Software Design and Algorithms



Building blocks of OOP, part 1

EPAM Systems Inc.

Learn & Development

Software Design and Algorithms

[www.epam.com](http://www.epam.com/)

Contents

[1. Inheritance 3](#_Toc65752654)

[1.1. Intro 3](#_Toc65752655)

[1.2. Recognizing where to use inheritance 4](#_Toc65752656)

[1.3. Creating an Abstract Superclass with shared Abstract Behavior 7](#_Toc65752657)

[1.4. Template Method Pattern: Default Implementation 8](#_Toc65752658)

[1.5. Using Hook Messages: Decoupling Subclasses 9](#_Toc65752659)

[1.6. Summary 10](#_Toc65752660)

[2. Interface 11](#_Toc65752661)

[2.1. When should a class implement an Interface? 13](#_Toc65752662)

[2.2. When should a class depend on an interface? 13](#_Toc65752663)

[2.3. Summary: Creating a Message-Based Application 18](#_Toc65752664)

[3. The Law of Demeter 19](#_Toc65752665)

[3.1. Avoiding Violations 21](#_Toc65752666)

[4. Interface vs Abstract class 23](#_Toc65752667)

# Inheritance

## Intro

JavaScript is a bit confusing for developers experienced in class-based languages (like Java or C++), as it is dynamic and does not provide a **class** implementation per se (the **class** keyword is introduced in ES2015, but is syntactical sugar, JavaScript remains prototype-based).

When it comes to inheritance, JavaScript only has one construct: objects. Each object has a private property which holds a link to another object called its ***prototype***. That prototype object has a prototype of its own, and so on until an object is reached with **null** as its prototype. By definition, **null** has no prototype, and acts as the final link in this **prototype** chain**.**

Nearly all objects in JavaScript are instances of **Object** which sits on the top of a prototype chain.

While this confusion is often considered to be one of JavaScript's weaknesses, the prototypal inheritance model itself is, in fact, more powerful than the classic model. It is, for example, fairly trivial to build a classic model on top of a prototypal model.

Inheritance mechanisms, which play a key role in the object approach in terms of extensibility and reuse, model the relationship (IS-A relationship) and exploit the relationship between the base class and its descendant.

Inheritance is the mechanism of basing an object or class upon another object (prototypical inheritance) or class (class-based inheritance), retaining similar implementation.

In most class-based object-oriented languages, an object created through inheritance (a "child object") acquires all the properties and behaviors of the parent object

Experienced developers can tell you that overuse of inheritance leads to code that is difficult to understand and maintain. This is primarily because the IS A relationship is much stronger than the relationship that appears during composition. Therefore, when making changes, you need to be very careful and see if any methods have been overridden, what is the contract of the parent class, at the level of coupling.

Inheritance is essentially an automatic message delegation mechanism. Inheritance creates a relationship in which if one object cannot respond to a received message, it passes that message to another. And this transfer happens automatically.

## Recognizing where to use inheritance

Let's look at an example where inheritance can be applied. Let's imagine that we want to rent a bike.

class Bicycle {  
 constructor(options) {  
 this.size = options.size;  
 this.tapeColor = options.tapeColor;  
 }  
   
 spares() {  
 return {  
 chain: '11-speed',  
 tireSize: '28',  
 tapeColor: this.tapeColor,  
 }  
 }  
}  
  
const bike = new Bicycle({size: 'M', tapeColor: 'red'});  
bike.size; // => 'M'  
bike.spares() // => {chain: '11-speed', tireSize: '28', tapeColor: 'red'}

And we need to know what parts need to be replaced in case of a breakdown, so we added **spares** method.

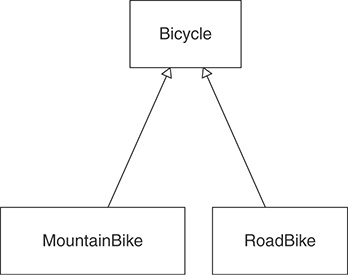
But what happens if we add another bike style?   
For example, a mountain bike. Now our method looks a little strange.

class Bicycle {  
 constructor(options) {  
 // previous options  
 this.style = options.style;  
 this.frontShock = options.frontShock;  
 }  
  
 spares() {  
 if (this.style == 'road') {  
 return {  
 chain: '11-speed',  
 tireSize: '28',  
 tapeColor: this.tapeColor,  
 }  
 }  
 return {  
 chain: '11-speed',  
 tireSize: '29',  
 frontShock: this.frontShock,  
 }  
 }  
}  
  
const crossCountryBike = new Bicycle({style: 'XC', size: 'M', frontShock: 'mountain'})  
const roadBike = new Bicycle({style: 'road', size: 'M', tapeColor: 'red'})

We added 2 new aruments for MountainBike. The new **style** property defines what the spares method (details) will return using the if statement inside the method.

Already this code is hard to maintain, and if we need to add another type of bicycles, our code will turn into a big dump.

It's time for an Abstract class.



Subclasses are specializations of their superclasses. MountainBike should have everything a Bicycle plus something else.

The picture above shows a class diagram where Bicycle is a superclass for MountainBike and RoadBike. The bike will contain general behavior, and MountainBike and RoadBike will add specializations. The public interface of the **Bike** should include a spares method and a size property, and the interfaces of its subclasses will add the necessary parts.

The bike is now an abstract class.

## Creating an Abstract Superclass with shared Abstract Behavior

abstract class Bicycle {  
 // keep only common parts  
}  
  
class RoadBike extends Bicycle {  
 constructor(options) {  
 super(options)  
 this.tapeColor = options.tapeColor;  
 }  
  
 spares() {  
 return { ...super.spares(), tapeColor: this.tapaColor, }  
 }  
}  
  
class MountainBike extends Bicycle {  
 constructor(options) {  
 super(options)  
 this.frontShock = options.frontShock;  
 }  
  
 spares() {  
 return { ...super.spares(), frontShock: this.frontShock, }  
 }  
}

Abstract classes exist in order to inherit from them. They provide a common repository that stores the behavior common to all subclasses, and each of them is a specialization of an abstract class.

It almost never makes sense to create an abstract superclass with a single subclass.

## Template Method Pattern: Default Implementation

This gives subclasses the ability to inject specialization by overriding the default values ​​set in the parent class. This technique of describing the basic structure / algorithm in a superclass and redefining parts of this structure / algorithm to those that are already specific for a particular class is called the template method.

Bicycle now provides a structure, a general algorithm, for its subclasses. In those places where the base class provides the ability to influence this algorithm to the derived classes, it sends messages **default\_chain** and **default\_tire\_size**.

abstract class Bicycle {  
 protected readonly defaultChain = '11-speed';  
 constructor(opts) {  
 // ...  
 this.chain = opts.chain || this.defaultChain;  
 this.tireSize = opts.tireSize || this.defaultTireSize;  
 }  
}  
  
class RoadBike extends Bicycle {  
 protected readonly defaultTireSize = '28';  
}  
  
class MountainBike extends Bicycle {  
 protected readonly defaultTireSize = '29';  
}

Something similar can be found in React js. When creating a component, it has methods **componentDidMount()**, **shouldComponentUpdate()** you can change their implementation yourself.

Okay, we split the MountainBike and the RoadBike, but now there are new problems:

* Mountain bike and road bike classes depend on their abstract class;
* Abstract class depends on children;
* If you forget to call super methods – the result might not contain all data required;
* Users of road and mountain bike depend on the abstract class, even if they don’t know anything about it.

## Using Hook Messages: Decoupling Subclasses

This strategy removes the knowledge of the algorithm from the subclass and returns control to the superclass. Which was done by adding the **postInitialize** method.

RoadBike and MountainBike no longer control the initialization process, but instead bring specialization to a more abstract algorithm. This algorithm is defined in the abstract superclass Bicycle, which in turn is responsible for sending **postInitialize**.

This same technique can be used to remove the dispatch of super in the spares method.

abstract class Bicycle {  
 constructor(opts) {  
 this.size = opts.size;  
 this.chain = opts.chain;  
 this.tireSize = opts.tireSize;  
 this.postInitialize(opts);  
 }  
   
 protected postInitialize() {};  
   
 spares() {  
 return {  
 tireSize: this.tireSize,  
 chain: this.chain,  
 ...this.localSpares()}  
 }  
 }

class RoadBike extends Bicycle {  
 protected postInitialize(opts) {  
 this.tapeColor = opts.tapeColor;  
 };  
  
 protected localSpares() {  
 return { tapeColor: this.tapeColor };  
 }  
}

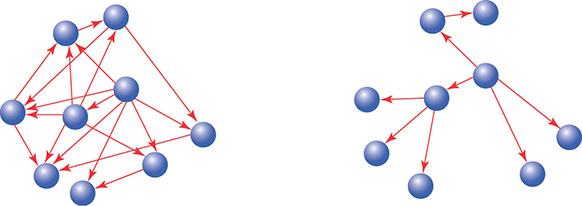
class MountainBike extends Bicycle {  
 protected postInitialize(opts) {  
 this.frontShock = opts.frontShock;  
 };  
  
 protected localSpares() {  
 return { frontShock: this.frontShock };  
 }  
}

## Summary

* Inheritance solves the problem of related types that share a great deal of common behavior but differ across some dimension;
* The best way to create an abstract superclass is by pushing code up from concrete subclasses;
* Abstract superclasses use the template method pattern to invite inheritors to supply specializations, and they use hook methods to allow these inheritors to contribute these specializations without being forced to send super;
* Well-designed inheritance hierarchies are easy to extend with new subclasses, even for programmers who know very little about the application;

# Interface

As mentioned earlier, OOP is directly related to messages sent between objects. Moreover, it is important not only what objects know, what behavior they have inside, on whom they depend (to whom messages are sent), but also how they communicate with each other. Objects communicate through interfaces.



Public Interfaces

* Reveal its primary responsibility.
* Are expected to be invoked by others.
* Will not change on a whim.
* Are safe for others to depend on.
* Are thoroughly documented in the tests.

Private Interfaces

* Handle implementation details.
* Are not expected to be sent by other objects.
* Can change for any reason whatsoever.
* Are unsafe for others to depend on.
* May not even be referenced in the tests

What do interfaces give us? The presence of interfaces allows you to think about the task in a more abstract way, ignoring minor details.

There are different levels of abstraction or ignorance of details.

Every concrete class is abstract and has hidden behavior. And the class will be implemented, some interface will depend not on the requirement of abstraction, but on the model.

## When should a class implement an Interface?

1. Class implements Strategy pattern or it is part of family of objects: IRepository, IFormatter, IPrecondition;
2. Class implements role interface (as a result of ISP): ICloneable, IComparable, etc.;
3. Class implements interface required for connection with other classes. Class is an Adapter, the need for an interface is determined by DIP;
4. Class implements interface, because external environment depends on it. Provides testability to users of this class. This is not a single reason to create an interface.

## When should a class depend on an interface?

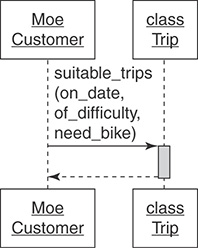
When should a class depend on an interface? For example, getting it through the constructor. The arguments of the class indicate that he cannot take some decision on his own and he needs help from the called class. It's easiest when a class depends on primitives, a little more complicated when it depends on specific classes, and most difficult when it depends on an interface.

1. Class deals with family of types: the “family of types” exists already and defined by requirements of existing model;
2. As a result of DIP: class wants to communicate with object of another level, it defines the interface by itself and requires its implementation;
3. For testing purposes: useful in case if implementation of the abstraction relies on external environment.

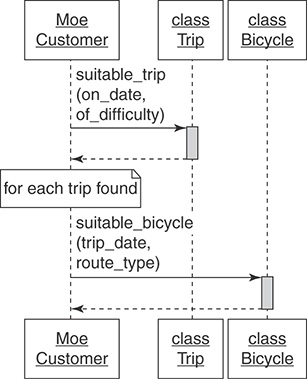
Don’t create interfaces “Just in case”!

Finding the public Interface

Simple sequence diagram:



Moe talks to trip and bicycle:



Each route is rated according to its difficulty. Mountain bike rides have an additional difficulty rating. Clients have a certain level of fitness and technical skill level for mountain biking, based on this we can determine if the requested ride is suitable for them.

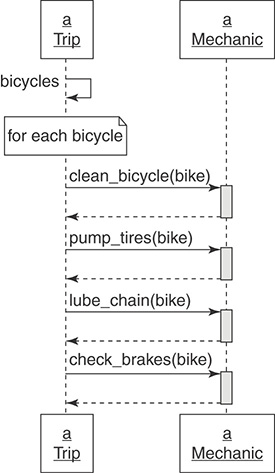
Customers can rent bicycles, or they can bring their own.

Should the Trip class be responsible for figuring out if there is a suitable bike for each given trip? Or more generally “Should the recipient of this message (**Trip**) be responsible for responding to this message? "

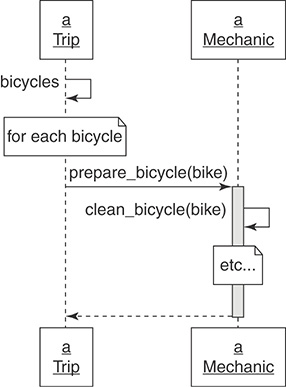
The Trip class will respond to the **suitable\_trips** message, and the Bicycle class will respond to the **suitable\_bicycle** message. The castomer can get the result if it communicates with 2 objects

Asking for “What” Instead of Telling “How”

Trip tells a Mechanic how to prepare each Bicycle:



Trip asks a Mechanic to prepare each Bicycle:



1st diagram: the **Trip** is almost ready and it needs to make sure that all the bicycles that will be participating in it are ready for it. Thus, the **Trip** must know exactly how to prepare the bike and what messages and in what sequence to send to the **Mechanic**. **Trip** each every bike, it washes it, fixes it and gives it away for a ride.

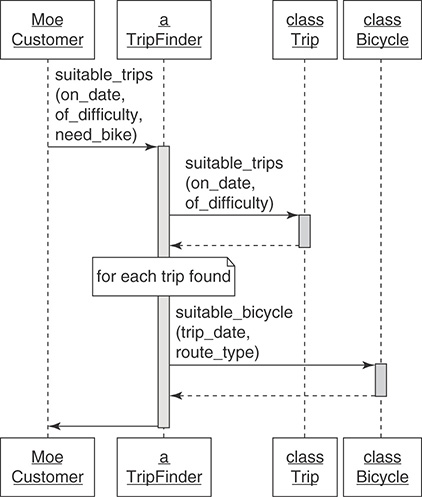
The downside is that the **Trip** needs to know this whole big **Mechanic** interface, and secondly, if the mechanic starts implementing new behavior, we will need to update the **Trip** class.

In the 2nd diagram, **Trip** asks the mechanic to prepare each bike, leaving the implementation details to the **Mechanic**.

This refactoring greatly improves support and extensibility. But this is not an ideal solution yet, because **Trip** knows about bicycles as well as about mechanics, perhaps in the future this interface can also be reduced.

**Using Messages to Discover Objects**

Moe asks the TripFinder for a suitable trip



This diagram is already based on refactoring. We have created the **TripFinder** class, which aggregates the **Trip** class and the **Bicycle** class in itself, it refers to the **Trip** class and says - tell us what are the available trips and for each available trip checks the required bike, and the user returns the list of available trips.

At this stage, **TripFinder** implements the logic of working with our internal systems.

## Summary: Creating a Message-Based Application

1. *Create Explicit Interfaces* - every time you create a class, declare its interfaces. Methods in the *public* interface should:
   * Be explicitly identified as such.
   * Be more about what than *how*.
   * Have names that, insofar as you can anticipate, will not change.
2. *Honor the Public Interfaces of Others* - do your best to interact with other classes using only their public interfaces.
3. *Exercise Caution When Depending on Private Interfaces* - despite your best efforts, you may find that you *must* depend on a private interface, this is a dangerous dependency that should be isolated.
4. *Minimize Context* - construct public interfaces with an eye toward minimizing the context they require from others. Keep the what versus *how* distinction in mind; create public methods that allow senders to get what they want without knowing how your class implements its behavior.

# The Law of Demeter

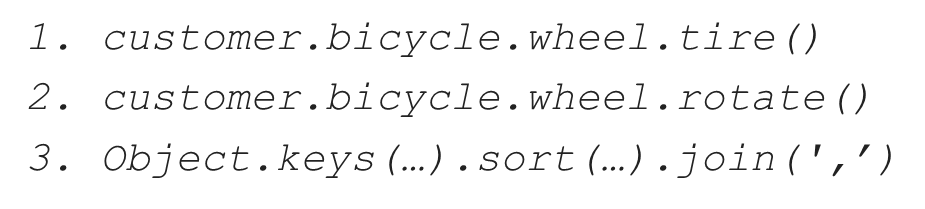
The purpose of the law of Demeter is lower cohesion. The low cohesiont shows that the design is of good quality. In short, the law of Demeter sounds like this:

Only talk to your immediate neighbors,

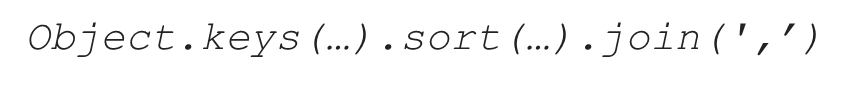
or in other words: use only one dot.

Some violations of the Law of Demeter are harmless, although others are a sign that the public interface has not been segregated correctly or has not been segregated at all.

Imagine that Trip has a function that checks whether the tire is pumped up or not, ask if the wheel is spinning, and so on. Each line contains a large number of points for accessing internal properties. This may be an indication that the law of demeter has been violated.



* If wheel changes tire or rotate, depart may have to change. Trip has nothing to do with wheel, yet changes to wheel might force changes in Trip. This unnecessarily raises the cost of change; the code is not reasonable.
* Changing tire or rotate may break something in depart. Because Trip is distant and apparently unrelated, the failure will be completely unexpected. This code is not transparent.
* Trip cannot be reused unless it has access to a customer with a bicycle that has a wheel and a tire. It requires a lot of context and is not easily usable.
* This pattern of messages will be replicated by others, producing more code with similar problems. This style of code, unfortunately, breeds itself. It is not exemplary.



The third line of messages *Object*.keys(…).sort(…).join(‘,’) is perfectly reasonable and does not violate the Law of Demeter.

*Object*.keys() returns an Array.

*Object*.keys().sort() also returns an Array.

*Object*.keys().sort().join()  returns a String.

It is also quite normal if you are working with a data structure that has a lot of innateness, but you also need to be careful here.

## Avoiding Violations

One example of how you can avoid unnecessary dependency.

We have a User this have Account and this account has a plan and User has **discountedPlanPrice** which violate the Law of Demeter.

abstract class User {  
 private account: Account;  
  
 discountedPlanPrice(coupon: Coupon) {  
 return coupon.discount(  
 this.account.getPlan().getPrice(),  
 );  
 }  
}  
  
class Account {  
 private plan: Plan;  
  
 get plan() {  
 return this.plan;  
 }  
}

One of the common ways to remove such chains is by using delegation. A wrapper method encapsulates or hides knowledge that would otherwise be implemented in the message chain.

abstract class User {  
 private account: Account;  
  
 discountedPlanPrice(coupon: Coupon) {  
 return this.account.discountedPlanPrice(coupon)  
 }  
}  
  
class Account {  
 private plan: Plan;  
  
 discountedPlanPrice(coupon) {  
 return coupon.discount(this.plan.getPrice());  
 }  
}

To avoid such problems, you need to think over the application architecture in advance

Delegation is an effective technique to avoid Law of Demeter violations, but only for behavior, not for attributes.

# Interface vs Abstract class

|  |  |  |
| --- | --- | --- |
| **Parameters** | **Interface** | **Abstract class** |
| Multiple inheritances | Implement several interfaces | Only one abstract class |
| Structure | Abstract methods | Abstract & concrete methods |
| When to use | Future enhancements | To avoid independence |
| Adding new methods | Could be hard | Easy to do |
| Access modifiers | Only public | Public, protected, private |
| Usage | Defines the peripheral abilities of a class | Defines the identity of a class |

An interface is more flexible from a client's point of view: any class can implement any interface. But the interface is "**stiffer**" from the point of view of its developer: it is more difficult to change it (the work of all clients will be broken), restrictions cannot be imposed on the client's constructor, and the code cannot be reused.

Important Reasons For Using Interfaces:

* Interfaces are used to achieve abstraction.
* Designed to support dynamic method resolution at run time
* It helps you to achieve loose coupling.
* Allows you to separate the definition of a method from the inheritance hierarchy

An abstract class is "**stiffer**" from the clients' point of view: the client will be forced to abandon the current base class. But an abstract class is "more flexible" from the point of view of its developer: it allows you to reuse code, restrict the constructor of descendants, allow you to make changes (easily add a virtual method without breaking existing clients), and more clearly define a "contract" with descendants using Template Methods.

Important Reasons For Using Abstract Class:

* Abstract classes offer default functionality for the subclasses.
* Provides a template for future specific classes
* Helps you to define a common interface for its subclasses
* Abstract class allows code reusability.