

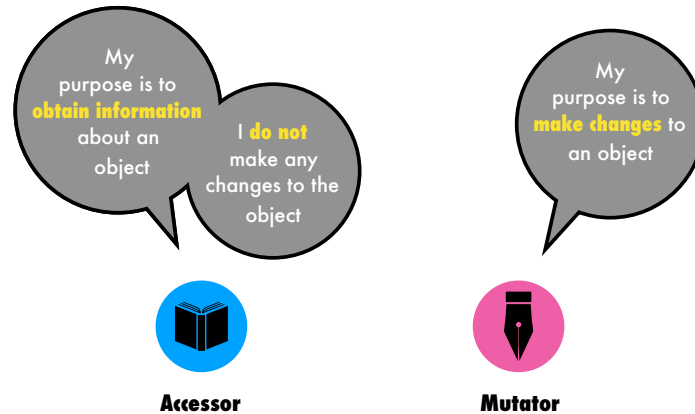


Accessors vs Mutators

presentation by Dr. K

Hello, this is Dr. K and welcome to this Java short cut. [space] In this video I'm going to talk about the difference between accessor methods and mutator methods.

Accessor vs Mutator methods



You will sometimes hear Java programmers refer to a method in a Java class as either an accessor method [space] or a mutator method [space]. The purpose of an accessor method is to [space] obtain information about an object. An important feature of accessor methods is that they do **not** make any changes to an object. In contrast, making changes to an object [space] **is** the whole purpose of a mutator method. Another way to look at this is that accessor methods are read-only -- they only read information from an object, while mutator methods will always **write** to an object.

The various meanings of "accessor method"

1

Methods used to obtain information about an object are known as **accessor methods**. One accessor method that you can use with strings is the `length()` method.

– Strings, The Java Tutorial

2

Constructs a `PropertyDescriptor` for a property that follows the standard Java convention by having `getFoo` and `setFoo` **accessor methods**.

– Property Descriptor, Java API

3

Often a setter is accompanied by a getter (also known as an **accessor**), which returns the value of the private member variable.

– Mutator Method, Wikipedia

You have to be a little careful when you hear people talk about *accessor* methods. In the Java Tutorial on String, they give the same definition that we just gave: [space] accessor methods are used to obtain information about an object. However, the Java API for a `PropertyDescriptor` refers to both [space] `getFoo` and `setFoo` as accessor methods. In other words, accessors are simply getters and setters. [pause] I hope you can see that this usage is *not* compatible with the first definition, because a method that *sets a property* of an object is, by its very nature, modifying that object. Why the different definitions? Because this second definition actually used to be quite common. Getters gave you read access to object properties and setters gave you write access. So they were both called accessor. Nowadays, this definition is not used as much, but you *will* still see it used by Java developers, especially if they work a lot with Java Beans. Finally, [space] the Wikipedia article on "mutator methods" talks about accessors and mutators as if they are *equivalent* to getters and setters. In my opinion, this is just a bad equivalency and can only lead to more confusion. A getter method should have the form "getProperty" and a setter method should have the form "setProperty". That's the easiest way to think about getters and setters.

The various meanings of "accessor method"

①

Methods used to obtain information about an object are known as **accessor methods**. One accessor method that you can use with strings is the `length()` method.

We use
this one

– Strings, The Java Tutorial

②

~~Constructs a `PropertyDescriptor` for a property that follows the standard Java convention by having `getFoo` and `setFoo` **accessor methods**.~~

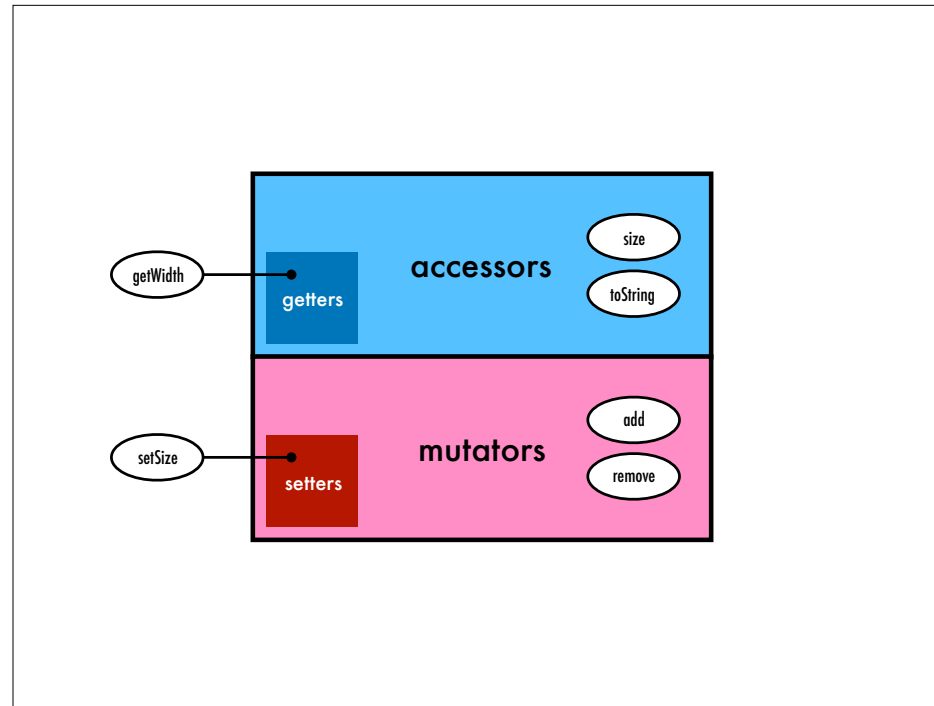
– Property Descriptor, Java API

③

~~Often a **setter** is accompanied by a **getter** (also known as an **accessor**), which returns the value of the private member variable.~~

– Mutator Method, Wikipedia

You should absolutely be aware of these different usages of the term "accessor methods". However, when "we" use the term "accessor", we will be using it in the sense of this "first" definition.



So our categorization looks like this. Accessors and mutators are mutually exclusive -- a method cannot be both. Getters always start with the word "get", followed by the property name, although if the property is a boolean value, they may start with the words "is" or "has". Getters are always accessors. Setters always start with the word "set" followed by the property name. Setter are always mutators. [pause] Examples of accessors are [space] the "size" method for a collection and the "toString" method for any object. These methods do **not** modify the object, rather they return information about the object. Examples of mutators are [space] the "add" and "remove" methods of collections. These methods **do** modify the collection. Finally [space] a good example of a getter is the "getWidth" method of a Java rectangle component, and an example of a setter is the "setWidth" method of a Java rectangle. Since setSize takes two arguments, you may get some disagreement on whether it is a true setter method, but we're going to call it a setter anyway.

Accessors or Mutators

Type	Method and description
int	<code>depth()</code> Returns the number of elements in this stack.
boolean	<code>isEmpty()</code> Tests if this stack is empty.
E	<code>peek()</code> Returns a handle to the element at the top of this stack without removing it.
E	<code>pop()</code> Removes and returns the element at the top of this stack.
void	<code>push(E element)</code> Adds the specified element to the top of this stack.

Let's try a little exercise. Do you think you can spot the accessor methods and the mutator methods in this stack component? Please pause the video and try to figure out which of these methods are mutators and which are accessors. I'll give you 10 seconds to do that.

Pause and Think

Which of the following methods are **accessors** and which are **mutators**? Why?

Type	Method and description
int	<code>depth()</code> Returns the number of elements in this stack.
boolean	<code>isEmpty()</code> Tests if this stack is empty.
E	<code>peek()</code> Returns a handle to the element at the top of this stack without removing it.
E	<code>pop()</code> Removes and returns the element at the top of this stack.
void	<code>push(E element)</code> Adds the specified element to the top of this stack.

[10 seconds]

Pause and Think

Which of the following methods are **accessors** and which are **mutators**? Why?

Type	Method and description	
int	<code>depth()</code> Returns the number of elements in this stack.	Accessor
boolean	<code>isEmpty()</code> Tests if this stack is empty.	Accessor
E	<code>peek()</code> Returns a handle to the element at the top of this stack without	Accessor
E	<code>pop()</code> Removes and returns the element at the top of this stack.	Mutator
void	<code>push(E element)</code> Adds the specified element to the top of this stack.	Mutator

Okay, I hope you were able to see that the first three methods -- `depth`, `isEmpty`, and `peek` -- [space] are all accessors. They don't change the state of the stack at all. They just return information. The last two methods -- `push` and `pop` -- [space] are mutators. They *do* change the state of the stack. That is their purpose. And even though `pop` returns an element (it's not a void method like `push`) it also mutates the stack, so it must be a mutator.

Accessors or Mutators

Type	Method and description
char	<code>charAt(int index)</code> Returns the character value at the specified index.
<code>String</code>	<code>concat(String str)</code> Concatenates the specified string to the end of this string.
boolean	<code>contains(String str)</code> Returns true if and only if this string contains the specified string.
char[]	<code>toCharArray()</code> Converts this string to a new character array.
<code>String</code>	<code>toUpperCase()</code> Returns a copy of this string in which all characters are upper case.

Let's look at another example. Do you think you can spot the accessor methods and the mutator methods in this Java string class? Once again, I'll give you 10 seconds to think about this or go ahead and pause the video if you need to.

Pause and Think

Which of the following methods are **accessors** and which are **mutators**? Why?

Type	Method and description
char	<code>charAt(int index)</code> Returns the character value at the specified index.
<code>String</code>	<code>concat(String str)</code> Concatenates the specified string to the end of this string.
boolean	<code>contains(String str)</code> Returns true if and only if this string contains the specified string.
char[]	<code>toCharArray()</code> Converts this string to a new character array.
<code>String</code>	<code>toUpperCase()</code> Returns a copy of this string in which all characters are upper case.

[10 seconds]

Pause and Think

Which of the following methods are **accessors** and which are **mutators**? Why?

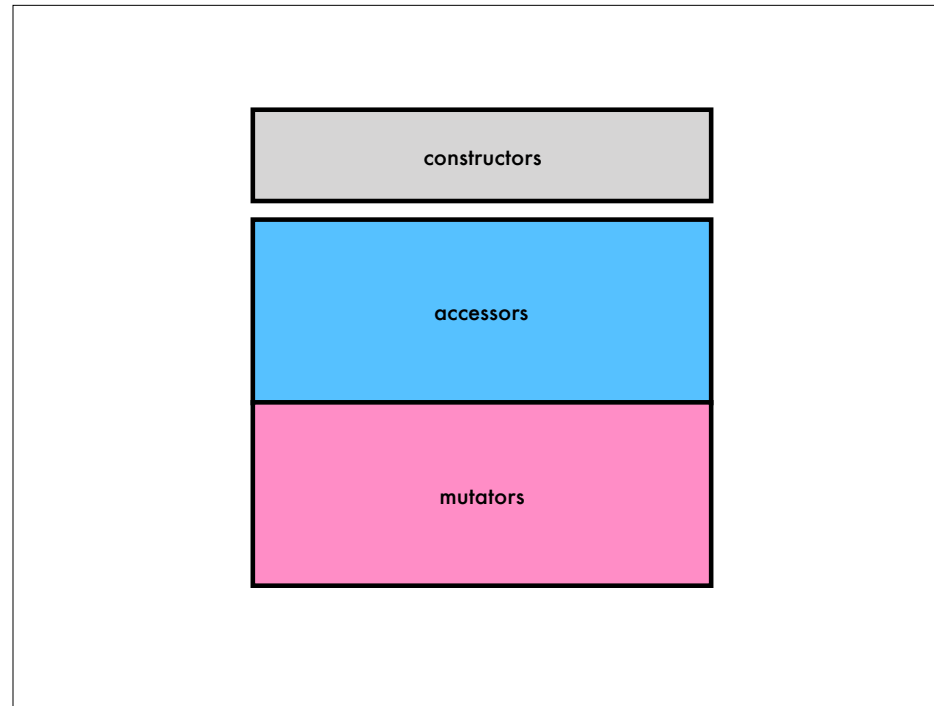
Type	Method and description	
char	<code>charAt(int index)</code> Returns the character value at the specified index.	Accessor
<code>String</code>	<code>concat(String str)</code> Concatenates the specified string to the end of this string.	Accessor
boolean	<code>contains(String str)</code> Returns true if and only if this string contains the specified string.	Accessor
char[]	<code>toCharArray()</code> Converts this string to a new character array.	Accessor
<code>String</code>	<code>toUpperCase()</code> Returns a copy of this string in which all characters are upper case.	Accessor

This one can be tricky if you actually go through each of the methods and consider whether they are returning information *about the string object*. It becomes a lot easier to figure out if you simply ask yourself: do any of these methods modify the object. The obvious answer to this question, is no. Why? Because Java strings are *immutable*. And that means that all string methods *must* be accessors. [space] This is important. If a type has even a single mutator method, it is *not* an immutable type. [pause] This might leave some people feeling a little unsatisfied. Look at the `concat` method, for example. Yes, it is obtaining information about the current string, but it's also obtaining information about the string that is passed in as an argument. And it's not returning either one of those strings. It's returning the *concatenation* of those strings. We're not modifying the current string, so obviously this is not a mutator method, but it also seems that we're doing a lot more than just obtaining information about the calling object.

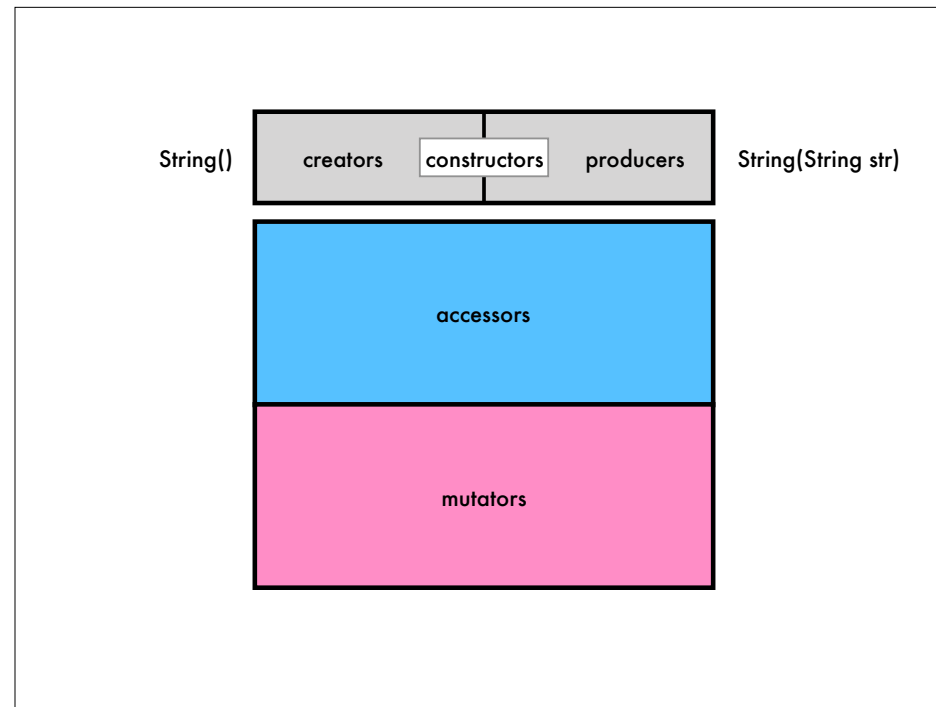
Liskov Categories

- Creators** create objects of their type from scratch
- Producers** create objects of their type based on other objects of their type
- Mutators** modify objects of their type
- Observers** return a type that is different from their type

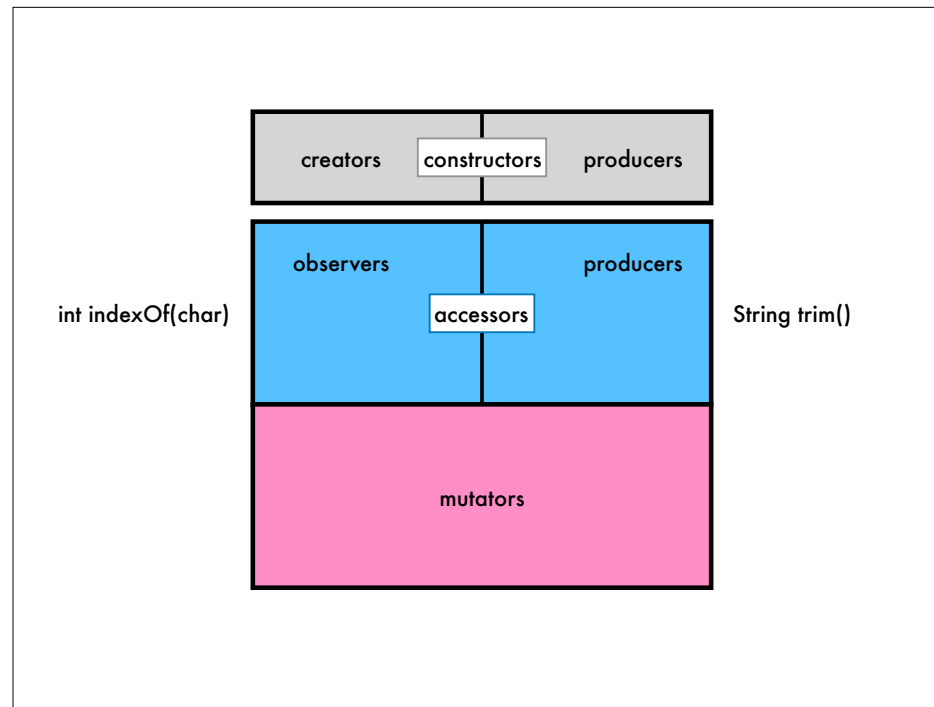
There are some Java developers out there might say that operation categories limited to accessor and mutator are simply not *rich* enough. For example, Barbara Liskov created her own categorization for operations. And who is Barbara Liskov? Barbara Liskov is a computer science professor at MIT who has done a lot of work in the area of formal methods. She came up with the Liskov substitution principle, and she is a Turing Award winner. In other words, she's pretty amazing. [pause] This is Barbara Liskov's categorization. *Creators* create objects from scratch. This category does not apply to Java methods – it only applies to Java constructors. *Producers* create objects based on other objects of their type. So if a string method creates another string, it is a producer. Producers can be constructors or they can be methods. *Mutators* modify objects of their type. This is essentially the same definition we are using for mutators. And *observers* - unlike producers - return a type that is different from their type. Since neither producer methods nor observer methods modify their objects, they are essentially just two different kinds of accessors.



Let's revisit the diagram we looked at before and figure out how Liskov's operation categories fit into the big picture. This time around, we're going to ignore getter and setter methods. However, since Liskov's categories also apply to constructors, we're going to bring them into the picture. [pause] If you look over Liskov's operation categories carefully, you'll see that constructors can either be creators or producers.



If a constructor does not take any arguments of its type, then it is a creator. An example of a creator is the **default** constructor for a string. [space] This constructor does not take any arguments, and initializes a new string object whose value is the empty string. An example of a constructor that is a producer is the **copy** constructor for a string. [space] This constructor takes an existing string as an argument, and it initializes a new string object whose value is the same as the original string. What about **accessor** methods? Well, there are also two types of accessor methods. Accessor methods can be observers or producers.



Remember, we are using accessor to mean a non-mutator - any method that does not modify an object of its type, is an accessor. The distinction between observers and producers is simple. If the method returns an object of its type, then its a producer. If the method returns anything else, then it's an observer. So the method [space] `indexOf`, which takes a character and returns the position of that character in the string, would be an observer, because it returns an integer, not a string. And the method `trim`, which returns a copy of the original string with the leading and trailing whitespace removed, would be a producer, because it *does* return a string.

Producers or Observers

Type	Method and description
char	<code>charAt(int index)</code> Returns the character value at the specified index.
<code>String</code>	<code>concat(String str)</code> Concatenates the specified string to the end of this string.
boolean	<code>contains(String str)</code> Returns true if and only if this string contains the specified string.
char[]	<code>toCharArray()</code> Converts this string to a new character array.
<code>String</code>	<code>toUpperCase()</code> Returns a copy of this string in which all characters are upper case.


Let's revisit the string methods we looked at before in light of Liskov's categories. We already figured out that since strings are immutable, all of these methods are accessors. But what kind of accessor are they? Are they producers or observers? I hope the answer is fairly obvious. I won't do a "pause and think" slide here, but I will give you 5 seconds to take a breath and collect your thoughts while you convince yourself which methods are producers and which are observers. 5 seconds starting from now. [5 seconds] Okay, all of these methods belong to the string class, so if any of them return a string object, they are producers, otherwise they are observers.

Producers or Observers

Type	Method and description	
char	<code>charAt(int index)</code> Returns the character value at the specified index.	Observer
<code>String</code>	<code>concat(String str)</code> Concatenates the specified string to the end of this string.	Producer
boolean	<code>contains(String str)</code> Returns true if and only if this string contains the specified string.	Observer
char[]	<code>toCharArray()</code> Converts this string to a new character array.	Observer
<code>String</code>	<code>toUpperCase()</code> Returns a copy of this string in which all characters are upper case.	Producer

Both `concat` and `toUpperCase` are producers, and the rest are observers. [pause] So, with Liskov we get a categorization that effectively breaks up **accessors** into producers and observers. This distinction seems to be especially useful for immutable classes like `String`, where all methods are accessors to begin with. And actually, that is **not** an accident.

Mutators and Producers



Mutators play the same role in mutable types that **producers** play in immutable ones.

Liskov

This is a direct quote from Liskov: **Mutators** play the same role in mutable types that **producers** play in immutable ones. What does she mean by that? Well, let's look at an example.

Mutators and Producers

```
/**
 * Adds the specified element
 * to the top of this stack.
 */
public void push(E element);
```

The push method in a stack type is mutable. It adds an element to the current stack object, thereby changing the *state* of that object. If someone said to you: "We need a stack type, but we don't want a *mutable* stack, we want an *immutable* stack. Is it possible to take a mutable type and turn it into an immutable type? It turns out that it is, and it's not too difficult either. You just take all the mutator methods and turn them into non-mutator methods - in other words, accessor methods. Can you see how to do that with this push method? Go ahead and pause the video now and see if you can figure it out.

Pause and Think

How would you change this **mutator** method into a **non-mutator** method?

```
/**  
 * Adds the specified element  
 * to the top of this stack.  
 */  
public void push(E element);
```

mutator

[10 seconds] Ready? Okay. The push method is mutable. It modifies the stack object by pushing an element on to it. We need a method that does not modify the stack object, but for this method to make sense we *also* need access to a stack object that is the same as the original stack except that it has that extra element. Therefore, what we're going to do is return a *new* stack with the extra element. And here is the signature for that.

Pause and Think

How would you change this **mutator** method into a **non-mutator** method?

```
/**  
 * Adds the specified element  
 * to the top of this stack.  
 */  
public void push(E element);
```

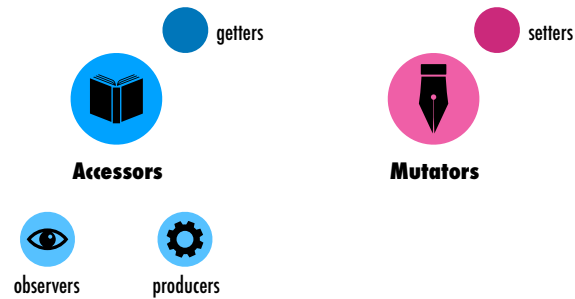
mutator

```
/**  
 * Returns a copy of this stack with the  
 * specified element added to the top.  
 */  
public Stack<E> push(E element);
```

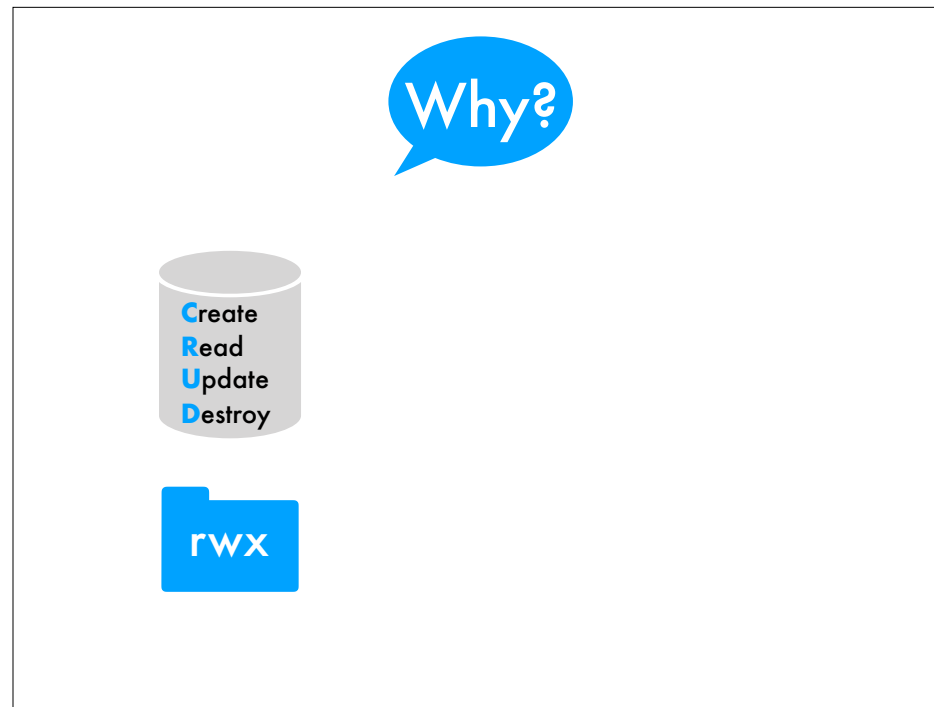
producer

The original stack is not modified, but the stack returned has the extra element. So this method is not a mutator. Instead, this is a producer method.

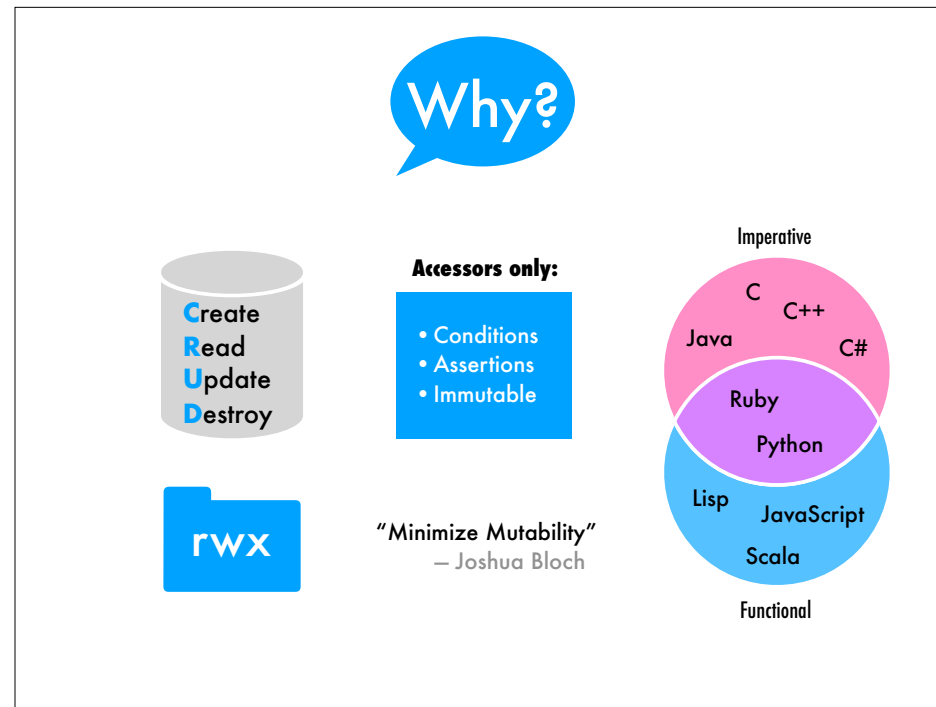
Accessors vs Mutators



Okay, so we've talked in this video mainly about the distinction between accessor methods and mutator methods. That's the primary distinction you know. [space] We've also talked a bit about getters and setters. [space] And we took a short diversion into observers and producers. But before we end the video there's one more question that we have to answer...



That question is "Why?" Why do we need to know this? How is **knowing** the distinction between accessors and mutators going to make us better Java programmers? That's a great question. And I'm going to try to give you some compelling answers. First, the notion of accessors and mutators are all over the place in computer science. [space] Databases have the basic CRUD operations - create, read, update, and destroy. Reads provide **access** to data, and updates **mutate** the data. The distinction between reads and writes is important to database maintainers. Since most of the time people just want to access information, reads are much more common. Database load balancing patterns will take advantage of this distinction. [space] File systems keep track of read-write-execute permissions on files. Without this distinction, maintaining security would be even more challenging than it is today. There are many more examples: the observer pattern distinguishes between producers and consumers. Web servers distinguish between a GET request and a POST request. I'm sure **you** can find even **more** examples if you think about it for a while.



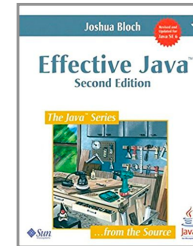
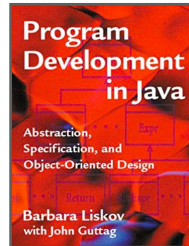
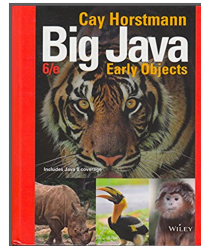
There are times when you are writing code that you should only use accessors. In IF conditions and WHILE conditions you should not change the state of the program by using mutators. The same is true of assertions in unit testing. And of course no mutators are allowed in an immutable class. Since accessors don't change anything, they are easier to reason about. [space] This is one of the reasons that Joshua Block tells us in his famous book "Effective Java" to minimize mutability. Make a class **im**-mutable unless you have a **very** good reason to make it mutable. Finally [space] the whole distinction between immutable and mutable objects is related to the distinction between functional and imperative languages. Functional language encourage immutable objects. In fact, pure functional language prohibit mutable objects. Imperative languages make it easy to create mutable types. So they give you more flexibility, but the code also tends to be more difficult to reason about.

Take-Aways

- **Accessors** read information while **Mutators** modify objects
- Getters get properties and setters set them
- Some use **accessor** to refer to both getters and setters
- Richer categorizations account for other method types
- Immutable classes do not have mutator methods!

So what do you need to take away from this discussion? First and foremost, [space] you need to know that accessors read information, and mutators modify objects. [space] You also need to know that getters **get** property values, and setters set them. [space] When you hear other Java programmers talk about both getters and setters as accessor methods, you should know that they are not using accessor in the same sense that we are using it here. The programmers who do this, probably work with Java Beans. [space] You should understand that there are other method categorizations out there, but you don't have to worry too much about what they are. I personally think that Liskov's operation categories are very useful and relevant, but if you start talking about observers vs producers to other Java programmers, you are probably going to get a lot of blank stares. Finally, [space] remember that immutable classes do **not** have mutator methods, because even one mutator makes the class mutable.

Look It Up



And that's it for this "Java short cut" on accessor methods vs mutator methods. If you are interested in learning more about some of the topics we covered here, you might want to take a look at one - or even all - of these books. Thanks for watching. I'll see you in the next video.