teamnet transforming technology

An introduction to the Java Collections Framework

Trainers:

Cristian DUMITRU
Cătălin BÎNĂ
Radu HOAGHE
Andrei MARICA
Oana BEȘLIU
Ioan aka Jonny DINU
@teamnet.ro

Author: Bogdan ȘTEFAN

Document version: 1.3

2.2.1

A case for Iterators

java.util.Iterator

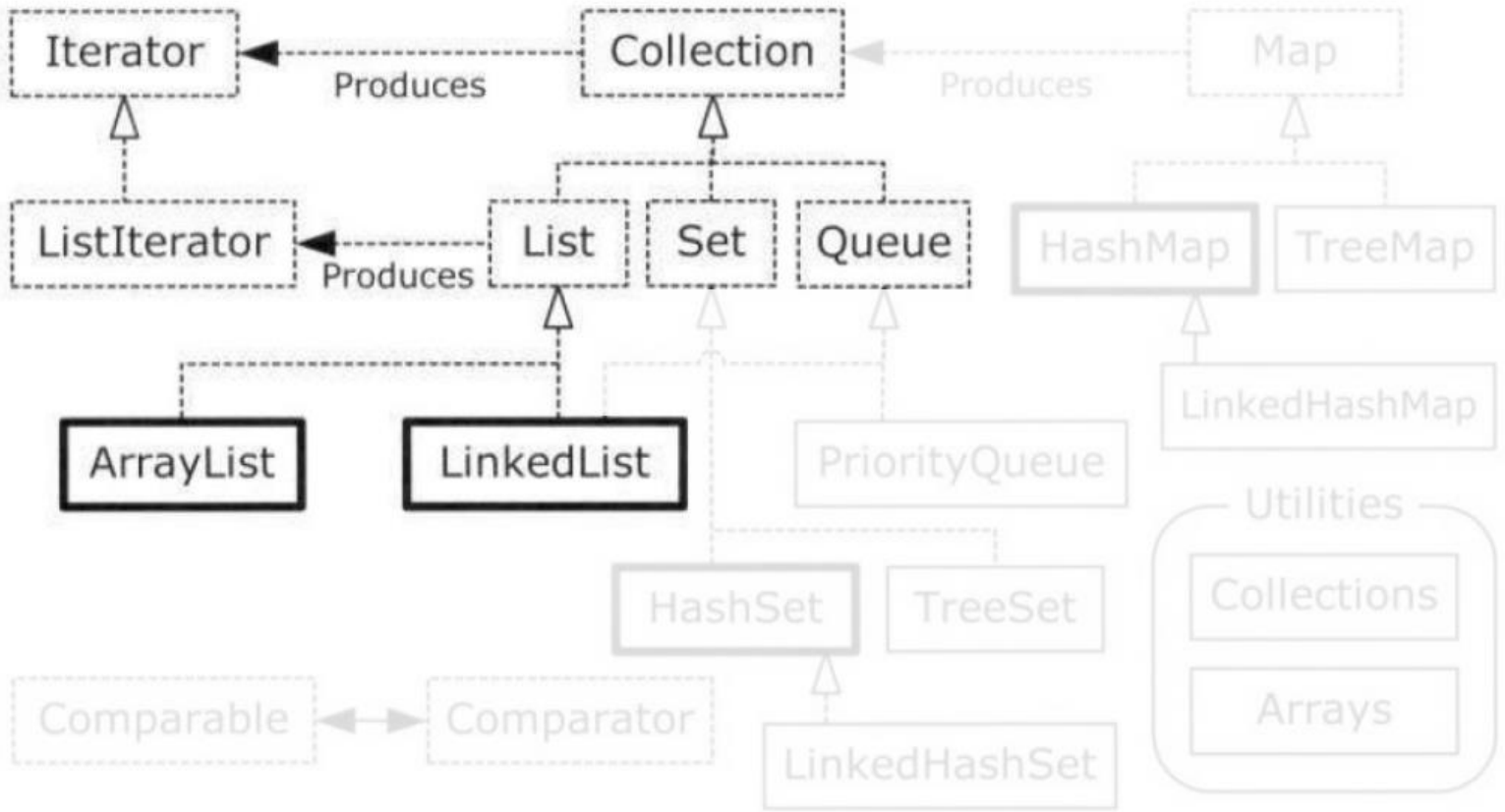


Image source: Thinking in Java (4th Edition), Bruce Eckel

java.util.Iterator – Notes (2)

- An **iterator** is a *lightweight object* that **moves** through a **sequence**.
- It **selects each element** of that **sequence** without having the programmer worry about the underlying type (i.e. enforces *loose coupling*).
- A usual interaction with an iterator would look like:
  1. Ask a `Collection` for an `Iterator`, by calling `iterator()`
  2. Get the next object in the sequence using `next()`
  3. See if there are more elements with `hasNext()`
  4. Remove the last element returned using `remove()`

java.util.Iterator – Quick example

```
public static void main(String[] args) {
    List<Pet> pets = Pets.arrayList(12);
    // Iteration via iterator
    Iterator<Pet> it = pets.iterator();
    while (it.hasNext()) {
        Pet p = it.next();
        System.out.print(p.id() + ":" + p + " ");
    }
    System.out.println();
    // A simpler approach, when possible:
    for (Pet p : pets)
        System.out.print(p.id() + ":" + p + " ");
    System.out.println();
    // An Iterator can also remove elements:
    it = pets.iterator();
    for (int i = 0; i < 6; i++) {
        it.next();
        it.remove();
    }
    System.out.println(pets);
}
```

ask for the collection's Iterator

if there are elements in the sequence

retrieve an element

use *foreach* when *reading*

remove the current element

Code source: Thinking in Java (4th Edition), Bruce Eckel

java.util.Iterator – A (better) typical use case

```
public class CrossContainerIteration {
    public static void display(Iterator<Pet> it) {
        while (it.hasNext()) {
            Pet p = it.next();
            System.out.print(p.id() + ":" + p + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        ArrayList<Pet> pets = Pets.arrayList(8);
        LinkedList<Pet> petsLL = new LinkedList<Pet>(pets);
        HashSet<Pet> petsHS = new HashSet<Pet>(pets);
        TreeSet<Pet> petsTS = new TreeSet<Pet>(pets);
        display(pets.iterator());
        display(petsLL.iterator());
        display(petsHS.iterator());
        display(petsTS.iterator());
    }
}
```

if there are elements in the sequence

retrieve an element via next()

ask for each container's Iterator

Code source: Thinking in Java (4th Edition), Bruce Eckel

2.4

Maps in Java

Image and logos used may be subject to copyright and are trademarks of their respective owners.

The java.util “toolbox”

Here’s an overview of the most often used Java containers:

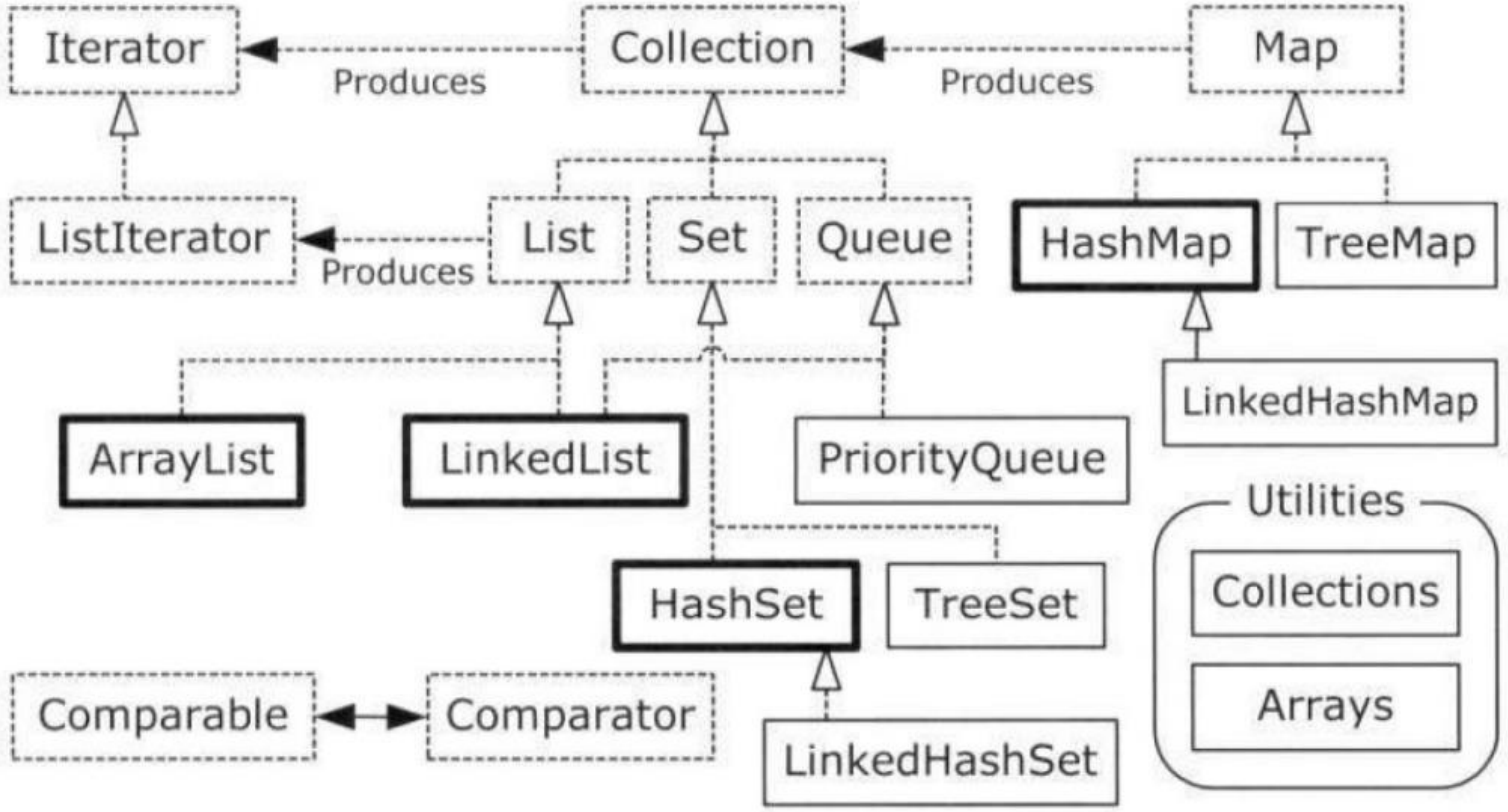


Image source: Thinking in Java (4th Edition), Bruce Eckel

java.util.Map(s)

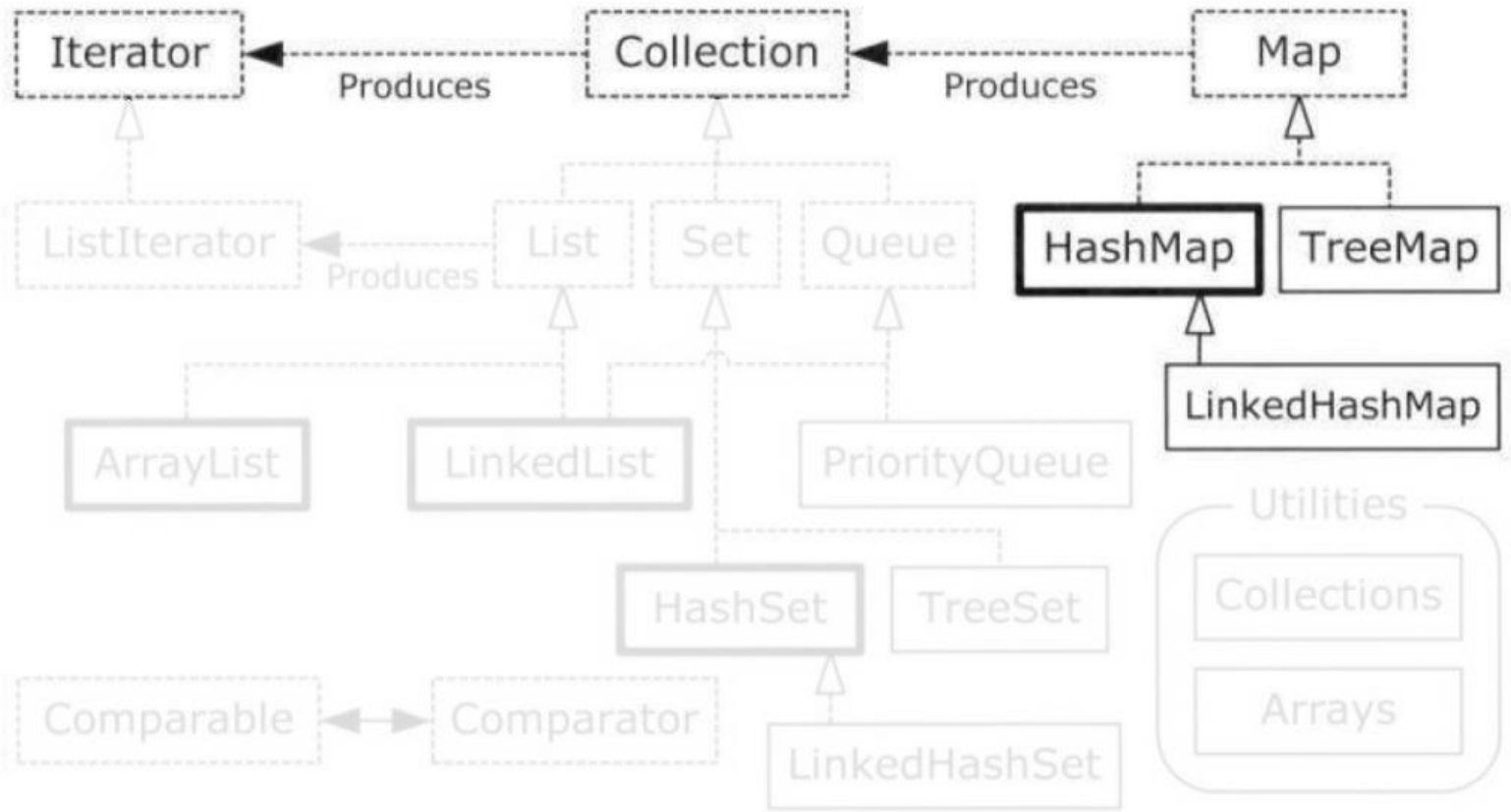
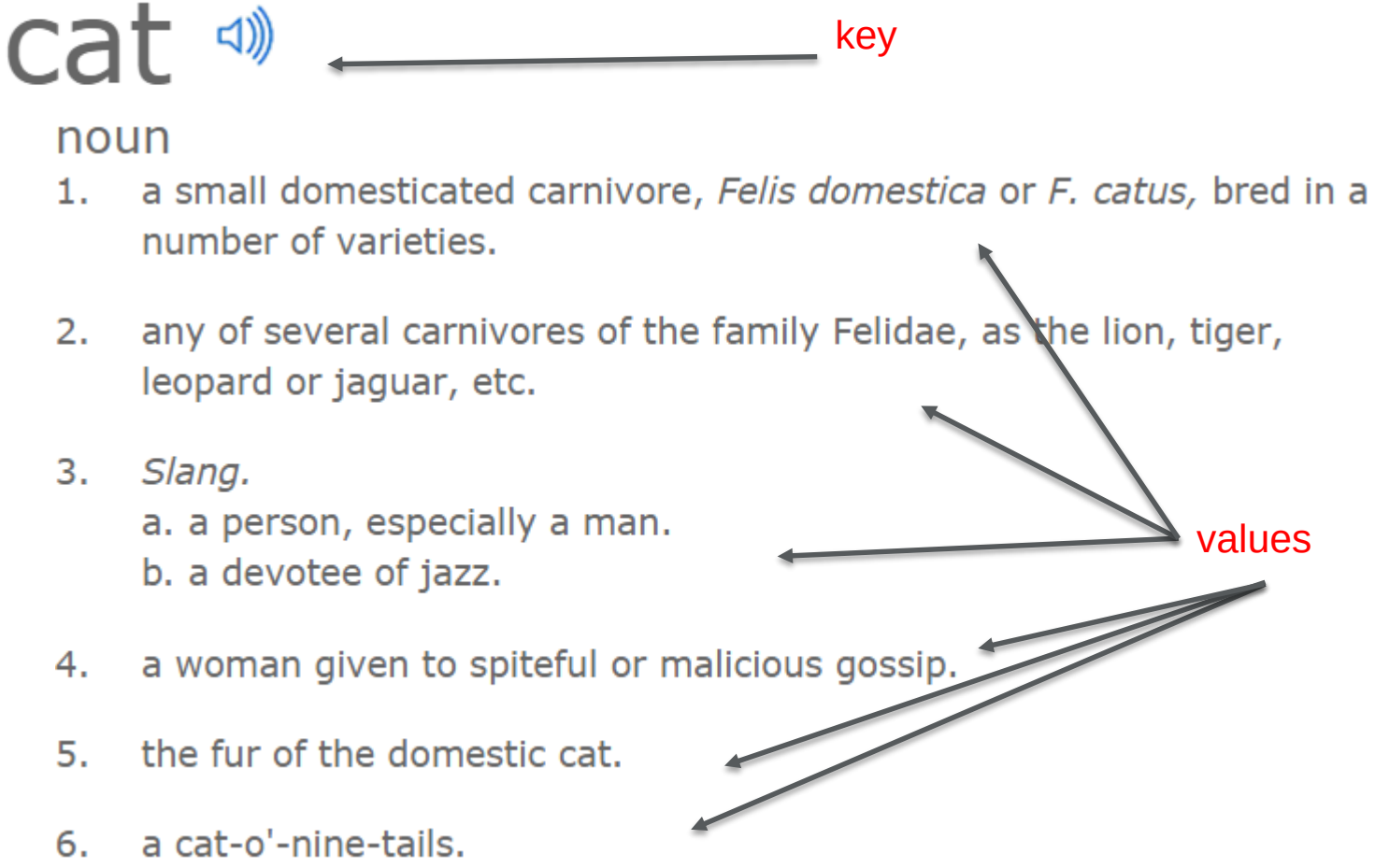


Image source: Thinking in Java (4th Edition), Bruce Eckel

java.util.Map(s) – Notes

- Allows for a way to easily **associate objects with** other **objects**.
- It works on the principle of a **dictionary**: a **key maps** to one (or more) **associated value**(s).
- Maps use an inner class to store data: **`Entry<K,V>`**
- A **Map** can **return** a **`Set`** of its **keys**, a **`Collection`** of its **values** or a **`Set`** of its pairs (i.e. **entries**).
- **Automatic resizing** to accommodate new keys, if needed.

java.util.Map(s) – Notes (2)



java.util.Map(s) – Quick example

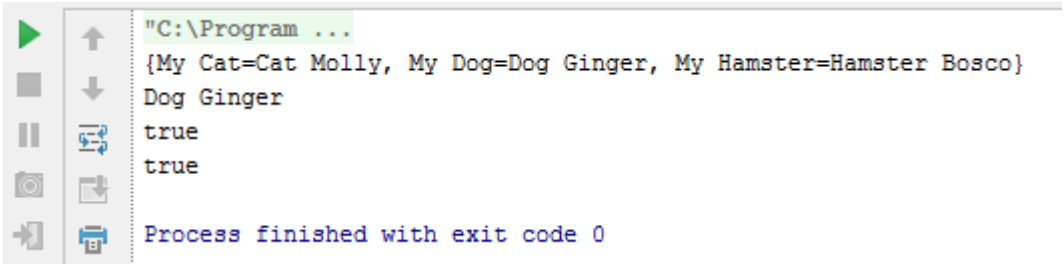
```
public static void main(String[] args) {
    Map<String, Pet> petMap = new HashMap<String, Pet>();

    petMap.put("My Cat", new Cat("Molly"));
    petMap.put("My Dog", new Dog("Ginger"));
    petMap.put("My Hamster", new Hamster("Bosco"));

    System.out.println(petMap);
    Pet dog = petMap.get("My Dog");
    System.out.println(dog);

    System.out.println(petMap.containsKey("My Dog"));
    System.out.println(petMap.containsValue(dog));
}
```

declaration establishes bounds on `<Key, Value>`

insert items via `put(key, value)`

retrieve an item via `get(key)`

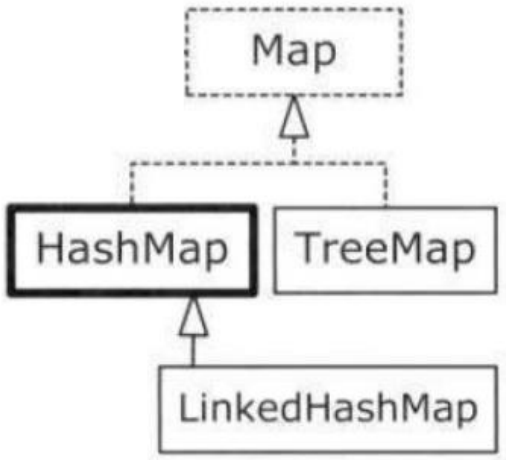
keys are stored as a `Set`; values as a `Collection`



teamnet

Code source: Thinking in Java (4th Edition), Bruce Eckel

java.util.Map(s)



Maps are available in **many** flavors. The **three** most used are:

- `HashMap`
- `LinkedHashMap`
- `TreeMap`

Legacy:

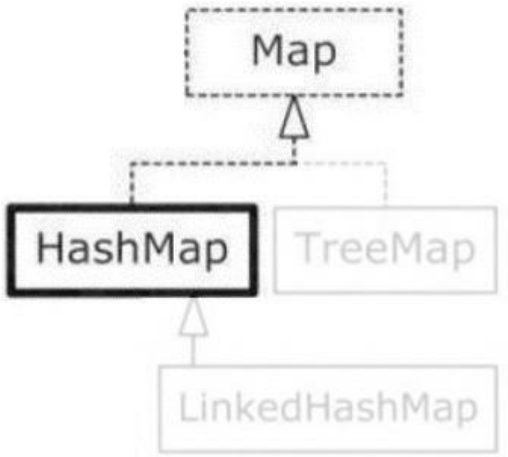
- `Hashtable`

  (old school, but offers **thread-safety**; now **replaced** by `ConcurrentHashMap`)

When and why would one use such data structures?

Image source: Thinking in Java (4th Edition), Bruce Eckel

java.util.HashMap



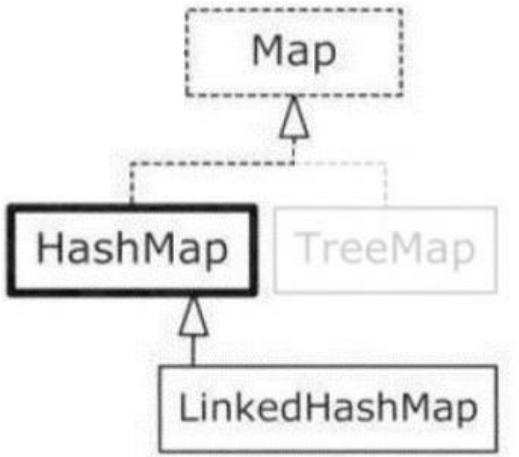
Implementation is based on the concept of a **hashtable**.

Insertion and locating of held pairs is done in near constant time – favors **lookup speed**.

Elements *appear* in no apparent **order**, because of hashing.

Image source: Thinking in Java (4th Edition), Bruce Eckel

java.util.LinkedHashMap



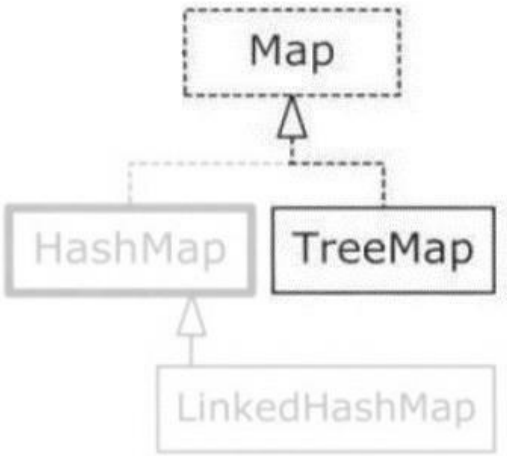
Similar to a `HashMap`, but keys are stored based on **insertion order**.

Can be tweaked (through a constructor param) to permit LRU behavior – useful for building *caches*.

Faster when **iterating** than a `HashMap`, because of underlying linked list used to keep internal order.

Image source: Thinking in Java (4th Edition), Bruce Eckel

java.util.TreeMap



Underlying implementation is a **red-black tree** (holds entries, or *pairs*).

The pairs are stored in **sorted order**, based on a `Comparator`.

Slower than `HashMap` and `LinkedHashMap`.

Image source: Thinking in Java (4th Edition), Bruce Eckel

java.util.TreeMap – Quick example

```
public static void printKeys(Map<Integer, String> map) {
    System.out.println("Size = " + map.size() + ", ");
    System.out.println("Keys: ");
    System.out.println(map.keySet()); // Produce a Set of the keys
}

public static void test(Map<Integer, String> map) {
    System.out.println(map.getClass().getSimpleName());
    // Map has 'Set' behavior for keys:
    map.putAll(new CountingMapData(25));
    // Thus, no duplicate keys are added
    map.putAll(new CountingMapData(25));
    printKeys(map);
    // Producing a Collection of the values:
    System.out.println("Values: ");
    System.out.println(map.values());

    // Operations on the Set change the Map:
    Set<Integer> keySet = map.keySet();
    keySet.removeAll(map.keySet()); // A goofy alternative to map.clear() :)
    System.out.println("map.isEmpty(): " + map.isEmpty());
}

public static void main(String[] args) {
    test(new TreeMap<Integer, String>());
}
```

a method that prints the key set nicely (works w/ any Map implementation)

retrieve implementation name

we try to add duplicate keys

retrieve values as a collection

retrieve underlying key set and modify it

Code source: Thinking in Java (4th Edition), Bruce Eckel

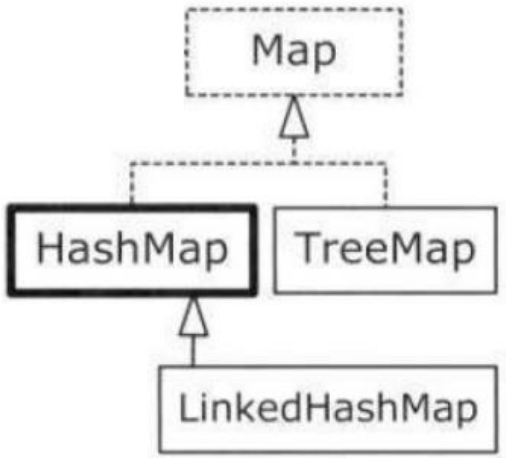
java.util.TreeMap – Quick example

```
public static void printKeys(Map<Integer, String> map) {
    System.out.println("Size = " + map.size() + ", ");
    System.out.println("Keys: ");
    System.out.println(map.keySet()); // Produce a Set of the keys
}

public static void test(Map<Integer, String> map) {
    System.out.println(map.getClass().getSimpleName());
    // Map has 'Set' behavior for keys:
    map.putAll(new CountingMapData(25));
    // Thus, no duplicate keys are added
    map.putAll(new CountingMapData(25));
    printKeys(map);
    // Producing a Collection of the values:
    System.out.println("Values: ");
    System.out.println(map.values());

    // Operations on the Set change the Map:
    Set<Integer> keySet = map.keySet();
    keySet.removeAll(map.keySet()); // A goofy alternative to map.clear() :)
    System.out.println("map.isEmpty(): " + map.isEmpty());
}

public static void main(String[] args) {
    test(new TreeMap<Integer, String>());
}
```

```
"C:\Program ...
TreeMap
Size = 25,
Keys:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
Values:
[A0, B0, C0, D0, E0, F0, G0, H0, I0, J0, K0, L0, M0, N0, O0, P0, Q0, R0, S0, T0, U0, V0, W0, X0, Y0]
map.isEmpty(): true

Process finished with exit code 0
```


Code source: Thinking in Java (4th Edition), Bruce Eckel

java.util.Map(s)



The most **common operations** you will do with/on a **Map** are:

- put(key, value)
- `get(key)`
- `entrySet().iterator()`
- `keySet()`
- `values()`
- `containsKey()`
- `containsValue()`
- `remove(key)`

Image source: Thinking in Java (4th Edition), Bruce Eckel

java.util.Map – Conclusions

- Work on the principle of a **dictionary**: a **key maps** to one (or more*) **associated value**(s).
- `HashMap(s)` are best used for **fast lookup time**.
- `LinkedHashMap(s)` have similar lookup time, and maintain an **order** based on **insertion**.
- `TreeMap(s)` focus on maintaining a **sorting order** for held keys.
- *Be aware, that the above* (`HashMap, LinkedHashMap, TreeMap`) *are not thread-safe!*

2.4.1

A word about `equals()` and `hashCode()`

`equals()` **method**

- Is inherited by all object instances from `java.lang.Object`
- Indicates whether some other object is "*equal to*" the current object, whose method is called.
- **Returns** `true` if the object is "equal to" the object that calls it and `false` otherwise

```
String s1 = "Pet";
String s2 = "Pet";
String s3 = "Pets";

System.out.println(s1.equals(s2)); // returns true
System.out.println(s1.equals(s3)); // returns false
```

`equals()` method constraints

- A proper implemented `equals()` method **must satisfy** the following **five conditions**:
  1. **Reflexive**: For any **x**, `x.equals(x)` should return `true`.
  2. **Symmetric**: For any **x** and **y**, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
  3. **Transitive**: For any **x**, **y**, and **z**, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.

`equals()` method constraints (2)

4. **Consistent:** For any **x** and **y**, multiple invocations of `x.equals(y)` **consistently return `true`** or **consistently return `false`**, provided no information used in equals comparisons on the object is modified.

5. For any **non-null x**, `x.equals(null)` should **return `false`.**

Alright, alright enough theory!

- As you can see, a proper implementation of `equals()` is essential for your own classes to work well with the Java Collection classes.
  So how does one implement `equals()` "properly"?

- Q: When are two objects equal?
  A: That depends on your application, the classes, and what you are trying to do.

`equals()` and `hashCode()` example

```
public class WeekDay {

    private final int id;
    private final String name;

    private static String[] daysNames = new String[]{"MON", "TUE", "WED", "THU", "FRI", "SAT", "SUN"};

    public WeekDay(int id) {...}

    public String getName() { return name; }

    @Override
    public String toString() { return name; }
}
```

- You could decide that two `WeekDay` objects are equal to each other if only their `ids` are *equal*.
- Or, you could decide that all fields must be used to establish equality (i.e. `id` and `name`, above), provided they are *immutable/unchangeable* (a.k.a. `final` as in above examples).

`equals()` example (2)

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    WeekDay weekDay = (WeekDay) o;

    if (id != weekDay.id) return false;
    if (!name.equals(weekDay.name)) return false;

    return true;
}
```

we check if object **references** match

we also check if **types** are the same

we then proceed by checking immutable fields

`String` objects already have `.equals()` implemented; so use that

`hashCode()` **method**

- Is inherited by all objects instances, from java.lang.Object
- Used when you insert an Object into a `HashSet`, `LinkedHashSet`, `HashMap` or `LinkedHashMap` to identify appropriate underlying bucket to store an entry.
- Returns an `int`, representing the `hash code` or `hash` value for the Object for which this method was called upon.

```
Integer i = 7;
Double d = 4.25;
String s = "Pets";

System.out.println(i.hashCode()); // prints 7
System.out.println(d.hashCode()); // prints 1074855936
System.out.println(s.hashCode()); // prints 2484052
```

`hashCode()` **method (2)**

- When inserting an object into a `HashSet`, `LinkedHashSet`, `HashMap` or `LinkedHashMap` you use a key.
- The hash code of this key is calculated, and used to determine where to **store** the object internally (which bucket).
- Later, when you need to lookup an object you also use a key – the same key as before.
- The hash code of this key is calculated and used to determine where to **search** for the object, in the list internal storage.

`hashCode()` **rules**

1. If `object1` and `object2` are equal according to their `equals()` method, they must also have the same hash code.

2. If `object1` and `object2` have the **same hash code**, they **do NOT have to be equal too**.

`hashCode()` **recipe** (Joshua Bloch):

1. Store some constant nonzero value, say *17*, in an `int` variable called `result`.
2. For each significant field in your object (that is, each field taken into account by the `equals()` method), calculate an `int` hash code, "`c`" :
3. For each "`c`", combine the hash code(s) computed above with `result`:

```
result = 31 * result + c;
```

4. Finally, return **result**.
5. Test/Use the resulting hash code in your code

`hashCode()` example

```
@Override
public int hashCode() {
    int result = id;
    result = 31 * result + name.hashCode();

    return result;
}
```

object of type `String` already have `.hashCode()` implemented/overridden; so use that

hashCode() recipe (2)

Field type	Calculation
boolean	**c = (f ? 0 : 1)**
byte, char, short, or **int**	**c = (int)f**
long	**c = (int)(f ^ (f>>>32))**
float	**c = Float.floatToIntBits(f);**
double	**long l = Double.doubleToLongBits(f);** **c = (int)(l ^ (l>>>32))**
Object, where **equals( )** calls **equals( )** for this field	**c = f.hashCode( )**
Array	Apply above rules to each element

lower-case “L”

teamnet transforming technology

Thank you!

Questions or comments on these topics and more, are welcome:

Cristian DUMITRU, Cătălin BÎNĂ, Radu HOAGHE, Andrei MARICA, Oana BEȘLIU, Ioan aka Jonny DINU
@teamnet.ro

We salute you! ☺