

Collections – Part IV

Nico Ludwig (@ersatzteilchen)

TOC

- Collections – Part IV
 - Object-based vs. generic Collections
 - Generics Types in Java
 - Sequential vs. associative Collections
 - Associative Collections
 - Equivalence-based associative collections: Java's TreeMap
 - Operations on Java's Maps
 - Comparison and Implementation of Java's Interfaces Comparable and Comparator
 - The Strategy Pattern
 - Implementation Strategies of associative Collections
 - BST-based Implementation
 - A 20K Miles Perspective on Trees in Computer Science

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!

The Type System in Java 1.4 – Part I

- Explicitly typed – i.e. type names must be explicitly written at their declaration:

```
double sum = 5.2; // Type name "double" must be explicitly written for the variable sum.
```

- Safely typed – i.e. unrelated types can't fool the compiler.

- Cross casting and other "pointer magic" is generally illegal.

- Statically typed – i.e. variables' types are determined at compile time:

```
int count = 42; // count is of type int, we can only perform int operations on it!
```

- Sorry? But the same variable can hold different sub type objects at run time!

- Indeed! We can define variables that may hold e.g. any derived type:

```
Object o = new Car(); // Type Car is derived from type Object.
```

- Java differs the static (*Object*) and the dynamic type (*Car*) of an instance.
- The idea of static/dynamic types is used for run time polymorphism in Java.

The Type System in Java 1.4 – Part II

- Esp. object-based collections rely on the cosmic hierarchy of Java's type system: everything is an *Object*:
 - Java 1.4 collections: Hold items of static type *Object*, their dynamic types can vary.

```
ArrayList lines = new ArrayList(Files.readAllLines(Paths.get("/Library/Logs/Software Update.log"))); // Files.readAllLines() was introduced with Java 1.8!
Object item1 = lines.get(0);
Object item2 = lines.get(1);
// We have to use cast contact lenses to get the dynamic types out of the Objects:
String text1 = (String)item1;
// Access String's interface:
int result = text1.indexOf('g');
// We can also cast directly from the index notation:
String text2 = (String)lines.get(1);
```

- In fact, collections must be able to hold any type in order to be versatile.
 - Java 1.4 "resorted" to run time polymorphism (i.e. Liskov Substitution Principle (LSP)) to get this versatility.
 - So in a Java 1.4 collection, objects of any dynamic type can be stored.
- Object-based collections work great, as long as...
 - we'll only store objects of dynamic types that dependent code awaits,
 - we'll only cast down to dynamic types that dependent code put in,
 - we'll never put mixed unrelated dynamic types into the same collection,
 - well as we don't fall into a type problem during run time.

The Type System in Java 1.4 – Part III

- The problem or lack in the type system using LSP can be presented with the just created collection *lines*:

```
Object item1 = lines.get(0);  
// But, these are not the awaited types, we can't deal with Cars, we await Strings!  
// This cast will fail! An ClassCastException will be thrown!  
Car car = (Car)item1;  
car.startEngine();
```

Car
+ startEngine()

- The problem is that we as programmers have to know about the contained dynamic types (*Strings* and not *Cars* in *lines*).
 - Whenever *Object* is used in an interface (the return value of *lines*' *ArrayList.get()*-method), some convention must be around.
 - This convention describes the dynamic type, we're really dealing with (*Strings* and not *Cars*).
 - The contributed programmers have just to obey the convention... ;-)
- We as programmers have to cast down to the type we await.
 - The cast is needed to access the interface of a static type (e.g. *car.startEngine()*).
 - These casts indicate that the programmer knows more than the compiler!
 - The compiler wears "cast contact lenses"; it can't type check, it trusts the programmer!
 - These casts are type checked at run time, so they are consuming run time!
 - Type errors (an item of *lines* was casted to *Car* instead of *String*) will happen at run time, not at compile time.

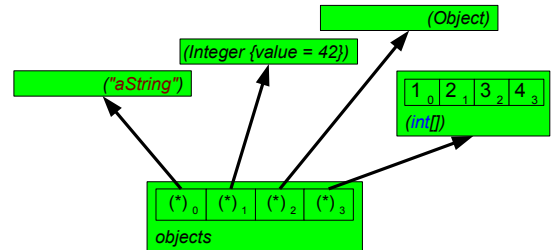
Homogeneous and heterogeneous Collections

- Object-based collections have an interesting feature: they can be heterogeneous collections:

- In a heterogeneous collection every item can have any different dynamic type:

```
ArrayList objects = new ArrayList();
// Since each item is of type object, we can add any object to an ArrayList:
objects.add("aString");           // String
objects.add(42);                  // int
objects.add(new Object());        // Object
objects.add(new int[] {1, 2, 3, 4}); // int[]

for (Object item : objects) {
    System.out.println(item);
}
// >aString
// >42
// >java.lang.Object@27973e9b
// >[I@312b1dae
```



- It works, but heterogeneous collections are tricky: developers must know, on which indexes objects of certain types reside:

```
// This statement will fail, we can't deal with string, on index 1 an int was stored!
// An ClassCastException will be thrown!
String result = (String)objects.get(1);
```

- Heterogenous collection should be avoided! Strive to using homogenous collections or other solutions!

The Type System in Java 1.4 – Some Remedy

- What could we do close the lack in the Java 1.4 type system to stay type safe?
- We could use "strictly" typed arrays as collections if possible, they're no object-based and therefore more type safe:

```
// Create an array that hold Strings:  
String[] lines = Files.readAllLines(Paths.get("/Library/Logs/Software Update.log")).toArray(String[]::new);  
// With arrays we don't need casts, instead we can directly access lines' items as String objects:  
String text1 = lines[0];  
// Access String's interface:  
int result = text1.indexOf('g');
```

- However, arrays cannot be used, when the collection in question needs to be modified (enlarged or downsized) after creation.
- But in the world of collections, there is a much better way to work in a type safe manner: generic collections!

8

- We could create own, type safe not object-based collections as UDTs (e.g. a special String-list for the discussed example).
- In .NET we could resort to so called specialized collections in the namespace *System.Collections.Specialized*, e.g. *StringCollection*.

```
// Create a specialized StringCollection:  
StringCollection lines = new StringCollection();  
lines.AddRange(File.ReadAllLines("/Library/Logs/Software Update.log"));  
// With the specialized StringCollection we don't need casts, instead we  
// can directly access lines' items as string objects:  
string text1 = lines[0];  
// Access string's interface:  
int result = text1.IndexOf('g');
```


(Java 1.5) Generics to the Rescue

- Let's improve our problematic code with the generic collection `ArrayList<T>` (`java.util.ArrayList<T>`):

```
// The generic type ArrayList<T>:  
public class ArrayList<T> { // (members hidden)  
    public void add(T item) {  
        // pass  
    }  
}
```

```
// Create a generic ArrayList that holds Strings:  
ArrayList<String> lines = new ArrayList<String>(Files.readAllLines(Paths.get("/Library/Logs/Software Update.log")));  
// This time we don't need casts, instead we can directly access lines' items as String objects:  
String text1 = lines.get(0);  
// Access String's interface:  
int result = text1.indexOf('g');
```

- Java 1.5 introduced generic types to get rid of the presented type problems.

- Generic types are types that leave a part of their type information open.
- Generics are an outstanding feature in Java 1.5.

```
// This time we'll get a compile time error:  
Car car1 = lines.get(1);  
// The type argument in lines was string, not Car!  
// Cannot convert string to Car.  
car1.startEngine();
```

- Generics (here `ArrayList<T>`) leave type information open? How should this help me out?

- The open type information is accessible by type-parameters (here `T`).
- As programmers we can fill the type-parameters with concrete type-arguments (here `String`).
- By setting type-arguments of a generic type we create another type, the "constructed" type (`ArrayList<String>` in this case).
 - The full name of the constructed type is `ArrayList<String>!`
- => Net effect: The type-argument (`String`) will be of static type, so the formerly open type information can be checked by the compiler.
- In the end type safety means that types are checked by the compiler at compile time and no longer at run time.

Bottom line:

Generics provide another way to create code, that works with multiple types. Other ways: oo-polymorphism and dynamic typing.

Generic Collections – Overview – Part I

- Java's formally *Object*-based collections have just been generified in Java.
 - I.e. where we formally found the type *Object* in collections' interfaces, we now find type-parameters.
 - This was done to get maximal compatibility with old code and at run time and enable painless "generic upgrade paths".

- Thus, the generic "counterpart" of the collection *ArrayList* is *ArrayList<T>*, *T* is the type-parameter.
 - *ArrayList<T>*'s interface is set to this type-parameter in parameter- and return-types of methods.

```
// The generic type ArrayList<T>:  
public class ArrayList<T> { // (members hidden)  
    public void add(T item) { /* pass */ }  
    public T get(int index) { /* pass */ }  
}
```

- The type-parameter *T* acts as a placeholder for the actual type-argument.
- Also Java interfaces, which abstract over collections have been generified: i.e. *ArrayList<T>* implements *List<T>*.
 - Hence, instead of this

```
ArrayList<String> objects = new ArrayList<String>();
```

- we can safely write this to enforce the DIP also with generic types:

```
List<String> objects = new ArrayList<String>();
```

10

- In our discussion of generics we will ignore the fact, that when methods are called on an open type, the types offering these methods must be set as bounds (Java) or constraints (.NET) on the open type in the definition of the generic type.
 - This indeed constrains the usage of generic types somewhat, we can, e.g. not call any operators on the open type, because operators cannot be overloaded in Java and operators are static and cannot be part of an interface in .NET.
 - However, it should be said, that bounds or constraints are usually not a problem, when generic collections are used. – This is because many collections never call any methods on its elements, so the generic collection types are not required to set any bounds/constraints. Without bounds/constraints, the Java/C# compiler and the JVM/.NET runtime consider the open type is *Object/object*.
 - Even collection types, which do call methods on the open type do usually not set any constraints. E.g. *SortedSet*, which must compare the open type, does not constrain the open type to *Comparable<T>*, but creates a suitable non-generic *Comparer* instance at run time. It is done this way to allow a broader amount of types to be usable in *SortedSet*. The constraint on *Comparable<T>* would only make types available in .NET 2.0 and after usable in *SortedSet*.

Generic Collections – Overview – Part II

- When generics are used in code, the need for the specification of type-arguments can quickly prolong the code:

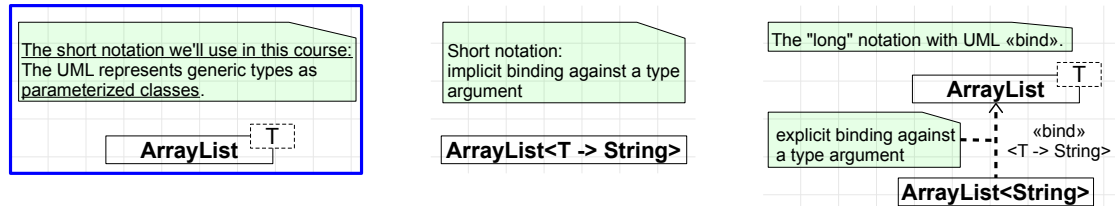
```
ArrayList<StringBuilder> stringBuilders = new ArrayList<StringBuilder>();
```

- Java doesn't provide many simplifications here, but we can leave the type-argument away, when a generic object is instantiated:

```
ArrayList<StringBuilder> stringBuilders = new ArrayList<>();
```

- The elided type-argument is inferred by the compiler by analyzing the expression (in this case simply the left side from the assignment)
- When the type-argument is removed, the remaining syntax "<>" resembles a diamond, therefore it is called "diamond-operator".
 - ... but it is not an operator, it's just a syntactic idiom.

- The UML provides some notations to put generics into effect:



Slightly advanced Generics

- It is also possible to define generic types, whose type argument is another generic type.
 - E.g. we can define a list of lists, i.e. a collection of collections, which is effectively a two-dimensional array or matrix.

```
// A jagged array in Java as matrix:  
int[][] matrix = new int[3][];  
matrix[0] = new int[] {1, 2, 3, 4};  
matrix[1] = new int[] {6, 7};  
matrix[2] = new int[] {8, 9, 0};  
System.out.printf("Item: %d%n", matrix[1][0]);  
// >Item: 6
```

```
// A list of lists of ints as matrix:  
List<List<Integer>> matrix = new ArrayList<>();  
matrix.add(List.of(1, 2, 3, 4));  
matrix.add(List.of(6, 7));  
matrix.add(List.of(8, 9, 0));  
System.out.printf("Item: %d%n", matrix.get(1).get(0));  
// >Item: 6
```

- Such collections are better than multidimensional arrays in some aspects:
 - They offer the same concept to express rectangular as well as jagged arrays.
 - "Real" collections are very flexible, as elements can be added and removed (or: the matrix' dimensions can be enlarged and shrunk).
- Java also supports generic types having two or more type parameters:
 - E.g. with the type `TreeMap<K, V>` (`java.util.TreeMap<K, V>`) (this type will be discussed later):

```
// The generic type TreeMap<K, V>:  
public class TreeMap<K, V> { // (details hidden)  
    // pass  
}
```

```
// Creating a TreeMap object:  
TreeMap<String, Integer> aTreeMap = new TreeMap<>();  
aTreeMap.put("fortytwo", 42);
```

- To understand collections in Java, C++ and .NET it is required also to understand advanced generics.

The better Type System – from object-based to generic Collections

- Hence, we'll use generic types whenever appropriate. We'll define following substitutes for already introduced collections:

- *Collection* → *Collection<T>*
- *List* → *List<T>*
- *ArrayList* → *ArrayList<T>*
- *Iterable* → *Iterable<T>*
- *Iterator* → *Iterator<T>*

- From object-based to generic collections:

```
List names = new ArrayList();
```

```
names.add("James");
```

```
names.add("Miranda");
```

```
// Get the values back (object-based collection: static type Object):
```

```
Object name1 = names.get(0); // "James"
```

```
Object name2 = names.get(1); // "Miranda"
```

```
String name1AsString = (String)name1; // Cast the string out of the object.
```

```
String name2As = (String)names.get(1); // Cast directly from the index notation.
```

```
names.set(1, "Meredith");
```

```
System.out.println(names.get(1));
```

```
// >Meredith
```

```
List<String> names = new ArrayList<>();
```

```
names.add("James");
```

```
names.add("Miranda");
```

```
// Get the values back (generic string-collection: static type string):
```

```
String name1 = names.get(0); // "James"
```

```
String name2 = names.get(1); // "Miranda"
```

```
names.set(1, "Meredith");
```

```
System.out.println(names.get(1));
```

```
// >Meredith
```

13

Excursus: The Containers' Type System of the C++ STL

- C++ [templates](#) are, at least principally, C++' equivalent to .NET's and Java's generics.
 - C++ provides the [template](#) `std::vector<T>` ([<vector>](#)), which is functionally equivalent to Java's `ArrayList<T>` and .NET's `List<T>`:

```
// <vector>
namespace std {
    template <typename T> class vector { // (details hidden)
    public:
        void push_back(const T& newItem);
        T& operator[](size_type n);
    };
}
```

```
// names is a vector containing string-elements.
std::vector<std::string> names;
```

```
names.push_back("James");
names.push_back("Miranda");
```

```
// Get the values back (template string-container: static type std::string):
std::string name1 = names[0]; // "James"
std::string name2 = names[1]; // "Miranda"
```

```
names[1] = "Meredith";
std::cout<<names[1]<<std::endl;
// >Meredith
```

- Before C++ had [templates](#), `void*`-based collections have been used, incl. cast contact lenses. – This is still true for C.
 - `void*`-based collections are the functional equivalent to object-based collections!
- Objective-C and many scripting languages use so called dynamic typing instead of generic or object-based structures.
 - (We're not going to discuss these concepts in this course.)

14

- [templates](#) are very powerful, they are even Turing-complete as a programming concept in itself.
- In opposite to what is currently understood as generic type, C++' [template](#) parameters do not need to implement a specific interface, i.e. no constraints or bounds as in .NET's generics and Java's generics must be regarded. – Instead, the compiler checks the compatibility of calls more flexible.
- For every usage of e.g. a [templated classes object](#) as an instance, the C++ compiler creates a new class in itself. – We say the templated [class](#) is instantiated. So there is no type erasure going on. The mechanism is even more clever, the C++ compiler does only create code for calls, which are effectively performed in code that instantiates [templated classes](#).
 - Benefit: the (unconstrained) [templated](#) code allows calling any kind of [operator](#), because the code is only instantiated and checked when used.
 - Downsides:
 - (1) Because [templates](#) are instantiated at compile time, two DLLs having the same [template](#) instance, e.g. `std::list<int>`, would have two individual instances on their own, w/o a way to share those at run time.
 - (2) Because of (1), the same [template](#) instances in two individual DLLs are incompatible. [template](#) instances cannot be exported from DLLs.
 - (3) [template](#) definitions alone cannot reside in binary code. They can only exist in source code, esp. in h-files, because it is a pure compiler feature.

Excursus: Generics Collections/Generics in .NET/C#

- .NET's generic collections are completely different from .NET's **object**-based collections.
 - Generic collections are incompatible to **object**-based collections in general, but each "old" collection has a generic counterpart.
 - The generic collections live in the **namespace** `System.Collections.Generic`.

- The generic counterpart of the **object**-based collection `ArrayList` is `List<T>`, `T` is the type-parameter.

- `List<T>`'s interface is set to the open type `T` in **parameter**- and **return**-types of methods.

```
// The generic type List<T>:  
public class List<T> { // (details hidden)  
    public void Add(T item) { /* pass */ }  
    public List<T> GetRange(int index, int count) { /* pass */ }  
}
```

- The type-parameter `T` acts as a placeholder for the actual type-argument.

- When a type is constructed, `T` is replaced by a type argument (the result is the constructed type).

- `List<string>`'s interface is set to the static type `string` in **parameter**- and **return**- types of methods.

```
// Create a generic List that holds strings:  
List<string> lines;
```

```
// The constructed type List<string> has following interface:  
public class List<string> { // (details hidden)  
    public void Add(string item) { /* pass */ }  
    public List<string> GetRange(int index, int count) { /* pass */ }  
}
```

- The idea of generics is to use types (i.e. not values) as arguments!

- The type safety we got is that **generics move type run time errors to compile time errors**.

15

The Collection Type System on the Java Platform – Part I

- The story about generic collections in Java is somewhat advanced, but Java programmers should know these facts.
 - A good way to understand Java's generics (and the problems that come with it) is to compare them to, e.g., .NET's generics.

- Here we have basically the same generic **class** in Java and .NET/C#:

```
// Java
public class MyList<T> {
    public List<T> items = new ArrayList<T>(5);
}
```

```
// C#
public class MyList<T> {
    public IList<T> items = new List<T>(5);
}
```

- The definition of instances of these **classes** has almost the same syntax in Java and C#:

- In Java we can only use reference types as type arguments for generic types. E.g. we can not create *MyList<int>* or *MyList<double>*.

```
MyList<String> stringList = new MyList<>();
MyList<Integer> intList = new MyList<>();
```

```
MyList<string> stringList = new MyList<string>();
MyList<int> intList = new MyList<int>();
```

- As we know the idea behind generics is to have **type safety**. So the last statements won't compile respectively:

```
// Java
stringList.items.add("Frank"); // OK for the compiler! A String literal can be set, because items is a List<String>.
stringList.items.add(42); // Invalid for the compiler! Generics are type safe! An int literal can't be set, because items is not a List<Integer>.
```

```
// C#
stringList.items.Add("Frank"); // OK for the compiler! See explanation for Java above.
stringList.items.Add(42); // Invalid for the compiler! See explanation for Java above.
```

16

The Collection Type System on the Java Platform – Part II

- The difference between Java and .NET is that different things are generated at compile time.

- A compiler creating .NET IL code will produce a so called constructed type for each "filled out" generic type in the code:

```
// C#
MyList<string> stringList = new MyList<string>();
MyList<int> intList = new MyList<int>();
```

type construction of a .NET IL compiler

```
public class MyList<string> { // Constructed type
    public IList<string> items = new List<string>(5);
}
```

```
public class MyList<int> { // Constructed type
    public IList<int> items = new List<int>(5);
}
```

- A compiler creating Java bytecode will just erase the type argument and will fall back to an object-based type:

- This compilation step is called type erasure. The resulting type (w/o generic type parameters) is called the raw type of the generic type.

```
// Java
MyList<String> stringList = new MyList<>();
MyList<Integer> intList = new MyList<>();
```

type erasure of a Java bytecode compiler

```
public class MyList { // Raw type
    public List items = new ArrayList(5);
}
```

- Often this isn't a problem in Java, because compile time type safety is fine, but it hurts the run time type safety!

```
stringList.items.add("Frank"); // Ok as before!
((MyList)stringList).items.add(42); // Huh? Casting (cast contact lenses) to the raw type and adding an int: ok for compiler and at run time!
String itemAt1 = stringList.items.get(1); // Bang! Getting the int at the index one via the generic type: ok for the compiler but crashes at run time!
// java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
```

- Ouch! But Java provides a means to reestablish run time type safety: run time type checked collection wrappers. 17

- Java doesn't allow value types as type arguments. If this was allowed, a lot of boxing would go on, when the type-erased type is used, because everything in its interface is *Object*-based. – This is where .NET, w/o type erasure, but constructed types does shine.
- So, .NET's constructed types of a specific open type are all equivalent for different reference types, only the interfaces of methods are different according to the actual type argument. But the constructed types for value types as type arguments differ in their interfaces, as well as in their code, because some operation, e.g. assigning, work different for a value type (assembly instructions to copy objects on the stack with the exact memory layout), than for a reference type (just move references around).

The Collection Type System on the Java Platform – Part III

- The companion [class](#) `java.util.Collections` provides **static** methods (simple factories) to create run time type checked collections.
 - A present collection needs to be passed to `Collections.checked...()` and a new collection wrapping the passed one will be returned.
 - Let's wrap `MyList`'s encapsulated `List` to be a checked `List` with `Collections.checkedList()`:

```
stringList.items = Collections.checkedList(stringList.items, String.class); // Wrap items, so that it can only deal with Strings at compile time.
stringList.items.add("Frank"); // Ok as before! It was already checked by the compiler.
((MyList)stringList).items.add(42); // Bang! An int can't be added to a list that is only allowed to contain Strings!
String itemAt1 = stringList.items.get(1); // Will not be reached at all!
```

- More simple factories can be found in the [class](#) `Collections` as **static** methods (`checkedSet()`, `checkedMap()` etc.).
 - The return type of all of these simple factories are just [interface](#) types, the type checked implementations are always hidden.
 - So the UML diagram of the relations between the contributing types looks like so:



- Run time type checked wrappers were added to the JDK in order have compatibility w/ non-generic collections, retaining compile time type safety of generics and having run time type safety.
 - There is an initiative in the Java community to introduce generic types with constructed types into Java: reified generics.
 - Mind that run time type checking is costly!

18

- To be fair it must be said, that Java uses type erasure instead of reified generics for a reason: This way type-erased generic types stay bytecode-compatible to pre-Java 5 code, esp. collections.

Indexed, sequential and associative Collections

- Up to now, we discussed relatively simple indexed and sequential collections.
- Besides indexed and sequential collections there also exist so called associative collections.
- In opposite to indexed and sequential collections, associative collections inspect the stored objects.
 - E.g. the indexed collection *ArrayList* just stores references to objects, but it does never, e.g., call methods on these objects.
 - For indexed and sequential collections the contained items are said to be transparent.
 - Associative collections have to inspect stored objects for equivalence or equality in order to provide very mighty features.
- Let's start by analyzing a problem, which can be solved with associative collections.

19

- It should be mentioned that some methods of list-types already needed to inspect the contained items to function, e.g. *contains()*, *remove()*, *indexOf()*. These methods use equality to inspect/compare values.

Example – Color Names to HTML Color Codes – Part I

- Let's assume that we have to handle color names and figure out the belonging to HTML color codes:

```
// Java
// Here we have two Lists that hold color names and HTML color codes:
List<String> colorNames = new ArrayList<>(List.of("Blue", "Red", "Green"));
List<Integer> htmlColorCodes = new ArrayList<>(List.of(0x0000FF, 0xFF0000, 0x00FF00));

// With the information given, the task to pick an HTML color code from a color name is trivial: Because the
// index of the color name matches the index of the color HTML code, we can formulate a simple algorithm:
int indexOfRed = colorNames.indexOf("Red");
int colorCodeOfRed = htmlColorCodes.get(indexOfRed);
System.out.printf("Color code for Red: %#08x\n", colorCodeOfRed);
// >Color code for Red: 0xff0000
```

- Of course the task can be solved this way, but the solution is based on a possibly fragile assumption:

- The two separate Lists need to hold belonging to values at the same indexes!
- Basically we've to go through all items sequentially and compare (*List.indexOf()*).
- We say the association is index-based.

Blue	Red	Green	colorNames
0	1	2	matching indexes
0x0000FF	0xFF0000	0x00FF00	htmlColorCodes

- The presented algorithm is a so called lookup-algorithm (lookup), it is based on *List.indexOf()* in this case.
 - Lookups are used very often in programming.
 - A problem of lookups is, that separate collections (*colorNames* and *htmlColorCode*) need to be evaluated (like tables in relational databases).
 - Modern collection frameworks provide dedicated collections to cover lookups in a comfortable manner: associative collections.

- There is also another technical limitation using an int-index to associate elements means we can have max. max(int) associated elements.

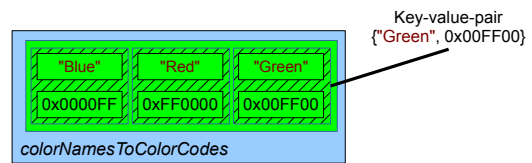
Example – Color Names to HTML Color Codes – Part II

- Associative collections abstract the usage of multiple collections to solve the just presented lookup in an intuitive manner:

- Here, Java's associative collection `TreeMap<String, Integer>` is shown (it implements `java.util.Map<String, Integer>`):

```
// Associating color names with HTML color codes using a TreeMap:
Map<String, Integer> colorNamesToColorCodes = new TreeMap<>();
colorNamesToColorCodes.put("Blue", 0x0000FF);
colorNamesToColorCodes.put("Red", 0xFF0000);
colorNamesToColorCodes.put("Green", 0x00FF00);

int colorCodeOfRed = colorNamesToColorCodes.get("Red");
System.out.printf("Color code for Red: %#08x%n", colorCodeOfRed);
// >Color code for Red: 0xff0000
```



- The association of both, color names and color codes is available via the single collection `colorNamesToHtmlColorCodes`.

- We have to rethink our termination: associative collections associate a key (color name) to a value (color code).

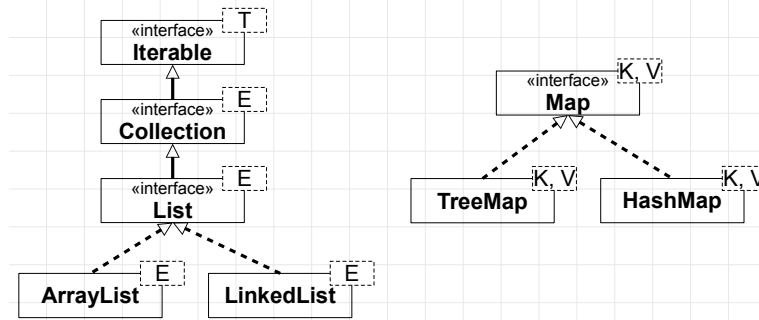
- The association is key-based and no longer index-based.
 - The internal "organization" is key-value-pair-based.
 - In a sense an array is nothing but an associative collection that associates a value with an index (mind the `[]`-syntax).
 - In fact real problems often involve associating values with other values, e.g.: White pages, thesaurus, document index.

21

- Theoretically, the technical limitation using an `int`-index to associate elements can be broken with "real" associative collections.
- Mind how the term key (primary/foreign key) is also applied in the design of databases.
- The .NET collection `KeyedCollection<K, T>` allows lookups of data stored in the value (i.e. there are no key-value-pairs). Indeed `KeyedCollection` is a very useful collection. However, it is not a classical associative collection, because it uses linear search (it is backed by a `List` and calls `Contains()` on that `List` potentially for each item). The `abstract class` `KeyedCollection` needs to be specialized (overriding `GetKeyForItem()`, i.e. we have the template method design pattern here) to get the key from the managed item/value.

The Type-Dichotomy of Collection and Map in Java – Part I

- When we discussed Java's [interface](#)-based approach of its collection framework, we identified the [interface](#) *Collection*.
 - Being an [interface](#), *Collection*'s idea is that some operations are common across all collection types.
- But there is a problem with Java: *Maps* are not derived from *Collection*!
 - Instead *Collection* and *Map* make up a type-dichotomy: they are sibling types, which create two distinct type-hierarchies in Java.



- From the view of the type-hierarchy, *Collection* and *Map* have nothing in common, not even being *Iterable*!
 - And this is bad in a sense: We can't use *Collections* and *Maps* in a common way although they both are "data-containers".

22

The Type-Dichotomy of Collection and Map in Java – Part II

- The type dichotomy of *Collection* and *Map* was esp. a problem of Java versions before Java 8 e.g. concerning iterability:

```
Map<String, Integer> colorNamesToColorCodes = getColorCodes();
for (Object item : colorNamesToColorCodes) { // Invalid!
    System.out.println(item);
}
```

```
List<String> colorNames = getColorNames();
for (Object item : colorNames) {
    System.out.println(item);
}
```

- Beginning with Java 8, the interfaces *Iterable* and *Map* got the method *forEach()*, which remedies this lack somewhat:

```
colorNamesToColorCodes.forEach((key, value) -> {
    System.out.printf("key: %s, value: %s%n", key, value);
});
```

```
colorNames.forEach( item -> {
    System.out.println(item);
});
```

- Map.forEach()* was set in place with Java 8, because it added support for lambda expressions which makes the syntax lightweight enough resembling *for(;;)*.
- Map.forEach()* even deconstructs the managed pairs into key and value respectively.

- Effectively, Java 8's *Stream* API does fully abstract data processing from the underlying "data-container":

```
colorNamesToColorCodes
    .entrySet()
    .stream()
    .forEach( item -> {
        System.out.println(item);
    });
```

```
colorNames
    .stream()
    .forEach( item -> {
        System.out.println(item);
    });
```

- As can be seen, when we get a *Stream* from the data-container in question further data processing looks equivalent.
- The only quirk with *Map* is, that we have to get the *Stream* from *Map.entrySet()*, but the remaining code is equivalent.

A practical Application of associative Collections: Histograms

- A pretty common task in data processing is to count occurrences of individual data. The result is a so called histogram.
 - E.g. assume we have a file with a list of forenames of new born babies across the UK last year and we should count the names.
 - The following code shows, how an associative collection (*TreeMap*<K, V>) can be used to collect the data for the histogram.
 - The resulting data, which associates data-items to the count of their occurrences for the histogram is sometimes called spectrum:

```
String namesFile = Files.readString(Path.of("/Users/nico/Documents/names.csv"));
String[] names = namesFile.split("\\W?\\.\\W?");
Map<String, Integer> namesSpectrum = new TreeMap<>();
for (String name : names) {
    if (namesSpectrum.containsKey(name)) {
        namesSpectrum.put(name, namesSpectrum.get(name) + 1);
    } else {
        namesSpectrum.put(name, 1);
    }
}
```

```
<names.csv>
Liam, Olivia, Noah, Emma, Oliver, Ava, William,
Sophia, Elijah, Isabella, James, Charlotte, Benjamin,
Amelia, Lucas, Mia, Mason, Harper, Ethan, Evelyn,
Benjamin, Amelia, Lucas, James, James, Arthur, Olivia,
Noah ...
```

- We exploit the idea of an associative collection, that we can associate the name with the count of its occurrences!
 - We check if a name is contained in the spectrum: if so we increment the occurrences, if not we put the name with occurrence 1.

```
System.out.println("The histogram of names:");
namesSpectrum.forEach( (key, value) -> {
    System.out.printf("Name: %s, Occurrences: %s%n", key, value);
});
```

```
Terminal
The histogram of names:
Name: Evelyn, Occurrences: 2
Name: Charlotte, Occurrences: 1
Name: James, Occurrences: 3
Name: Mia, Occurrences: 1
Name: Ethan, Occurrences: 5
...
```

24

- Sure, spectrums can be created simpler (e.g. via *Streams* in Java), but the idea is to show how it can be done using associative collections, which is already a pretty simple solution.

Organization of Items in associative Collections

- New about associative collections is that associative collections inspect the items they hold.
 - I.e. associative collections operate on contained items: they call methods on the keys of the contained key-value-pairs.
 - Associative collections need to compare the keys they hold.
 - In opposite to other collections, items are got and set by analyzing their relationships.
- Associative collections are very powerful and allow writing elegant/readable algorithms to solve complicated problems.
 - Associative collections could be implemented with two arrays as shown before, but this would be inefficient for common cases.
 - A clever internal organization of associative collections is required to allow efficient access and modification of contained items.
- There are two general ways to organize items (i.e. their keys) in an associative collection:
 - (1) By equivalence. I.e. by their relative order/order-relationship, such as less than and greater than.
 - (2) By equality. I.e. by hash-codes and the result of the equality check, e.g. with the methods `hashCode()/equals()` in Java.
- In this and the next lecture we're going to discuss associative collections that organize their keys by equivalence.
 - This is the case for *TreeMap*, so we'll discuss Java's *TreeMap*.

25

- It should be mentioned that some methods of list-types already needed to inspect the contained items to function, e.g. *contains()*, *remove()*, *indexOf()*. These methods use equality to inspect/compare values.

Operations on Maps – Part I

- Create the empty `TreeMap<String, Integer> colorNamesToColorCodes`:

```
Map<String, Car> ownersToCars = new TreeMap<>();
```

- Then we can add some key-value-pairs like so:

```
ownersToCars.put("Bud", new Car("KL-GS 23"));
ownersToCars.put("Gil", new Car("SB-DF 522"));
ownersToCars.put("James", new Car("WND-KR 30"));
ownersToCars.put("Will", new Car("KL-VV 98"));
ownersToCars.put("Helen", new Car("WND-SG 76"));
ownersToCars.put("Clark", new Car("KIB-TI 52"));
ownersToCars.put("Ed", new Car("KIB-MJ 49"));
```

Car
- licencePlateID : String
+ getLicencePlateID() : String
+ setLicencePlateID(licencePlateID : String)

- Adding a pair for an already existing key will overwrite the value associated with the existing key with the new value.

- Java's *Maps* don't allow having a key with multiple values. (However, the C++ STL-container `std::multimap` (<map>) can handle this.)

```
Car edsFormerCar = ownersToCars.put("Ed", new Car("HH-FM 17"));
```

- `Map.put()` returns the old value of the key. If the put key-value pair was not in the *Map* before, `null` will be returned.

- There is a problem: what if there already is a key, but its value is `null`? Then `Map.put()` will also return `null`!

- The return value of `Map.put()` doesn't tell us, if a key-value pair was really brand new in the *Map*, or if the former value was just `null`.

- To be sure, if a key is already contained in a *Map*, we can use the method `Map.containsKey()`:

```
boolean pamelAlreadyPresent = ownersToCars.containsKey("Pamela");
// pamelAlreadyPresent = true
```

26

Operations on Maps – Part II

- The next important method is `Map.get()`, it performs the lookup and returns the value associated with the key:

```
Car willsCar = ownersToCars.get("Will");  
// willsCar = Car{LicencePlateID = "KL-VV 98"}
```

- Querying the value for a key that doesn't exist via `Map.get()` will also return `null`:

```
Car pamelasCar = ownersToCars.get("Pamela"); // Returns null: a key-value-pair with that key doesn't exist in ownersToCars.
```

- The situation is the same as with `Map.put()`: we cannot distinguish, if an associated value was `null`, or if the key didn't exist
- To avoid such surprises we should check the key for existence (with `Map.containsKey()`), before looking it up.

```
if (ownersToCars.containsKey("Pamela")) {  
    Car pamelasCar = ownersToCars.get("Pamela");  
}
```

- Of course we can also remove keys from the `Map` via `Map.remove()`:

```
Car gilsFormerCar = ownersToCars.remove("Gil");  
// willsCar = Car{LicencePlateID = "KL-VV 98"}
```

- Again, if the key for which the entry should be removed is not present in the `Map`, `Map.remove()` will return `null`.

- We can get the current count of elements stored in the `Map` via `Map.size()`:

```
System.out.println(ownersToCars.size());  
// >4
```

Operations on Maps – Part III

- When we discussed *Map*-iteration, we came across the fact, that *Map.forEach()* allows iterating over the key-value pairs:

```
// Iteration via Map.forEach():
ownersToCars.forEach((name, car) -> {
    System.out.printf("Car of %s: %s%n", name, car.getLicencePlateID());
});
```

- We also had a glimpse over *Map.entrySet()*, which is able to provide the key-value pairs as entry objects:

```
ownersToCars
    .entrySet()
    .stream()
    .forEach( entry -> {
        System.out.printf("Car of %s: %s%n", entry.getKey(), entry.getValue().getLicencePlateID());
    });
```

- The value returned from *Map.entrySet()* is an interesting object: it represents the entries as object of type *Set*.

- Actually, *Set* is a *Collection*! We know, that *Collections* are *Iterable*, so they can be used with *for(*:

```
for (Map.Entry<String, Car> entry : ownersToCars.entrySet()) {
    System.out.printf("Car of %s: %s%n", entry.getKey(), entry.getValue().getLicencePlateID());
}
```

- For now only the type *Map.Entry<K, V>*, more specifically *Map.Entry<String, Car>* is interesting for us, we'll discuss it on the next slides.
- In a future lecture, we will discuss *Set*.

Map-Entries – Part I

- The type `Map.Entry<K, V>` is a static nested interface in the interface `Map`:

```
// Details hidden
public interface Map<K, V> {
    interface Entry<K, V> {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

- The bare `interface` allows to get and set the value, but the key can only be retrieved and not be set.
- => The key of `Map.Entry<K, V>` is meant to be immutable, but the value can be changed.

- Java offers two public implementations of `Map.Entry<K, V>`.

- `AbstractMap.SimpleEntry<>`, a static nested class in the class `AbstractMap<K, V>` (which is the super class of `Map<K, V>`):

```
Map.Entry<String, Car> entry = new AbstractMap.SimpleEntry<>("Michael", new Car("SB-GF 652"));
entry.setValue(new Car("KIB-TI 52"));
```

- `AbstractMap.SimpleImmutableEntry<>` is an implementation, which disallows modifying the value after creation:

```
Map.Entry<String, Car> entry1 = new AbstractMap.SimpleImmutableEntry<>("Michael", new Car("SB-GF 652"));
entry1.setValue(new Car("KIB-TI 52")); // Invalid! Throws UnsupportedOperationException
```

- There is yet another implementation of `Map.Entry<K, V>`, which can be instantiated using the simple factory `Map.entry()`.

- `Map.entry()` accepts two non-null arguments and wraps them into a `Map.Entry<K, V>` disallowing modification of the value after creation:

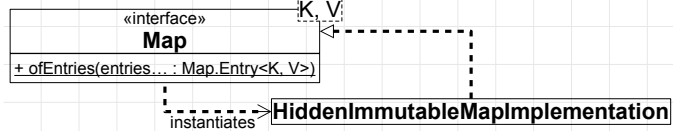
```
Map.Entry<String, Car> entry2 = Map.entry("Michael", new Car("SB-GF 652"));
entry2.setValue(new Car("KIB-TI 52")); // Invalid! Throws UnsupportedOperationException
```

29

Map-Entries – Part II

- *Map.Entry*<K, V> per se is a useful type, because often we need pairs of objects, or 2-tuples.
- Back to *Maps*, we can use another simple factory, namely *Map.ofEntries()* creating an immutable *Map* of *Map.Entry*<K, V>s:

```
Map<String, Car> ownersToCars = Map.ofEntries(
    Map.entry("Bud", new Car("KL-GS 23")),
    Map.entry("Gil", new Car("SB-DF 522")),
    Map.entry("James", new Car("WND-KR 30")),
    Map.entry("Will", new Car("KL-VV 98")),
    Map.entry("Helen", new Car("WND-SG 76")),
    Map.entry("Clark", new Car("KIB-TI 52")),
    Map.entry("Ed", new Car("KIB-MJ 49")));
```



- It is not specified, which implementation of *Map* is returned by *Map.ofEntries()*, however it doesn't really matter it is "some" *Map*.
- Notice, that the returned *Map* is immutable, calling methods like *put()* or *remove()* will throw *UnsupportedOperationException*:

```
Car edsCar = ownersToCars.get("Ed");
ownersToCars.put("Ed", new Car("KA-TF 24")); // Invalid! Throws UnsupportedOperationException
```

- Additionally, *Map* provides the simple factory *Map.of()*, which creates an immutable *Map* from an even count of arguments:
 - *Map.of()* has 11 overloads to accept up to 10 key-value pairs.
 - The arguments of *Map.of()* must not be null!

```
Map<String, Car> ownersToCars = Map.of(
    "Bud", new Car("KL-GS 23"),
    "Gil", new Car("SB-DF 522"),
    "James", new Car("WND-KR 30"),
    "Will", new Car("KL-VV 98"),
    "Helen", new Car("WND-SG 76"),
    "Clark", new Car("KIB-TI 52"),
    "Ed", new Car("KIB-MJ 49"));
```

```
Map<String, Car> emptyMap = Map.of();
```

30

- Mind how the pattern of *Map*, the simple factories *Map.ofEntries()*/*Map.of()* and the hidden implementation of immutable *Maps* is similar to the simple factories in *Collections* (e.g. *Collections.checkedList()*), collection-interfaces as return types and hidden implementations.
- *Map.ofEntries()* and *Map.of()* can also be called without any arguments, then an empty *Map* is created respectively.

Associative Collections in other Languages

- In many scripting languages associative collections are first class citizens and have special syntactical support.

- The names "hash", "hashmap" or "associative array" are common in scripting languages.

```
# E.g. in Ruby associative collections are supported as "hashes" with integrated syntactical support.
# Just initialize a variable with a list of key => value pairs:
colorNamesToColorCodes = {"Blue" => 0x0000FF, "Red" => 0xFF0000, "Green" => 0x00FF00}
# The items can be accessed with the []-operator:
printf("Color code for Red: %06X", colorNamesToColorCodes["Red"])
# >Color code for Red: FF0000
```

- In JavaScript every object is a collection of property-value-pairs per se. An array is also an associative collection:

```
// JavaScript
// Create associative collection as object with properties:
var colorNamesToColorCodes = new Object();
colorNamesToColorCodes.Blue = 0x0000FF;
colorNamesToColorCodes.Red = 0xFF0000;
colorNamesToColorCodes.Green = 0x00FF00;
console.log("Color code for Red: "
+colorNamesToColorCodes.Red.toString(16));
// >Color code for Red: FF0000
```

```
// JavaScript
// Create associative collection as associative array:
var colorNamesToColorCodes = new Object();
colorNamesToColorCodes["Blue"] = 0x0000FF;
colorNamesToColorCodes["Red"] = 0xFF0000;
colorNamesToColorCodes["Green"] = 0x00FF00;
console.log("Color code for Red: "
+colorNamesToColorCodes["Red"].toString(16));
// >Color code for Red: FF0000
```



- On other platforms associative collections are often called Maps (Java) or Dictionaries (Cocoa, .NET).
 - Esp. the term "Map" is taken from the phrase "to map something", which means "to associate something".
 - The term "hash" still needs to be clarified in this context: we have to understand how associative collections work.
 - We'll start discussing equivalence-based associative collections and then equality-based associative collections using hashing.

TreeMap and Equivalence Comparison

- Now its time to understand how *TreeMaps* work basically, we'll discuss this in depth in the next lecture.
 - Nevertheless we will discuss some mechanics right now.
 - We have to clarify, how *TreeMap* knows if a key is already present or not! – It needs to compare keys somehow!
- The question is: "How does *TreeMap* know if two keys are equal?"
 - A part of the answer is that *TreeMap* doesn't know, if two keys are equal, but it knows if two keys are equivalent!
 - *TreeMap*'s comparison is based on the order-relationship among items. This is called equivalence comparison.
- The order-relationship can be expressed by evaluating two values one being less than, greater than or equal the other.
 - If, of two values, one is neither less than nor greater than the other the values are said to be equivalent. (Not equal!)
 - Java provides a way for types to declare equivalence comparability: implementing the interface *java.lang.Comparable<T>*.
 - (The C++ STL requires types to override operator< to implement equivalence comparability.)
- Now we are going to understand how *Comparable<T>* has to be implemented for own UDTs.

UDTs as Keys of associative Collections – Part I

- Up to now we've only used objects of builtin types for the keys of associative collections (e.g. *String*).
 - Now we are going to explore how to use keys, that are instances of our own types. Let's do this with the UDT *Car*:

```
Map<Car, String> carToManufacturer = new TreeMap<>();  
// Adding a Car as key won't work:  
Car car1 = new Car();  
car1.setLicencePlateID("KL-SD 14");  
carToManufacturer.put(car1, "Ford"); // Invalid! Exception in thread "main"  
// ClassCastException: Car cannot be cast to  
// Comparable
```

- Obviously it doesn't work this way! – What did we miss?
- The problem is that nobody (esp. *TreeMap*) knows, how to compare *Cars*!
 - One way to implement this: *Car* could itself know, how it can be compared – we've to implement *Comparable<Car>* in *Car*.
 - Comparable<T>.compareTo()* should return a value less than 0, if *this* < *other*, a value greater than 0, if *this* > *other* and otherwise 0.
 - For *Car* we'll delegate the implementation of *Comparable<Car>.compareTo()* to *Car*'s field *name*, *name* is of type *String* (*String* implements *Comparable<String>*).
- With this implementation of *Car* *TreeMap<T, V>* works as expected:

```
public class Car implements Comparable<Car> { // (members hidden)  
    public int compareTo(Car other) {  
        // Very simple implementation that compares the Cars' name fields:  
        return this.getLicencePlateID().compareTo(other.getLicencePlateID());  
    }  
}
```

```
Map<Car, String> carToManufacturer = new TreeMap<>();  
carToManufacturer.put(car1, "Ford"); // OK!  
Car car2 = new Car();  
car2.setLicencePlateID("KIB-HT 26");  
carToManufacturer.put(car2, "Toyota");
```

33

- The associative collection does really look into the items: just set a breakpoint into the *Car.compareTo()* method!

UDTs as Keys of associative Collections – Part II

- But there are cases, in which implementing `Comparable<T>` is no option to make types usable as keys:
 - (1) if we can not modify the type in question, e.g. if it is a type of a 3rd party library,
 - (2) if the type in question does already implement `Comparable<T>`, but the implemented comparison doesn't meet our needs,
 - (3) if the type doesn't implement `Comparable<T>` at all.
- Let's assume that we want to use `Cars` as keys by case insensitive comparison of their licence plate ids.
 - We could change the implementation of `Car.compareTo()`, but this would hurt the Open-Close Principle (OCP).
 - Effectively we have the case (2). The present implementation of `Comparable<Car>` doesn't meet our needs.
- To cover these cases there is another way to implement comparison: implement comparison in a different type (not in `Car`).
 - I.e. we can implement a dedicated type that has only one responsibility: comparing `Cars`. Such a type is called comparator.
 - Java provides a special interface to support such special comparison types: `Comparator<T>`.
 - Now it's time to implement a comparer to compare the case insensitive licence plate ids of `Cars`.
- Takeaway: `TreeMaps` internal organization can be controlled by:
 - (1) the key-type's implementation of `Comparable`, or
 - (2) a `Comparator` passed to the `TreeMap` from outside (in this case the key-type need not implement `Comparable` at all).

UDTs as Keys of associative Collections – Part III

- The type `TreeMap<K, V>` accepts an object of type `Comparator<T>` in one of its ctors:

```
// The generic type TreeMap<TKey, TValue>:  
public class TreeMap<K, V> { // (details hidden)  
    public TreeMap(Comparator<K> comparator);  
}
```

- As can be seen the passed comparator refers to the key-type K.

- The comparator to specify in the ctor determines, how keys (Cars) are getting compared. The comparator looks like this:

```
public class CaseInsensitiveCarComparator implements Comparator<Car> {  
    public int compare(Car lhs, Car rhs) {  
        // Very simple implementation that compares the licence plate ids  
        // fields of two Cars ignoring the case:  
        return lhs.getLicencePlateID().compareToIgnoreCase(rhs.getLicencePlateID());  
    }  
}
```

Hint

lhs and *rhs* (for "left hand side" and "right hand side") are traditional parameter names for binary operators. So those parameters are operands in this case. Binary operators are basically functions, that follow the concept of a mathematical 2-ary operators like `compare()`.

- Here we'll finally pass an instance of the created `CaseInsensitiveCarComparator` to `TreeMap`'s ctor:

```
// Create a map and specify the special comparator in the ctor:  
Map<Car, String> carToManufacturer = new TreeMap<>(new CaseInsensitiveCarComparator());  
Car car1 = new Car();  
car1.setLicencePlateID("KL-SD 14");  
carToManufacturer.put(car1, "Ford");  
// Let's add another Car with the licence plate id "kl-sd 14", which case-insensitively equivalent to "KL-SD 14":  
Car car2 = new Car();  
car2.setLicencePlateID("kl-sd 14");  
// This won't add another "kl-sd 14", because the specified comparer evaluates "KL-SD 14" and "kl-sd 14" as being equivalent:  
carToManufacturer.put(car2, "Toyota"); // Will set "KL-SD 14"'s value to "Toyota".  
System.out.printf("Count of items: %d\n", carToManufacturer.size());  
// >Count of items: 1
```

Controlling String-Key Comparison

- If we use *Strings* as keys (i.e. not a UDT like *Car*), we can use predefined *Comparators* to compare case insensitively.
 - Java provides a special *Comparator* for case-insensitive *Strings*, namely *String.CASE_INSENSITIVE_ORDER*:
 - Let's rewrite the color code example for case insensitive color names:

```
// Create a map and specify the case insensitive string comparator in the ctor:
Map<String, Integer> colorNamesToColorCodes = new TreeMap<>(String.CASE_INSENSITIVE_ORDER);
colorNamesToColorCodes.put("Blue", 0x0000FF);
colorNamesToColorCodes.put("Red", 0xFF0000);
colorNamesToColorCodes.put("Green", 0x00FF00);
// Because of the case insensitive comparison, "Green" and "green" are seen as equivalent key objects:
int colorCodeOfGreen = colorNamesToColorCodes.get("green");
System.out.printf("Color code for Green: %#08x%n", colorCodeOfGreen);
// >Color code for Green: 0x00ff00
```

- Another aspect of *TreeMaps* is their "order of iteration", which is defined by the applied *Comparator*.
 - By default, the items in the *TreeMap* are iterated in ascending sort order of their key as defined in the used *Comparator*:

```
colorNamesToColorCodes.forEach((key, value) -> {
    System.out.printf("Color name: %s, color code: %#08x%n", key, value);
});
// The output will be sorted ascending after the key, i.e. the color name:
// >Color name: Blue, color code: 0x0000ff
// >Color name: Green, color code: 0x00ff00
// >Color name: Red, color code: 0xff0000
```

Comparison with reversed Order

- We can further exploit sorting of *TreeMaps*: we can have the keys sorted in descending order by reversing the Comparator:

```
// Create a map and specify the case insensitive string comparator in the ctor, the sorting should be descending:
Map<String, Integer> colorNamesToColorCodesDescending = new TreeMap<>(Collections.reverseOrder());
colorNamesToColorCodesDescending.put("Blue", 0x0000FF);
colorNamesToColorCodesDescending.put("Red", 0xFF0000);
colorNamesToColorCodesDescending.put("Green", 0x00FF00);
```

- The simple factory *Collections.reverseOrder()* provides a *Comparator*, which just reverses the comparison.
- The reversed comparison leads to an internal organization of the *TreeMap*, which leads to handling items "upside down" in the tree.
- This also concerns the order of iteration, which is in descending order then:

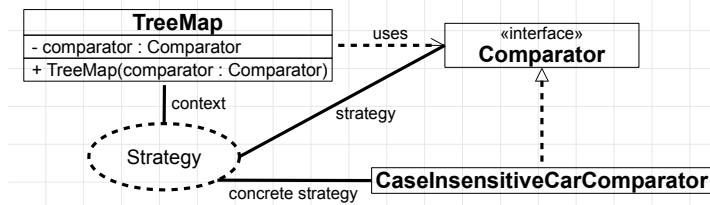
```
colorNamesToColorCodesDescending.forEach((key, value) -> {
    System.out.printf("Color name: %s, color code: %#08x%n", key, value);
});
// Now the output will be sorted descending after the key, i.e. the color name:
// >Color name: Red, color code: 0xff0000
// >Color name: Green, color code: 0x00ff00
// >Color name: Blue, color code: 0x0000ff
```

Comparator, TreeMap and the Strategy Pattern

- The idea behind *Comparators* is to separate the comparison algorithm from its usage.
 - E.g. *TreeMaps* allow to control its internal organization just by specifying a different *Comparator* at run time:

```
// The Comparator, i.e. the strategy to be applied in TreeMap can be determined at run time:  
Comparator<Car> comparator  
    = ascending  
    ? new CaseInsensitiveCarComparator()  
    : Collections.reverseOrder(new CaseInsensitiveCarComparator());  
Map<Car, String> carToManufacturer = new TreeMap<>(comparator);
```

- Notice, that the simple factory *Collections.reverseOrder()* can also be used to reverse the logic of a present *Comparator*.
 - Collections.reverseOrder()* is an example of the decorator pattern: it decorates a passed *Comparator* with a *Comparator* reversing the decorated *Comparator*'s logic.
- The interplay between *TreeMap* and the *Comparator* interface is an example of a famous oo design pattern: the strategy.



38

- Keep in mind: Creating a reversed *Comparator* by passing an existing *Comparator* to *Collections.reverseOrder()* is an example of the decorator pattern. A present *Comparator* is decorated with extra functionality and the new object has the same interface as the one being decorated.

Implementation of Associative Collections

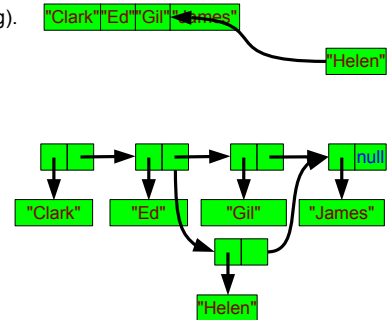
- Now we have a general idea of associative collections.
 - Associative collections allow associating keys with values.
 - Example: In white pages names are associated to phone numbers.
 - Associative collections allow to lookup a value for a certain key, e.g. getting the phone number of a specific person.
 - Associative collections could be implemented with two arrays as shown before, but this would be very inefficient for common cases.
- Now we're going to understand how associative collections are implemented.
- We'll start with *TreeMap*, which was introduced in the last lecture. We already know about *TreeMap* that
 - it allows only one value per key,
 - that it organizes its keys by equivalence,
 - that a Java type provides equivalence by implementing Comparable or delegating equivalence to objects implementing Comparator.
- Ok, the organization of keys is based on equivalence, but when a key is in a *TreeMap*, where does it have to "be"?
 - I.e. where and how will it be stored?
 - We're going to understand this in this lecture!

Associative Collections – What's in a TreeMap?

- In opposite to indexed/sequential collections, items got and set by analyzing their relationships in associative collections.
- The idea of *TreeMap* is to keep the contained items sorted by key whilst adding items!
 - So, its way of key organization is the analysis of the relative order/sort order of keys of the items.
- To make this happen, *TreeMap* is implemented with a tree-like storage organization for the keys.
 - When we analyzed algorithms we learned that the most efficient sorting algorithms can be visualized and analyzed with trees.
 - I.e. those implementing "divide and conquer", like mergesort and quicksort.
- Before we discuss *TreeMap*'s tree-based implementation, we're going to talk about alternative implementations.
- Btw. NET's implementation of a sorted associative collection is called *SortedDictionary*. It implements the [interface IDictionary](#).

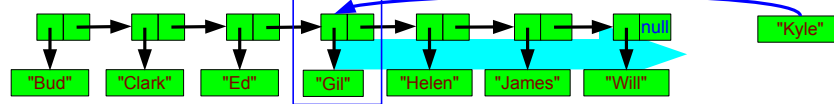
TreeMap – two alternative Implementations

- *TreeMap* could be implemented with a sorted list.
 - Finding would be fast ($O(\log n)$), but inserting would be costly ($O(n)$, because of resizing).
 - It is simple to code.
- *TreeMap* could be implemented with a sorted linked list.
 - Inserting would be fast if the location is known ($O(1)$), but finding would be costly ($O(n)$, finding involves the iteration of a linked list).
 - More memory per item is required (e.g. one item plus a few references, whereby (sorted) lists don't (just one item, due to continuous memory)).
- Problems of these approaches:
 - For (sorted) lists continuous memory is the problem. → It makes the data structure rigid. → Resizing required.
 - For (sorted) linked lists finding items is a problem due to the loose structure. → "Linked iteration"
- Let's finally discuss how we come from sorted lists and sorted linked lists to the real implementation of *TreeMap*.

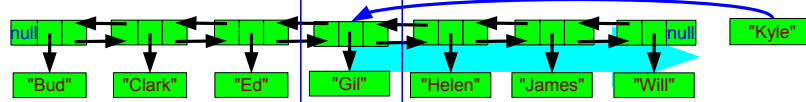


TreeMap – Coming to the real Implementation

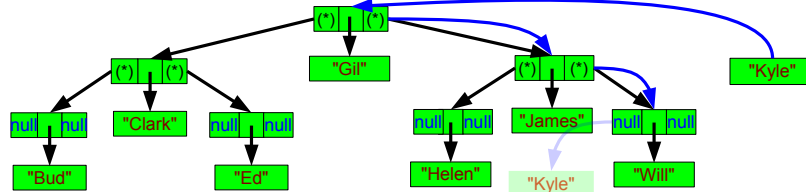
- When we examine a sorted linked list we'll find an interesting way of thinking about finding items.
 - If we had a reference to the middle item, we could determine the "direction" or half to insert a new item to keep the linked list sorted.



- Therefore it would be nice to allow moving into both directions, so we should use a sorted doubly linked list (3rd alternative)!



- The idea of referencing only the middle item of an already sorted doubly linked list can be recursively applied to the sorted halves.



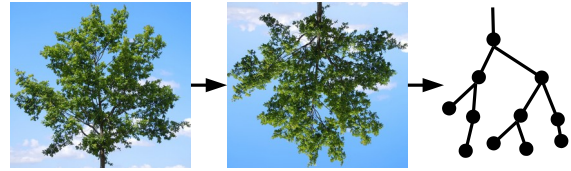
- The resulting structure is a tree, on which the real implementation of *TreeMap* is built upon!

Trees in Computer Science (cs)

- Trees represent one of the many data structures available in cs (others are graphs, relational, heap, stack etc.)
 - Similar to collections data structures can be categorized in different ways, e.g. a tree is a special form of a graph.

- Basic features of trees:

- Trees are recursive branching structures consisting of linked nodes.
- Trees have only one root node.
 - In cs the root of a tree is written on the top at graphical representations.
- A node can be reached by exactly one path from the root.
 - This is only true for linear paths, in xpath there exist various expressions to select any node from the root.



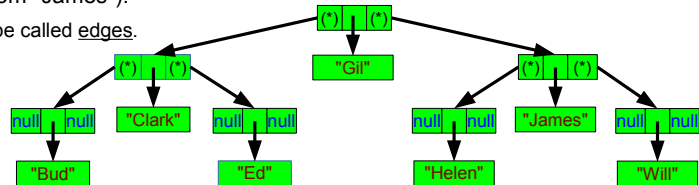
- Trees represent data that has a hierarchy. This is very common in computer business.
 - In the last example the hierarchy was the order: left => less than root, right => greater than root.
 - Directory and file structures.
 - A function call hierarchy: a call stack.
 - "Has a" and "is a" associations in object-oriented systems.
 - The structure of an HTML/XML file or a JSON structure.

Trees – Terminology

- The final solution we found to handle sorted collections is a special form of a tree, a binary search tree (BST).
 - The implementation of *TreeMap* is based on a BST (BST-based).
 - The core strategy of BSTs is to maintaining pointers/references to the middle of subtrees.

- Here some general terminology on trees (seen from "James"):

- "James" is a node, vertex or cell. The arrows can be called edges.
- "James" has two children.
 - Below "James" there is a subtree of two children.
- "James" parent is "Gil".
- "Gil" is the root of the tree.
 - In computer science trees are evolving upside down, having the root at the top.
- "Will" is a leaf, i.e. it has no children at all.
 - A node having only one child is sometimes called half-leaf.
- "Clark" and "James" are siblings.



- The BST is called "binary", because each node has at most two children.
 - A binary search can be issued on a BST in a very simple manner.

44

- There also exist ternary, n-ary etc. trees.
- There also exist trees, which rather have a "strong" trunk from which branches divide. E.g. trees of version control systems.

Traversal of Trees

- Basically all operations on trees involve recursion!
- The iteration of a tree is called traversal. It means to visit every node in the tree from a certain starting node. Schema:
 - (1) Do something with the current node.
 - (2) If the left (or right node) is not null make it the current node and continue with (1). → Recurse!
- There are three classical ways to traverse (binary) trees (seen from root):
 - Inorder: visit the left node, then the root and finally the right node.
 - Postorder: visit the left node, then the right node and finally the root.
 - Preorder/depth-first: visit the root, then the left node and finally the right node.

Node
+ value : Object
+ left : Node
+ right : Node

```
public static void printTreeInorder(Node root) {  
    if (null != root) {  
        printTreeInorder(root.Left); // Recurse!  
        System.out.println(root.Value);  
        printTreeInorder(root.Right); // Recurse!  
    }  
}
```

```
public static void printTreePostorder(Node root) {  
    if (null != root) {  
        printTreePostorder(root.Left); // Recurse!  
        printTreePostorder(root.Right); // Recurse!  
        System.out.println(root.Value);  
    }  
}
```

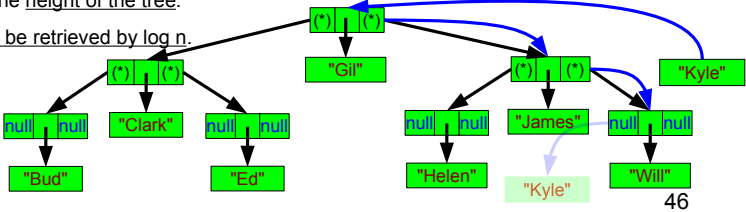
```
public static void printTreePreorder(Node root) {  
    if (null != root) {  
        System.out.println(root.Value);  
        printTreePreorder(root.Left); // Recurse!  
        printTreePreorder(root.Right); // Recurse!  
    }  
}
```

- The most important traversal for BSTs is inorder, because it visits the nodes in sorted order.
 - Notice, how the way of traversal could be encapsulated behind the iterator design pattern!

45

- The deletion of all nodes in a BST needs to be done postorder.
- Depth-first, also called Depth-First Search (DFS)
- Among others there also exists "level-order", which is also called "breadth-first" (Breadth-First Search (BFS)), it iterates all nodes beginning at the root from left to right.

TreeMap – Closing Words – Part I

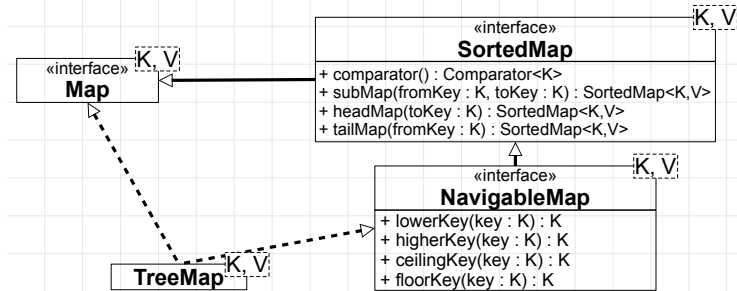
- So *TreeMap* is implemented with a BST.
 - The *String* that was used in each node in former examples, plays the role of *TreeMap*'s search key.
 - The nodes of *TreeMap* have one more field: the value that is associated with the search key.
- | | |
|--------|-------|
| "Gil" | key |
| 8758 | value |
| (*) | left |
| (*) | right |
| (Node) | |
- Using a BST in the implementation allows fast implementations to find items.
 - E.g. the methods *put()*, *get()*, *containsKey()*, *remove()* and *replace()* are based on key-equivalence on *TreeMap*'s BST to locate items.
 - The involved search algorithms are implemented via binary search (usually recursively).
 - Complexity.
 - The costs for finding one item only depends on the height of the tree.
 - We've already discussed that a tree's height can be retrieved by log n.
 - This is valid for perfectly balanced trees.
 - Other costs like swapping points will not be taken into consideration as always.
 - This makes the complexity for finding (and related operations) $O(\log n)$ for BSTs.
 - Max. ten comparisons to find an item in 1000 items.
- 

- Looking at the key we know on which half an item has to be inserted or found. (left => less than root; right => greater than root, this makes divide and conquer work).
- "Delete node" is often the most complex operation. The node to be deleted must be unlinked and if it had children (esp. costly if it had two children) they must be relinked to maintain the order of the tree.
- BSTs can be kept balanced internally to maintain logarithmic behavior, this yields an additional cost factor.
 - Java's *TreeMap* red-black tree implementation is self-balancing.

TreeMap – Closing Words – Part II

```

NavigableMap<String, Car> ownersToCars = new TreeMap<>();
ownersToCars.put("Bud", new Car("KL-GS 23"));
ownersToCars.put("Gil", new Car("SB-DF 522"));
ownersToCars.put("James", new Car("WND-KR 30"));
ownersToCars.put("Will", new Car("KL-VV 98"));
ownersToCars.put("Helen", new Car("WND-SG 76"));
ownersToCars.put("Clark", new Car("KIB-TI 52"));
ownersToCars.put("Ed", new Car("KIB-MJ 49"));
    
```



- *TreeMap* implements the interface *NavigableMap* and *NavigableMap* implements *SortedMap* to visit elements in order.

- *SortedMap* offers methods, which e.g. yield sorted sub-Maps:

```

Map<String, Car> belowHelen = ownersToCars.headMap("Gil");
belowHelen.forEach((name, car) -> {
    System.out.printf("%s drives '%s' %n", name, car.getLicencePlateId());
});
// >Bud drives 'KL-GS 23'
// >Clark drives 'KIB-TI 52'
// >Ed drives 'KIB-MJ 49'
    
```

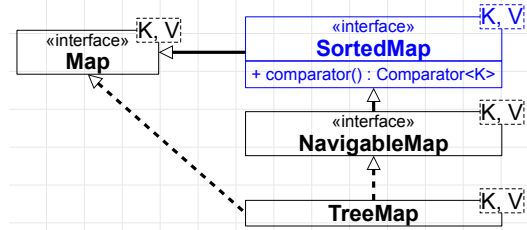
- *NavigableMap* offers methods to navigate in sorted Maps, e.g. to get the key, which is equivalently above or below the passed key:

```

String belowHelen = ownersToCars.higherKey("Helen");
// belowHelen = "James"
    
```

TreeMap – Closing Words – Part III

```
Map<String, Car> ownersToCars = new TreeMap<>();
ownersToCars.put("Bud", new Car("KL-GS 23"));
ownersToCars.put("Gil", new Car("SB-DF 522"));
ownersToCars.put("James", new Car("WND-KR 30"));
ownersToCars.put("Will", new Car("KL-VV 98"));
ownersToCars.put("Helen", new Car("WND-SG 76"));
ownersToCars.put("Clark", new Car("KIB-TI 52"));
ownersToCars.put("Ed", new Car("KIB-MJ 49"));
```



- That `TreeMap` implements `SortedMap` stresses an important detail: `TreeMaps` traverse after the keys in sorted order:

```
ownersToCars.forEach((name, car) -> {
    System.out.printf("%s drives %s%n", name, car.getLicencePlateId());
});
// >Bud drives 'KL-GS 23'
// >Clark drives 'KIB-TI 52'
// >Ed drives 'KIB-MJ 49'
// >Gil drives 'SB-DF 522'
// >Helen drives 'WND-SG 76'
// >James drives 'WND-KR 30'
// >Will drives 'KL-VV 98'
```

- The traversal order is only controlled by the way `TreeMap` organizes its elements via equivalence.
- Mind, that `SortedMap` is also the place, from which the `Comparator` of the equivalence-based management-strategy is accessible.

Thank you!