```

PhoneNumberValidator validates that a given String represents a correctly formatted phone number. There is no indication of internal state within the validator. Maybe the validator has state, or maybe it doesn't, but we do know it offers the service of phone number validation.

```
public interface PersonDataStore {
```

```
Person getPeople(PeopleQuery query);
```

```
void addPerson(Person person);
```

```
void removePerson(Person person);
```

```
}
```

PersonDataStore provides a mechanism for storing and querying Person data structures. In

PersonDataStore provides a mechanism for storing and querying Person data structures. In this example, both Person and PeopleQuery are data structures—they simply organize information. However, the PersonDataStore is an object that provides services. Namely,

PersonDataStore can take a PeopleQuery data structure and find all the Person data structures that match the criteria in the query.

Does PersonDataStore persist in-memory? Does it persist to disk? Does it index data? As a client of the PersonDataStore, we don’t know any of these things and we don’t care. We only care about what PersonDataStore does, because PersonDataStore is an OOP object.

Confusing Objects and Data Structures

Consider the Person data structure again. In the final example we added phone number validation behavior. You can imagine that we might want similar validations for first name and last name. Moreover, when we add more fields to this data structure, those fields might need to be validated as well.

The problem with validation **behavior** on our Person **data structure** is that we have created a candidate that will routinely violate the Open/Closed Principle, the Single Responsibility Principle, and the Interface Segregation Principle:

Open/Closed Principle:

Each new field requires opening the Person class and adding code for validation.

Single Responsibility Principle:

The Person class now has the responsibility of structuring data, validating first names, validating last names, and validating phone numbers. That’s a lot of responsibilities that can vary independently.

Interface Segregation Principle:

Imagine that you are concerned only with collecting a phone number. When you work with an instance of Person, you depend not only on the validatePhoneNumber() method, but also the

validateFirstName() and validateLastName() methods. Thus, you are dependent upon methods that you do not require and therefore you have violated the Interface Segregation Principle.

Data vs. Behavior

The general lesson that we can learn from this Person example is that data and behavior do not vary together. Data is grouped by I/O considerations like web API input formats and database schemas. However, behavior is grouped by use-cases that represent what a client wants to do with the application.

When we take data structures and we begin adding behavior then we invite all of the OOP violations seen in the Person example.

Instead, define behaviors based on your use-cases, define data structures based on your I/O requirements, and then define objects to connect the two. For example, an Android Activity might define a method like the following:

```
// Called when the user finishes entering a phone number.
private void onPhoneNumberSet() {
String phoneNumber = mPhoneNumberTextView.getText().toString();
boolean isValid = mPhoneNumberValidator.validate(phoneNumber);
if (isValid) {
mPerson.setPhoneNumber(phoneNumber);
} else {
// Notify user of problem.
}
```

```
    }
```

In this example, Person remains its own data structure that doesn't know about validation behavior. Then, there is a PhoneNumberValidator object that knows how to validate a phone number string, but doesn't know anything about the Person data structure. Finally, the containing Activity orchestrates the phone number validation followed by setting the phone number data on the Person data structure.

When its time to submit the Person data to a datasource (web or local DB), you can imagine something like this:

```
    private void doSubmit() {
    mPersonDataStore.addPerson(mPerson);
    }
```

We don't know whether the PersonDataStore is talking to a local DB, or to a web server, or even to local memory. All we know is that our use-case responsibility is to collect Person data and store it. We used OOP objects to validate the input, and now we use an OOP object to store the data. During this process, the data was collected in the Person data structure.

Why All The Fuss?

Why should anyone care about this nuanced distinction between data structures and objects?

Data structures are state.

Let me repeat that for effect.

Data structures are state.

Therefore, passing around data structures means sharing state, and shared state is the root of all evil. The reason OOP objects were invented was to provide a paradigm where shared state could be minimized and controlled (that's why we should Package Wisely).

Think of data structures as an interchange format within your code, between your OOP objects. Consider the following pseudo code:

```
objectA.doThing1();
objectA.doThing2();
```

```
myDataStructure = objectA.extractState();
```

```
objectB.setState(myDataStructure);
objectB.doThing3();
objectB.doThing4();
```

We use objectA to do some work. Then we extract the state of objectA by obtaining a data structure. We initialize objectB by sending in the data structure we obtain from A, and now objectB does some work. In this example there is an implied immutability to myDataStructure such that changing myDataStructure would not have any hidden, internal impact on objectA or objectB.

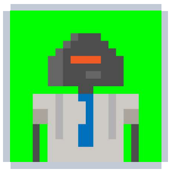
The data structure simply provides a mechanism for moving around organized state. It does not offer any behavior of its own. This is how we should aim to utilize data structures in our code.

Its also important to realize that state extraction should be a relatively low-frequency operation. Most of the time you should aim for objects to accept and return other objects, not data structures.

You should aim to employ data structures only when crossing between different domain in your application. For example, when you receive input in your I/O domain, then you might use a data structure to inject that data into your business domain. Likewise, when you need to present something visually to the user, you might use a data structure to extract the information from your business domain and send it into your visual domain. Data structures are an interchange format between application domain.

As you develop your applications, keep in mind that Java Objects are not necessarily OOP objects, data structures are never the same thing as OOP objects, and you should be sure to recognize and separate these different constructs.

Share this story



@mattcarroll

Matt Carroll

Read my stories