# HashMap<>

Класа `HashMap<K, V>` (`java.util.HashMap`)
имплементира интерфејсе: `Serializable`, `Cloneable`, `Map<K, V>`.

Мапа је начин смештања података који минимизује потребу за претраживањем када желимо да приступимо објекту.

Сваки објекат (у даљем тексту: вредност) има придружен јединствени кључ, који се користи за одређивање где сместити референцу на објекат. Објекат се у мапу смешта заједно са својим кључем. Са датом вредношћу за кључ, увек се мање-више директно долази до објекта који одговара том кључу.

Допуштене су `null` вредности, као и `null` кључ (само један, јер кључеви морају бити јединствени).

## Процес хеширања (само увод)

Парови кључ/вредност смештају се у низ. Одговарајући индекс овог низа добија се коришћењем хешкода објекта кључа. Том приликом користи се метод `hashCode()`. Овај метод је у свакој класи наслеђен од универзалне суперкласе `Object`, међутим, у највећем броју случајева неопходно га је предефинисати у класи кључа. Кôд добијен од кључа методом `hashCode()` даље се трансформише унутар `HashMap<>` објекта како би се добила позиција у низу на којој ће бити смештен пар кључ/вредност.

Иако сваки кључ мора бити јединствен, не мора се од сваког кључа добити јединствен хешкод. Када два или више различитих кључева дају исту хеш-вредност, то се зове колизија. `HashMap<>` објекат ради са колизијама тако што смешта све кључ/вредност парове са истом хеш-вредношћу у повезану листу. То доводи до успоравања процеса смештања података и приступања подацима.

Приступ објекту код кога је приликом смештања у мапу дошло до колизије састоји се из две фазе:
кључ се хешира како би се пронашла локација на којој би пар кључ/вредност требало да буде.
Затим треба претражити повезану листу, да би се дошло до кључа који тражимо (у листи су сви они кључеви који дају исту хеш-вредност као и тражени кључ).
Зато постоји јак подстрек да се минимизује појава колизија, а цена смањења могућности колизија у хеш-табели је много празног простора у табели.

У пратећем документу: `"Kljucevi u mapi - equals hashCode.pdf"` описано је на које начине је могуће добро предефинисати метод `hashCode()` (као и метод `equals()`, што је такође неопходно) у класи објеката кључева.

Потребна је огромна пажња уколико се променљиви објекти користе као кључеви у мапи.
Понашање мапе није одређено када се објекат промени на начин који утиче на поређења у методу `equals()` док је објекат кључ у мапи.

## Остале особине мапе

Основне операције (get и put) извршавају се у константном времену под претпоставком да хеш-функција равномерно распоређује парове по низу. Итерирање кроз погледе на колекцију захтева време пропорционално збиру капацитета мапе (дужина низа) и величине мапе (број парова кључ/вредност смештених у мапу). Према томе, уколико су перформансе ове операције од значаја, веома је важно не поставити превелик иницијални капацитет (или премали load factor).

Два параметра утичу на перформансе инстанце HashMap<>: иницијални капацитет и load factor. Капацитет је број елемената хеш-табеле, а иницијални капацитет је просто капацитет хеш-табеле у тренутку њеног креирања. Load factor означава колико максимално може бити попуњена хеш-табела пре него што се њен капацитет аутоматски увећа. Када број парова кључ/вредност у табели премаши производ load factor-а и тренутног капацитета, врши се тзв. рехеширање табеле тако да се њен капацитет приближно удвостручи. Рехеширање је временски захтевна операција.

Подразумевани load factor (0.75) обезбеђује добар компромис између потребног времена и меморије. Веће вредности смањују вишак меморије, али увећавају трошкове претраге (што се одражава на већину операција, укључујући get и put). Приликом задавања иницијалног капацитета мапе потребно је узети у обзир очекивани број уноса и load factor како би се минимизовао број рехеширања. Ако је иницијални капацитет већи од максималног броја уноса подељеног load factor-ом, неће бити извршено ниједно рехеширање.

Уколико је у мапу потребно сместити много уноса, креирање мапе довољно великог капацитета омогућује ефикасније смештање уноса него аутоматско рехеширање по потреби.

`HashMap<>` није thread-safe.

Под структуралном модификацијом мапе подразумева се додавање или брисање једног или већег броја парова. Просто мењање вредности придружене кључу који је већ у мапи не спада у структуралну модификацију.
Итератори које враћају методи свих погледа на ову колекцију су *„fail-fast“*: уколико се врши структурална модификација мапе након креирања итератора на било који други начин осим помоћу метода `remove()` самог итератора, итератор избацује `ConcurrentModificationException`.

Како класа имплементира интерфејс `Map<>`, то се објекат може реферисати променљивом типа `Map<>`.

Следи попис метода ове класе (нису побројани сви, али сви нама важни јесу):

**конструктори**

**HashMap()**
без аргумената, креира празну мапу подразумеваног иницијалног капацитета (16) и подразумеваног load factor-а (0.75).

```
HashMap<String, Osoba> mapa = new HashMap<>();  // zagradice <> su obavezne!
                    // prazna mapa; kljucevi su tipa String, a vrednosti tipa Osoba
```

**HashMap(int initialCapacity)**
креира празну мапу задатог иницијалног капацитета и подразумеваног load factor-а (0.75).
Избацује IllegalArgumentException ако је иницијални капацитет негативан.

Како би парови били што равномерније распоређивани по табели, када сами задајемо њен капацитет, требало би да користимо просте бројеве.

```
HashMap<String, Osoba> mapa1 = new HashMap<>(151);
```

**HashMap(int initialCapacity, float loadFactor)**
креира празну мапу задатог иницијалног капацитета и load factor-а.
Избацује IllegalArgumentException ако је иницијални капацитет негативан или load factor није позитиван.

Број смештених парова никада не може достићи капацитет. Увек је потребно одвојити капацитет за ефикасност операција. Са недовољно тог додатог капацитета, колизије постају вероватније. Количина додатног капацитета одређена је load factor-ом.

```
HashMap<String, Osoba> mapa2 = new HashMap<>(151, 0.6f);
```

**HashMap(Map<? extends K, ? extends V> m)**
креира мапу која садржи исте парове као и дата мапа `m`. Мапа се креира са подразумеваним load factor-ом (0.75) и иницијалним капацитетом довољним за смештај свих парова задате мапе.
Избацује NullPointerException ако је аргумент null.

**Методи за смештање, приступ и уклањање објеката**

**V put(K key, V value)**
смешта објекат value у мапу коришћењем кључа key. Ако је мапа претходно садржала пар за дати кључ, value ће заменити стару вредност. Метод враћа вредност претходно придружену кључу или null уколико кључ није коришћен за смештање вредности у мапу. (Повратна вредност null такође може указивати на то да је null вредност претходно придружена кључу.)

**void putAll(Map<? extends K, ? extends V> m)**
копира све парове из задате у текућу мапу замењујући све вредности за кључеве који већ постоје.
Избацује NullPointerException ако је аргумент null.

**V remove(Object key)**
уклања пар за дати кључ, ако постоји у мапи. Враћа референцу на вредност претходно придружену кључу или null уколико кључ није коришћен за смештање вредности у мапу. (Повратна вредност null такође може указивати на то да је null вредност претходно придружена кључу.)

**void clear()**
уклања све парове из мапе. Након позива овог метода мапа је празна.

**V get(Object key)**
враћа референцу на вредност придружену датом кључу или null уколико кључ није коришћен за смештање вредности у мапу. Вредност остаје у табели.
Формалније, ако мапа садржи пар за кључ k и вредност v, такав да је key==null ? k==null : key.equals(k), онда овај метод враћа v, а иначе null. (Може постојати највише један такав пар.) Повратна вредност null не означава нужно да мапа не садржи пар за дати кључ, већ је такође могуће да је кључ у мапи експлицитно упарен са вредношћу null. Метод containsKey() се може користити како би се направила разлика између ова два случаја.

**boolean isEmpty()**
враћа true ако мапа не садржи парове кључ/вредност.

**int size()**
враћа број парова кључ/вредност у мапи.

**boolean containsKey(Object key)**
враћа true ако мапа садржи пар за дати кључ. Помоћу овог метода може се утврдити да ли је вредност null смештена у мапу помоћу датог кључа или тај кључ уопште није коришћен за смештање вредности у мапу.

**boolean containsValue(Object value)**
враћа true ако мапа упарује један или већи број кључева са датом вредношћу.

Неопходно је осигурати да је вредност коју врати put() једнака null. Иначе, можемо несвесно уклонити објекат који је претходно смештен у табелу коришћењем истог кључа.
Пример:

```
HashMap<String, Integer> mapa = new HashMap<>();
String kljuc = "Perica";
int vrednost = 12345;
Integer staraVrednost = null;
for(int i=0; i<4; i++)
      if((staraVrednost = mapa.put(kljuc, vrednost++)) != null)
            System.out.println("odbacili smo vrednost: " + staraVrednost);
```

Други параметар метода put() је типа Integer, па компајлер обезбеђује тзв. **autoboxing конверзију**,
тј. конверзију вредности примитивног типа (овде: int) у тип одговарајуће омотач класе (овде: Integer).

Излаз из програма је:

```
odbacili smo vrednost: 12345
odbacili smo vrednost: 12346
odbacili smo vrednost: 12347
```

Када се смешта прва вредност, ништа није смештено раније у мапу за дати кључ, па нема поруке.
За све наредне позиве метода put(), претходно смештена вредност се уклања и враћа се референца на њу.

Операција get() враћа референцу на објекат придружен кључу, али га не уклања из табеле.
Да бисмо приступили објекту и обрисали пар који га садржи из табеле, морамо користити метод remove():

```
int vrednost = mapa.remove(kljuc);
```

Уклања се објекат који одговара кључу и враћа референца на њега. Ако се ова наредба надовеже на претходни фрагмент кôда, биће враћена референца на објекат типа Integer, који енкапсулира вредност 12348.
Пошто то смештамо у променљиву типа int, компајлер ће убацити тзв. **unboxing конверзију**, тј. конверзију вредности типа омотач класе за примитивни тип у тај примитивни тип.

**Стринг-репрезентација мапе**

**toString()**
враћа String-репрезентацију текуће мапе у облику:
**{**<kljuc1>**=**<vrednost1>**,** <kljuc2>**=**<vrednost2>**, …,** <kljucN>**=**<vrednostN>**}**

## Процесирање свих елемената мапе

**Set<K> keySet()**
враћа скуп референци на све кључеве мапе.
У питању је поглед на садржај мапе: промене мапе одражавају се на поглед и обрнуто.
Уколико се мапа мења док је у току итерирање кроз скуп (осим помоћу метода remove() самог итератора), резултат итерације није дефинисан. Скуп подржава уклањање елемента, што уклања одговарајући пар кључ/вредност из мапе, коришћењем метода: remove() за итератор, као и метода: remove(), removeAll(),
retainAll() и clear() за скуп. Нису подржане операције add() ни addAll().

**Set<Map.Entry<K, V>> entrySet()**
враћа скуп референци на све парове кључ/вредност мапе. У питању је поглед на садржај мапе.
Кључ/вредност парови смештени су у мапи као објекти класе која имплементира интерфејс Map.Entry<K, V> дефинисан унутар интерфејса Map<K, V>.
Уколико се мапа мења док је у току итерирање кроз скуп (осим помоћу метода remove() самог итератора или метода setValue() уноса враћеног итератором), резултат итерације није дефинисан. Скуп подржава уклањање елемента, што уклања одговарајући пар кључ/вредност из мапе, коришћењем метода: remove() за итератор,
као и метода: remove(), removeAll(), retainAll() и clear() за скуп. Нису подржане операције add() ни addAll().

**Collection<V> values()**
враћа колекцију референци на све вредности мапе. У питању је поглед на садржај мапе.
Уколико се мапа мења док је у току итерирање кроз колекцију (осим помоћу метора remove() самог итератора), резултат итерације није дефинисан. Колекција подржава уклањање елемента, што уклања одговарајући пар кључ/вредност из мапе, коришћењем метода: remove() за итератор, као и метода: remove(), removeAll(), retainAll() и clear() за колекцију. Нису подржане операције add() ни addAll().

примери:

```
Set<String> kljucevi = mapa.keySet(); // skup svih kljuceva gore definisane mape
Iterator<String> kljucIter = kljucevi.iterator();
```

или, краће:

```
Iterator<String> kljucIter = mapa.keySet().iterator();
while(kljucIter.hasNext())
      System.out.println(kljucIter.next());
```

Можемо користити кључеве за издвајање вредности, али објекат Collection<> који врати метод values() обезбеђује директнији начин за то.
Пример листања вредности смештених у мапа, под претпоставком да је она типа HashMap<String, Integer>:

```
Collection<Integer> vrednosti = mapa.values();
for(Integer i: vrednosti)
      System.out.println(i);
```

Интерфејс Set<> има као суперинтерфејс Iterable<>, па можемо користити collection-based for-петљу директно за објекат који врати метод keySet():

```
Set<String> kljucevi = mapa.keySet();
for(String kljuc: kljucevi)
      System.out.println(kljuc);
```

За оперисање над **Map.Entry<K, V>** на располагању су нам следећи методи:

**K getKey()** – враћа кључ текућег Map.Entry<K, V> објекта
IllegalStateException – може, а не мора, бити избачен, уколико је пар уклоњен из оригиналне мапе.
**V getValue()** – враћа вредност текућег Map.Entry<K, V> објекта
IllegalStateException – може, а не мора, бити избачен, уколико је пар уклоњен из оригиналне мапе.
**V setValue(V new)** – поставља вредност текућег Map.Entry<K, V> објекта на задату и враћа оригиналну вредност.
Овим се мења оригинална мапа. Избацује:
UnsupportedOperationException – ако оригинална мапа не подржава операцију put.
ClassCastException – ако класа задате вредности онемогућује њено смештање у мапу.
NullPointerException – ако оригинална мапа не допушта null вредности, а аргумент је null.
IllegalArgumentException – ако неки атрибут задате вредности онемогућује њено смештање у мапу.
IllegalStateException – може, а не мора, бити избачен, уколико је пар уклоњен из оригиналне мапе.

Када имамо скуп Map.Entry<> објеката, можемо приступати кључевима и одговарајућим вредностима мапе и мењати вредност-део сваког кључ/вредност пара, ако је то потребно.

**Не очекује се да се било шта од овога учи напамет, већ да се користи помоћ коју Eclipse свесрдно пружа** након што се уради неко од подешавања из упутства **Attach Source... Source Attachment... eclipse.ini.pdf**