CONTENTS

// TUTORIAL //

Java HashMap - HashMap in Java

Published on August 4, 2022

Java

Pankaj

Java HashMap is one of the most popular Collection classes in java. Java HashMap is Hash table based implementation. HashMap in java extends AbstractMap class that implements Map interface.

Java HashMap

Some of the important points about HashMap in Java are;

1. Java HashMap allows null key and null values.
2. HashMap is not an ordered collection. You can iterate over HashMap entries through keys set but they are not guaranteed to be in the order of their addition to the HashMap.
3. HashMap is almost similar to Hashtable except that it's unsynchronized and allows null key and values.
4. HashMap uses it's inner class Node<K,V> for storing map entries.
5. HashMap stores entries into multiple singly linked lists, called buckets or bins. Default number of bins is 16 and it's always power of 2.
6. HashMap uses hashCode() and equals() methods on keys for get and put operations. So HashMap key object should provide good implementation of these methods. This is the reason immutable classes are better suitable for keys, for example String and Interger.
7. Java HashMap is not thread safe, for multithreaded environment you should use ConcurrentHashMap class or get synchronized map using `Collections.synchronizedMap()` method.

Java HashMap Constructors

Java HashMap provides four constructors.

1. **public HashMap()**: Most commonly used HashMap constructor. This constructor will create an empty HashMap with default initial capacity 16 and load factor 0.75
2. **public HashMap(int initialCapacity)**: This HashMap constructor is used to specify the initial capacity and 0.75 load factor. This is useful in avoiding rehashing if you know the number of mappings to be stored in the HashMap.
3. **public HashMap(int initialCapacity, float loadFactor)**: This HashMap constructor will create an empty HashMap with specified initial capacity and load factor. You can use this if you know the maximum number of mappings to be stored in HashMap. In common scenarios you should avoid this because load factor 0.75 offers a good tradeoff between space and time cost.
4. **public HashMap(Map<? extends K, ? extends V> m)**: Creates a Map having same mappings as the specified map and with load factor 0.75

Java HashMap Constructors Example

Below code snippet is showing HashMap example of using all the above constructors.

```
Map<String, String> map1 = new HashMap<>();

Map<String, String> map2 = new HashMap<>(2^5);

Map<String, String> map3 = new HashMap<>(32,0.80f);

Map<String,String> map4 = new HashMap<>(map1);
```

Java HashMap Methods

Let's have a look at the important methods of HashMap in java.

1. **public void clear()**: This HashMap method will remove all the mappings and HashMap will become empty.
2. **public boolean containsKey(Object key)**: This method returns 'true' if the key exists otherwise it will return 'false'.
3. **public boolean containsValue(Object value)**: This HashMap method returns true if the value exists otherwise false.
4. **public Set<Map.Entry<K,V>> entrySet()**: This method returns a Set view of the HashMap mappings. This set is backed by the map, so changes to the map are reflected in the set, and vice-versa.
5. **public V get(Object key)**: Returns the value mapped to the specified key, or null if there is no mapping for the key.
6. **public boolean isEmpty()**: A utility method returning true if no key-value mappings are present.
7. **public Set<K> keySet()**: Returns a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-

versa.

8. **public V put(K key, V value)**: Associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced.
9. **public void putAll(Map<? extends K, ? extends V> m)**: Copies all of the mappings from the specified map to this map. These mappings will replace any mappings that this map had for any of the keys currently in the specified map.
10. **public V remove(Object key)**: Removes the mapping for the specified key from this map if present.
11. **public int size()**: Returns the number of key-value mappings in this map.
12. **public Collection<V> values()**: Returns a Collection view of the values contained in this map. The collection is backed by the map, so changes to the map are reflected in the collection, and vice-versa.

There are many new methods in HashMap introduced in Java 8.

1. **public V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)**: If the specified key is not already associated with a value (or is mapped to null), this method attempts to compute its value using the given mapping function and enters it into the HashMap unless Null.
2. **public V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)**: If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
3. **public V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)**: This HashMap method attempts to compute a mapping for the specified key and its current mapped value.
4. **public void forEach(BiConsumer<? super K, ? super V> action)**: This method performs the given action for each entry in this map.
5. **public V getOrDefault(Object key, V defaultValue)**: Same as get except that defaultValue is returned if no mapping found for the specified key.
6. **public V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)**: If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. Otherwise, replaces the associated value with the results of the given remapping function, or removes if the result is null.
7. **public V putIfAbsent(K key, V value)**: If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.
8. **public boolean remove(Object key, Object value)**: Removes the entry for the specified key only if it is currently mapped to the specified value.
9. **public boolean replace(K key, V oldValue, V newValue)**: Replaces the entry for the specified key only if currently mapped to the specified value.
10. **public V replace(K key, V value)**: Replaces the entry for the specified key only if it is currently mapped to some value.
11. **public void replaceAll(BiFunction<? super K, ? super V, ? extends V> function)**: Replaces each entry's value with the result of invoking the given function on that

```
System.out.println( key = 5, value=  + value);

boolean keyExists = map.containsKey(null);
boolean valueExists = map.containsValue("100");

System.out.println("keyExists=" + keyExists + ", valueExists=" + valu

Set<Entry<String, String>> entrySet = map.entrySet();
System.out.println(entrySet);

System.out.println("map size=" + map.size());

Map<String, String> map1 = new HashMap<>();
map1.putAll(map);
System.out.println("map1 mappings= " + map1);

String nullKeyValue = map1.remove(null);
System.out.println("map1 null key value = " + nullKeyValue);
System.out.println("map1 after removing null key = " + map1);
```

```
                Set<String> keySet = map.keySet();
                System.out.println("map keys = " + keySet);

                Collection<String> values = map.values();
                System.out.println("map values = " + values);

                map.clear();
                System.out.println("map is empty=" + map.isEmpty());

        }

}
```

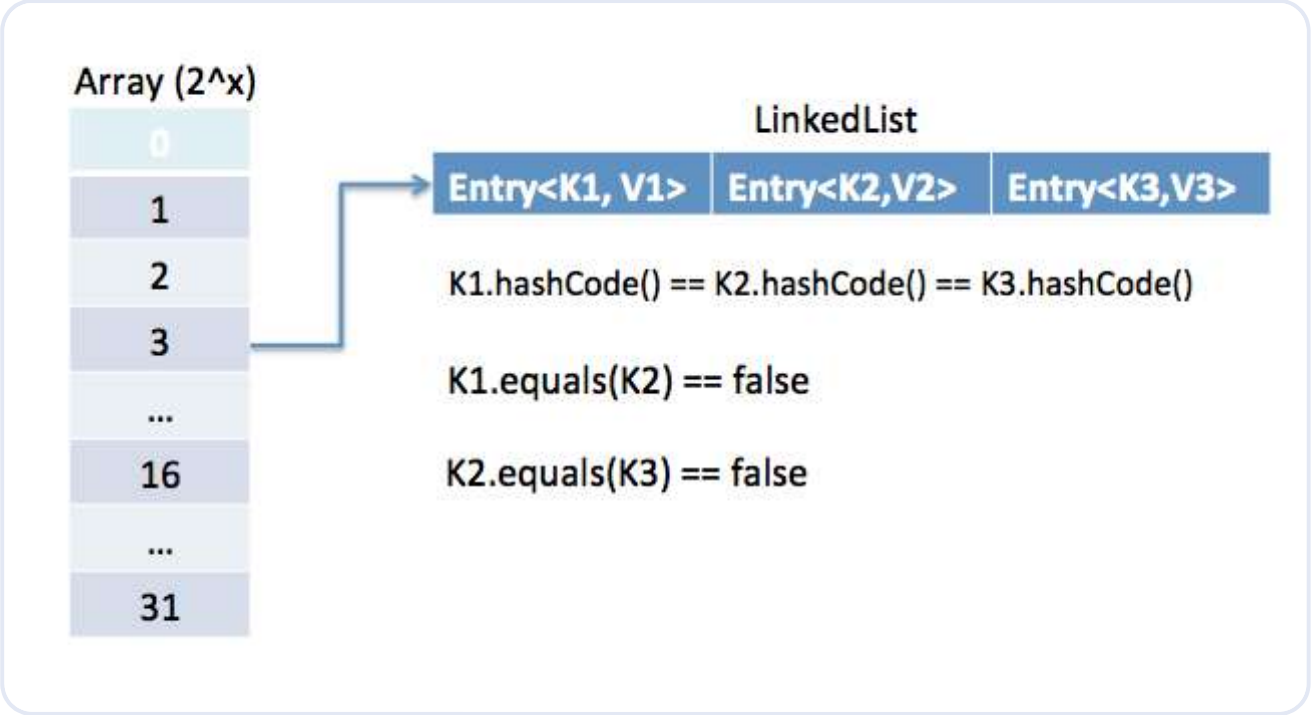
Below is the output of above Java HashMap example program.

```
Key = 3, Value = 3
Key = 5, Value=Default Value
keyExists=true, valueExists=true
[null=100, 1=1, 2=2, 3=3, 4=null]
map size=5
map1 mappings= {null=100, 1=1, 2=2, 3=3, 4=null}
map1 null key value = 100
map1 after removing null key = {1=1, 2=2, 3=3, 4=null}
map keys = [null, 1, 2, 3, 4]
map values = [100, 1, 2, 3, null]
map is empty=true
```

How HashMap works in java?

HashMap in java use it's inner class Node<K,V> for storing mappings. HashMap works on hashing algorithm and uses hashCode() and equals() method on key for get and put operations. HashMap use singly linked list to store elements, these are called bins or buckets. When we call put method, hashCode of key is used to determine the bucket that will be used to store the mapping. Once bucket is identified, hashCode is used to check if there is already a key with same hashCode or not. If there is an existing key with same hashCode, then equals() method is used on key. If equals returns true, then value is overwritten, otherwise a new mapping is made to this singly linked list bucket. If there is no key with same hashCode then mapping is inserted into the bucket. For HashMap get operation, again key hashCode is used to determine the bucket to look for the value. After bucket is identified, entries are traversed to find out the Entry using hashCode and equals method. If match is found, value is returned otherwise null is returned. There are much more things involved such as hashing algorithm to get the bucket for the key, rehashing of mappings etc. But for our working, just remember that HashMap operations work on Key and good implementation of hashCode and equals method is required to

avoid unwanted behaviour. Below image shows the explanation of get and put operations.



Recommended Read: hashCode and equals method importance in Java

Java HashMap Load Factor

Load Factor is used to figure out when HashMap will be rehashed and bucket size will be increased. Default value of bucket or capacity is 16 and load factor is 0.75. Threshold for rehashing is calculated by multiplying capacity and load factor. So default threshold value will be 12. So when the HashMap will have more than 12 mappings, it will be rehashed and number of bins will be increased to next of power 2 i.e 32. Note that HashMap capacity is always power of 2. Default load factor of 0.75 provides good tradeoff between space and time complexity. But you can set it to different values based on your requirement. If you want to save space, then you can increase it's value to 0.80 or 0.90 but then get/put operations will take more time.

Java HashMap keySet

Java HashMap keySet method returns the Set view of keys in the HashMap. This Set view is backed by HashMap and any changes in HashMap is reflected in Set and vice versa. Below is a simple program demonstrating HashMap keySet examples and what is the way to go if you want a keySet not backed by map.

```
package com.journaldev.examples;

import java.util.HashMap;
```

```
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

public class HashMapKeySetExample {

	public static void main(String[] args) {

		Map<String, String> map = new HashMap<>();
		map.put("1", "1");
		map.put("2", "2");
		map.put("3", "3");

		Set<String> keySet = map.keySet();
		System.out.println(keySet);

		map.put("4", "4");
		System.out.println(keySet); // keySet is backed by Map

		keySet.remove("1");
		System.out.println(map); // map is also modified

		keySet = new HashSet<>(map.keySet()); // copies the key to new Set
		map.put("5", "5");
		System.out.println(keySet); // keySet is not modified
	}

}
```

Output of the above program will make it clear that keySet is backed by map.

```
[1, 2, 3]
[1, 2, 3, 4]
{2=2, 3=3, 4=4}
[2, 3, 4]
```

Java HashMap values

Java HashMap values method returns a Collection view of the values in the Map. This collection is backed by HashMap, so any changes in HashMap will reflect in values collection and vice versa. A simple example below confirms this behaviour of HashMap values collection.

```
package com.journaldev.examples;

import java.util.Collection;
```

```
import java.util.HashMap;
import java.util.Map;

public class HashMapValuesExample {

	public static void main(String[] args) {
		Map<String, String> map = new HashMap<>();
		map.put("1", "1");
		map.put("2", "2");
		map.put("3", null);
		map.put("4", null);
		map.put(null, "100");

		Collection<String> values = map.values();
		System.out.println("map values = " + values);

		map.remove(null);
		System.out.println("map values after removing null key = " + values);

		map.put("5", "5");
		System.out.println("map values after put = " + values);

		System.out.println(map);
		values.remove("1"); // changing values collection
		System.out.println(map); // updates in map too

	}

}
```

Output of above program is below.

```
map values = [100, 1, 2, null, null]
map values after removing null key = [1, 2, null, null]
map values after put = [1, 2, null, null, 5]
{1=1, 2=2, 3=null, 4=null, 5=5}
{2=2, 3=null, 4=null, 5=5}
```

Java HashMap entrySet

Java HashMap entrySet method returns the Set view of mappings. This entrySet is backed by HashMap, so any changes in map reflects in entry set and vice versa. Have a look at the below example program for HashMap entrySet example.

```
package com.journaldev.examples;
```

```
import java.util.AbstractMap;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;

public class HashMapEntrySetExample {

	public static void main(String[] args) {
		Map<String, String> map = new HashMap<>();
		map.put("1", "1");
		map.put("2", null);
		map.put(null, "100");

		Set<Entry<String,String>> entrySet = map.entrySet();
		Iterator<Entry<String, String>> iterator = entrySet.iterator();
		Entry<String, String> next = null;

		System.out.println("map before processing = "+map);
		System.out.println("entrySet before processing = "+entrySet);
		while(iterator.hasNext()){
			next = iterator.next();
			System.out.println("Processing on: "+next.getValue());
			if(next.getKey() == null) iterator.remove();
		}

		System.out.println("map after processing = "+map);
		System.out.println("entrySet after processing = "+entrySet);

		Entry<String, String> simpleEntry = new AbstractMap.SimpleEntry<Strir
		entrySet.remove(simpleEntry);
		System.out.println("map after removing Entry = "+map);
		System.out.println("entrySet after removing Entry = "+entrySet);
	}

}
```

Below is the output produced by above program.

```
map before processing = {null=100, 1=1, 2=null}
entrySet before processing = [null=100, 1=1, 2=null]
Processing on: 100
Processing on: 1
Processing on: null
map after processing = {1=1, 2=null}
entrySet after processing = [1=1, 2=null]
map after removing Entry = {2=null}
entrySet after removing Entry = [2=null]
```

Java HashMap putIfAbsent

A simple example for HashMap putIfAbsent method introduced in Java 8.

```
package com.journaldev.examples;

import java.util.HashMap;
import java.util.Map;

public class HashMapPutIfAbsentExample {

	public static void main(String[] args) {
		Map<String, String> map = new HashMap<>();
		map.put("1", "1");
		map.put("2", null);
		map.put(null, "100");

		System.out.println("map before putIfAbsent = "+map);
		String value = map.putIfAbsent("1", "4");
		System.out.println("map after putIfAbsent = "+map);
		System.out.println("putIfAbsent returns: "+value);

		System.out.println("map before putIfAbsent = "+map);
		value = map.putIfAbsent("3", "3");
		System.out.println("map after putIfAbsent = "+map);
		System.out.println("putIfAbsent returns: "+value);
	}

}
```

Output of above program is;

```
map before putIfAbsent = {null=100, 1=1, 2=null}
map after putIfAbsent = {null=100, 1=1, 2=null}
putIfAbsent returns: 1
map before putIfAbsent = {null=100, 1=1, 2=null}
map after putIfAbsent = {null=100, 1=1, 2=null, 3=3}
putIfAbsent returns: null
```

Java HashMap forEach

HashMap forEach method is introduced in Java 8. It's a very useful method to perform the given action for each entry in the map until all entries have been processed or the action throws an exception.

```
package com.journaldev.examples;

import java.util.HashMap;
import java.util.Map;
import java.util.function.BiConsumer;

public class HashMapForEachExample {

	public static void main(String[] args) {
		Map<String, String> map = new HashMap<>();
		map.put("1", "1");
		map.put("2", null);
		map.put(null, "100");

		BiConsumer<String, String> action = new MyBiConsumer();
		map.forEach(action);

		//lambda expression example
		System.out.println("\nHashMap forEach lambda example\n");
		map.forEach((k,v) -> {System.out.println("Key = "+k+", Value = "+v);});
	}

}

class MyBiConsumer implements BiConsumer<String, String> {

	@Override
	public void accept(String t, String u) {
		System.out.println("Key = " + t);
		System.out.println("Processing on value = " + u);
	}

}
```

Output of above HashMap forEach example program is;

```
Key = null
Processing on value = 100
Key = 1
Processing on value = 1
Key = 2
Processing on value = null

HashMap forEach lambda example

Key = null, Value = 100
Key = 1, Value = 1
Key = 2, Value = null
```

Java HashMap replaceAll

HashMap replaceAll method can be used to replace each entry's value with the result of invoking the given function on that entry. This method is added in Java 8 and we can use lambda expressions for this method argument.

```
package com.journaldev.examples;

import java.util.HashMap;
import java.util.Map;
import java.util.function.BiFunction;

public class HashMapReplaceAllExample {

	public static void main(String[] args) {
		Map<String, String> map = new HashMap<>();
		map.put("1", "1");
		map.put("2", "2");
		map.put(null, "100");

		System.out.println("map before replaceAll = " + map);
		BiFunction<String, String, String> function = new MyBiFunction();
		map.replaceAll(function);
		System.out.println("map after replaceAll = " + map);

		// replaceAll using lambda expressions
		map.replaceAll((k, v) -> {
			if (k != null) return k + v;
			else return v;});
		System.out.println("map after replaceAll lambda expression = " + map);

	}

}

class MyBiFunction implements BiFunction<String, String, String> {

	@Override
	public String apply(String t, String u) {
		if (t != null)
			return t + u;
		else
			return u;
	}

}
```

Output of above HashMap replaceAll program is;

```
map before replaceAll = {null=100, 1=1, 2=2}
map after replaceAll = {null=100, 1=11, 2=22}
map after replaceAll lambda example = {null=100, 1=111, 2=222}
```

Java HashMap computeIfAbsent

HashMap computeIfAbsent method computes the value only if key is not present in the map. After computing the value, it's put in the map if it's not null.

```
package com.journaldev.examples;

import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

public class HashMapComputeIfAbsent {

	public static void main(String[] args) {
		Map<String, String> map = new HashMap<>();
		map.put("1", "10");
		map.put("2", "20");
		map.put(null, "100");

		Function<String, String> function = new MyFunction();
		map.computeIfAbsent("3", function); //key not present
		map.computeIfAbsent("2", function); //key already present

		//lambda way
		map.computeIfAbsent("4", v -> {return v;});
		map.computeIfAbsent("5", v -> {return null;}); //null value won't get
		System.out.println(map);
	}

}

class MyFunction implements Function<String, String> {

	@Override
	public String apply(String t) {
		return t;
	}

}
```

Output of above program is;

```
{null=100, 1=10, 2=20, 3=3, 4=4}
```

Java HashMap computeIfPresent

Java HashMap computeIfPresent method recomputes the value if the specified key is present and value is not-null. If the function returns null, the mapping is removed.

```
package com.journaldev.examples;

import java.util.HashMap;
import java.util.Map;
import java.util.function.BiFunction;

public class HashMapComputeIfPresentExample {

	public static void main(String[] args) {
		Map<String, String> map = new HashMap<>();
		map.put("1", "10");
		map.put("2", "20");
		map.put(null, "100");
		map.put("10", null);

		System.out.println("map before computeIfPresent = " + map);
		BiFunction<String, String, String> function = new MyBiFunction1();
		for (String key : map.keySet()) {
			map.computeIfPresent(key, function);
		}

		System.out.println("map after computeIfPresent = " + map);
		map.computeIfPresent("1", (k,v) -> {return null;}); // mapping will b
		System.out.println("map after computeIfPresent = " + map);

	}

}

class MyBiFunction1 implements BiFunction<String, String, String> {

	@Override
	public String apply(String t, String u) {
		return t + u;
	}

}
```

Output produced by HashMap computeIfPresent example is;

```
map before computeIfPresent = {null=100, 1=10, 2=20, 10=null}
map after computeIfPresent = {null=null100, 1=110, 2=220, 10=null}
map after computeIfPresent = {null=null100, 2=220, 10=null}
```

Java HashMap compute

If you want to apply a function on all the mappings based on it's key and value, then compute method should be used. If there is no mapping and this method is used, value will be null for compute function.

```
package com.journaldev.examples;

import java.util.HashMap;
import java.util.Map;

public class HashMapComputeExample {

	public static void main(String[] args) {
		Map<String, String> map = new HashMap<>();
		map.put("1", "1");
		map.put("2", "2");
		map.put(null, "10");
		map.put("10", null);

		System.out.println("map before compute = "+map);
		for (String key : map.keySet()) {
			map.compute(key, (k,v) -> {return k+v;});
		}
		map.compute("5", (k,v) -> {return k+v;}); //key not present, v = null
		System.out.println("map after compute = "+map);
	}

}
```

Output of HashMap compute example is;

```
map before compute = {null=10, 1=1, 2=2, 10=null}
map after compute = {null=null10, 1=11, 2=22, 5=5null, 10=10null}
```

Java HashMap merge

If the specified key is not present or is associated with null, then associates it with the given non-null value. Otherwise, replaces the associated value with the results of the

given remapping function, or removes if the result is null.

```
package com.journaldev.examples;

import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;

public class HashMapMergeExample {

	public static void main(String[] args) {
		Map<String, String> map = new HashMap<>();
		map.put("1", "1");
		map.put("2", "2");
		map.put(null, "10");
		map.put("10", null);

		for (Entry<String, String> entry : map.entrySet()) {
			String key = entry.getKey();
			String value = entry.getValue();
			//merge throws NullPointerException if key or value is null
			if(key != null && value != null)
			map.merge(entry.getKey(), entry.getValue(),
					(k, v) -> {return k + v;});
		}
		System.out.println(map);

		map.merge("5", "5", (k, v) -> {return k + v;}); // key not present
		System.out.println(map);

		map.merge("1", "1", (k, v) -> {return null;}); // method return null,
		System.out.println(map);

	}

}
```

Output of above program is;

```
{null=10, 1=11, 2=22, 10=null}
{null=10, 1=11, 2=22, 5=5, 10=null}
{null=10, 2=22, 5=5, 10=null}
```

That's all for HashMap in Java, I hope that nothing important is missed. Share it with others too if you liked it. Reference: API Doc

About the authors

Pankaj Author

Still looking for an answer? Ask a question

Search for more help

Was this helpful? Yes No

Comments

JournalDev • February 7, 2019

In Java 8, after threshold Java is using Balanced Tree instead of Linked List in a hash collision so that in the worst scenario we can achieve O(log n).

- Deepali

JournalDev • March 22, 2018

There are four things we should know about HashMap before going into How HashMap works in Java. They are, 1. HashMap working principal 2.Map.Entry interface 3.hashCode() 4.equals()

- kavya

JournalDev • August 5, 2017

Very well explained... You have made steps easy to understand.

- Ramesh B S

Popular Topics

Ubuntu

Linux Basics

JavaScript

Python

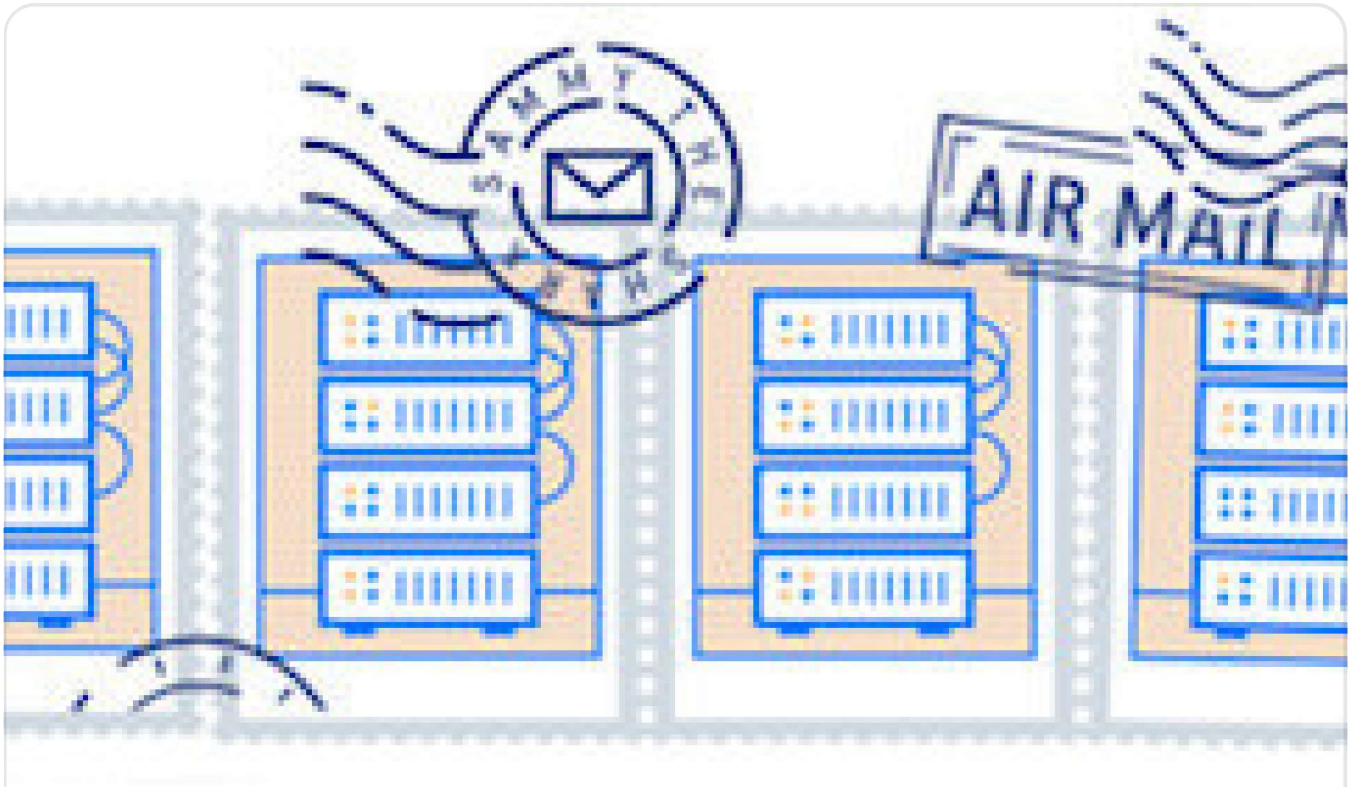
MySQL

Docker

Kubernetes

All tutorials →

Talk to an expert →



Get our biweekly newsletter

Sign up for Infrastructure as a Newsletter.

Sign up →

Hollie's Hub for Good

Working on improving health and education, reducing inequality, and spurring economic growth? We'd like to help.

Learn more →

Become a contributor

Get paid to write technical tutorials and select a tech-focused charity to receive a matching donation.

Learn more →

Featured Tutorials

Kubernetes Course Learn Python 3 Machine Learning in Python

Getting started with Go Intro to Kubernetes

DigitalOcean Products

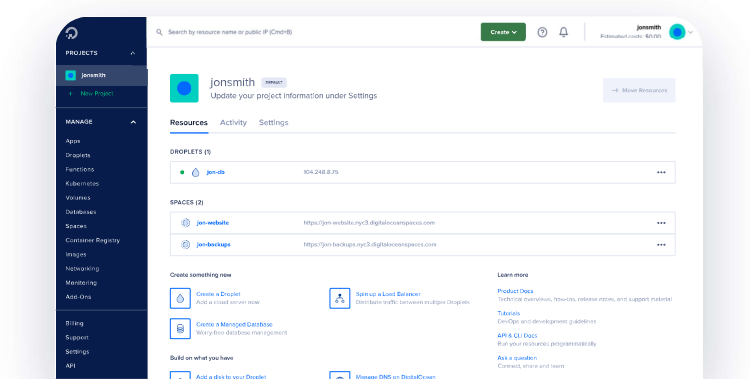
Cloudways Virtual Machines Managed Databases Managed Kubernetes

Block Storage Object Storage Marketplace VPC Load Balancers

Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow — whether you're running one virtual machine or ten thousand.

Learn more



Get started for free