(1) Java Advanced: Generics

Nico Ludwig (@ersatzteilchen)

TOC

- Cited Literature:
  - Just Java, Peter van der Linden
  - Thinking in Java, Bruce Eckel
  - https://docs.oracle.com/javase/tutorial/extra/generics/morefun.html

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 – Part 1

- The type system is 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.
```

- The type system is safely typed – i.e. unrelated types can't fool the compiler.
  - Cross casting is illegal in most situations (exception: interface casts), types must be related in to allow correct type conversions.

- The type system is 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 – Part 2

- Esp. *Object*-based collections rely on the cosmic hierarchy of Java's type system: everything is an *Object*:
  - Just mind collections like *List*s: they hold items of static type *Object*, their dynamic types can vary.

```
List names = new ArrayList();
// Add elements:
names.add("Sally");
names.add("Peter");
// We have to use cast contact lenses to get the dynamic types out of the objects:
String name = (String)names.get(0);
```

- In fact, collections must be able to hold any type in order to be versatile.
  - Java "resorts" to run time polymorphism (i.e. Liskov Substitution Principle (LSP)) to get this versatility.
  - So in a Java 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 – Part 3

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

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

```
public class Car {
        public void startEngine() {
                // pass
        }
}
```

- The problem is that we as programmers must know about the contained dynamic types (*String*s and not *Car*s in *names*).
  - Whenever *Object* is used in an interface (the return value of *names*' *List.get()*), some convention must be around.
  - This convention describes the dynamic type, we're really dealing with (*String*s and not *Car*s).
  - The contributed programers 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 programer 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 *names* 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: we can have heterogeneous collections:
  - In a heterogeneous collection every item can have any different dynamic type:

```
List objects = new ArrayList();
// Since each item is of type object, we can add any object to a List:
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);
}
```

- 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!
// A ClassCastException will be thrown!
String result =(String)objects.get(1);
```

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

Some Remedy by using Arrays as Collections

- What could we do to close the lack in the Java type system to stay type safe?

- We could use "strictly" typed arrays as collections if possible, they're not *Object*-based and therefore more type safe:

```
// Create an array that hold strings:
String[] names = { "Sally", "Peter" };
// With arrays we don't need casts, instead we can directly access names' items as String objects:
String name = names[0];
// Access String's interface:
int result = name.indexOf('g');
```

- Benefits:
  - type safety is given
  - working with arrays is simple
  - arrays are highly optimized, because of the support in the JVM

- Downsides:
  - arrays are inappropriate, if the collection in question needs to be modified (enlarged or downsized) after creation

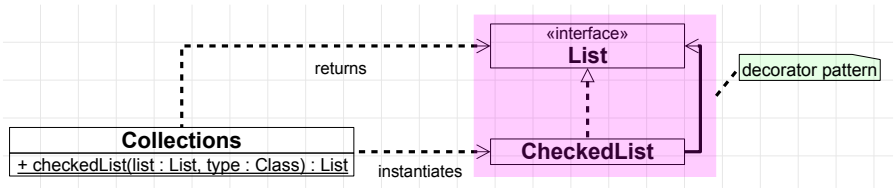
Run Time checked wrappers for Collections

- Back to *List*s: the companion class *java.util.Collections* provides a method to create run time type checked *List*s.
  - A present *List* needs to be passed to *Collections.checkedList()* and a new *List* wrapping the passed one will be returned.
  - Let's wrap *a List*, which holds the dynamic type *String* to be a checked *List* with *Collections.checkedList()*:

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

  - As can be seen, we have to specify the *List* to be wrapped (yet empty or with content) and the awaited run time type's *Class* object.

- An UML class diagram shows the 20k miles perspective of *Collections.checkedList()* and *List*:



  - => *Collections.checkedList()* is an implementation of the decorator pattern!

9

Generics to the Rescue – Part 1

- Let's improve our problematic code by using *List*/*ArrayList* as generic type, namely as *List<E>*/*ArrayList<E>*:

```
// Create a generic List that holds Strings:
List<String> names = new ArrayList<String>();
names.add("Sally");
names.add("Peter");
// This time we don't need casts, instead we can directly access names' items as String objects:
String name = lines.get(0);
// Access String's interface:
int result = name.indexOf('g');
```

  - The new syntax aspect is, that we pass a type name as argument into angle brackets right after the type names *List* and *ArrayList*.
  - *List* is still used as static type, whereas *ArrayList* is the dynamic type. – The DIP should also be applied on generics!

- When we specify a type argument on generic types like *List*/*ArrayList*, their interfaces are fixed to these type arguments.
  - The *List<String> names* is now controlled by the compiler to be only used with *String*s.
  - Trying to get a *Car* object from a *List<String>* results in a compile time error:

```
// This time we'll get a compile time error:
Car car1 = names.get(1); // Invalid! incompatible types: String cannot be converted to Car
car1.startEngine(); // Will not be reached at all!
```

    - The important point is, that this is no longer a check done at run time, i.e. no cast is performed! – The compiler knows, that a *String* is awaited!
  - Trying to add something different from a *String* also results in a compile time error:

```
names.add(42); // Invalid! argument mismatch; int cannot be converted to String
Object itemAt1 = names.get(1); // Will not be reached at all!
```

Generics to the Rescue – Part 2

- Put simple, the idea behind generic types is, that a type offers another type used in its interface as parameter!

```
// The generic type List<T>:
public interface List<E> { // (members hidden)
        E get(int index);
        void add(E e);
}
```

```
// The generic type ArrayList<T>:
public class ArrayList<E> implements List<E> { // (members hidden)
        @Override
        public E get(int index) {
                // pass
        }
        @Override
        public void add(E e) {
                // pass
        }
}
```

- Java 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 5.
- Java supports generic classes, interfaces and methods.
- Generics (here *List<E>*) leave type information open? How should this help me out?
  - The open type information is accessible by type parameters (here *E*).
  - As programmers we can fill the type parameters with concrete type arguments, e.g. for *List<String>*, *String* is the type argument.
  - => 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 only at run time.

Generics Overview – Part 1

- The place, where generic types are parameterized can be precisely defined in Java.
  - Before discussing this is depth, let's first have another look at Java's *Object*-based collections.

- Java's *Object*-based collections in *java.utils* already offer type parameters to set a type argument, instead of using *Object*.

```
// The generic type ArrayList<T>:
public class ArrayList<E> implements List<E> { // (members hidden)
        @Override
        public E get(int index) { /* pass */ }
        @Override
        public void add(E e) { /* pass */ }
}
```

- As as soon as we specify a type argument, when using an *ArrayList* type, we get a specific *ArrayList* type.

```
// The generic type ArrayList<T>:
ArrayList<String> stringList = new ArrayList<String>();
```

the compiler assumes *String*, were *E* was specified

```
public class ArrayList<String> implements List<String> {
        @Override
        public String get(int index) { /* pass */ }
        @Override
        public void add(String e) { /* pass */ }
}
```

```
// The generic type ArrayList<T>:
ArrayList objectList = new ArrayList();
```

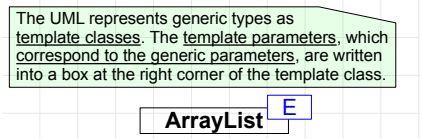
the compiler assumes *Object*, were *E* was specified

```
public class ArrayList<Object> implements List<Object> {
        @Override
        public Object get(int index) { /* pass */ }
        @Override
        public void add(Object e) { /* pass */ }
}
```

12

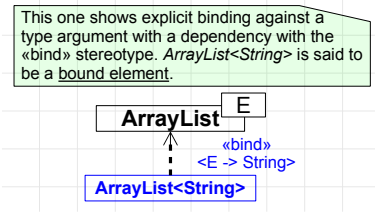
Generics Overview – Part 2

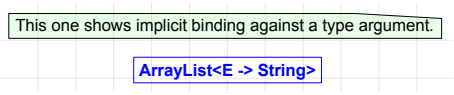
- The UML supports a representation for generic types, so called template classes:




```
// The generic type ArrayList<T>:
public class ArrayList<E> implements List<E> { // (members hidden)
        @Override
        public E get(int index) { /* pass */ }
        @Override
        public void add(E e) { /* pass */ }
}
```

- The template parameters must be filled with type arguments.
  - The UML calls this "binding types to template parameters":





Generics Overview – Part 3

- The implementation of own generic types is simple, consider following generic class *Box*, which holds exactly one field:

```
// <Box.java>
public class Box<T>  {
        private T value;

        public T get() {
                return this.value;
        }

        public void set(T value) {
                this.value = value;
        }
}
```

- The type parameter can be used in any instance method, also in the ctor, for method-parameter types and return types.
- static methods can also be generic, this is an extra topic, not discussed right now.
- The name of the type parameter can be any valid Java identifier.
  - But, Java offers a convention: upper case latin letters should be used.
  - *T* (for type) for the most common case, *K* for key types and *E* for the element type in collections (like *ArrayList*<*E*>).
  - Since Java 8 *R* is the recommended identifier for return types in so called functional interfaces.

Generics Overview – Part 4

- When we create *Box* instances, we have to decide, which type argument will be used.

```
Box<String> stringInABox = new Box<String>();
stringInABox.set("Hello there!");
System.out.println("The text: "+stringInABox.get());
// >the text: Hello there!
```

- We can remove the second mention of the type argument, when instantiating a generic class:

```
Box<String> stringInAnotherBox = new Box<>();
```

  - Because the left pair of empty angle brackets resembles a diamond, this construct is sometimes called diamond operator.
    - Although, it is not an "operator" at all ...
  - The compiler can automatically determine or "deduct" the diamond's type argument, this is called type inference or type deduction.

- However, if we leave the type argument and the angle brackets itself away, *Box*' type argument is implicitly *Object*:

```
Box anythingInABox = new Box();   <-->   Box<Object> anythingInABox = new Box<>();
```

  - When a generic type is used w/o specifying a type argument, it is called raw type.
  - The *Object*-based *ArrayList* is also a raw type.

Generics Overview – Part 5

- There is a important restriction on the type arguments we can use: only reference types can be used!
  - Esp. this means, we cannot use primitive types, so we cannot create a *Box<int>*!
  - Instead we have to create a generic types with the wrapper type of the respective primitive type, e.g. *Box<Integer>*:

```
Box<int> intInABox = new Box<>(); // Invalid! Type argument cannot be of primitive type.
```

```
Box<Integer> intInABox = new Box<>(); // Ok!
intInABox.set(24); // Boxing
int theValue = intInABox.get(); // Unboxing
```

- A very mighty feature of generics is, that type arguments can itself be generics, consider an *ArrayList* of *Box<String>*:

```
ArrayList<Box<String>> stringBoxes = new ArrayList<>();
Box<String> stringBox = new Box<>();
stringBox.set("Agamemnon");
stringBoxes.add(stringBox);
```

  - As can be seen, the diamond operator can also be used to instantiate generics of generic types.

- This way we can also define an *ArrayList* of *Integer-ArrayList*s, which makes it quasi-multidimensional like a matrix:

```
ArrayList<ArrayList<Integer>> matrix = new ArrayList<>();
matrix.add(new ArrayList<>(Arrays.asList(1, 2, 3, 4)));
matrix.add(new ArrayList<>(Arrays.asList(6, 7)));
matrix.add(new ArrayList<>(Arrays.asList(8, 9, 0)));
System.out.println("Item: "+matrix.get(1).get(0));
// >Item: 6
```

  - This usage of collections is better than multidimensional arrays in some aspects:
    - "real" collections like *ArrayList*s are very flexible, as elements can be added and removed (or: the *matrix*' dimensions can be enlarged and shrunk).

16

Generic Types with multiple Type Parameters – Part 1

- Generics can have multiple type parameters, consider the generic class *Pair*, that holds two values:

```
// <Pair.java>
public class Pair<T, S>  {
        private T first;
        private S second;

        public T getFirst() {
                return this.first;
        }

        public void setFirst(T first) {
                this.first = first;
        }

        public S getSecond() {
                return this.second;
        }

        public void setSecond(S second) {
                this.second = second;
        }
}
```

- The written order of type parameters has no meaning in Java.

- Also for multiple type parameters, Java offers conventions: *T*, *S*, *U*, *V* etc. should be used for 1st, 2nd, 3rd, 4th type parameter.
  - Alternatively numbers can be attached, like: *T1*, *T2*, *T3* … *Tn*.

17

- Typically at this point of the presentation people ask how many type parameters we can have at maximum in Java. This seems not to be clearly defined, however some JVM-languages show in their examples up to 26 parameters. 26 parameters are already too many also for advanced code I think...

Generic Types with multiple Type Parameters – Part 2

- The idea is, that different type aspects of a generic type can be left open and be parameterized differently.
  - Yes … and in oo terms, different (type) aspects can be abstracted as different types.
  - So, each type argument can be a different type, consider:

```
Pair<Integer, String> intStringPair = new Pair<>();
intStringPair.setFirst(42);
intStringPair.setSecond("fortytwo");
System.out.println("The int: "+intStringPair.getFirst()+", the String: "+intStringPair.getSecond());
// >The int: 42, the String: fortytwo
```

  - As can be seen, the diamond operator can also be used to instantiate generic types with multiple type arguments.

- We can of course also have generics of generics with multiple type parameters:

```
Pair<String, Pair<Integer, String>> stringIntStringPairPair = new Pair<>();
stringIntStringPairPair.setFirst("aPair");
stringIntStringPairPair.setSecond(intStringPair);
System.out.println("First String: "+stringIntStringPairPair.getFirst()+", the int: "
        +stringIntStringPairPair.getFirst().getSecond().getFirst()
        +", the String: "+stringIntStringPairPair.getSecond().getSecond());
// >First String: aPair, the int: 42, the String: fortytwo
```

- Generic types with multiple type arguments play an important role for functional interfaces.

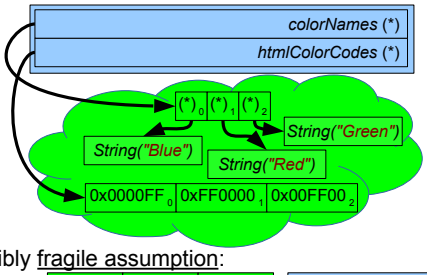
18

Maps

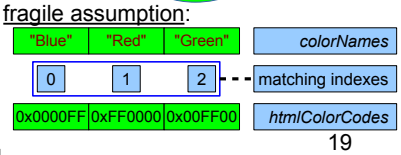
- After we have introduced generics with multiple type parameters, it is time to introduce another type of collection: the *Map*.
- More specifically, we'll use *HashMap* as implementation of the interface *Map*. We will look at a problem we want to solve.
- Let's assume that we have to handle color names and figure out the belonging to HTML color codes:

```
// Here we have two arrays that hold color names and HTML color codes:
List<String> colorNames = Arrays.asList("Blue", "Red", "Green");
List<Integer> htmlColorCodes = Arrays.asList(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: %06X%n", colorCodeOfRed);
// >Color code for Red: FF0000
```



- Of course the task can be solved this way, but the solution is based on a possibly fragile assumption:
  - The two separate arrays 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 between *colorNames* and *htmlColorCodes* is index-based.

19

- There is also another technical limitation: using an int-index to associate elements means we can have max. *Integer.MAX_VALUE* associated elements.

Example – Color Names to HTML Color Codes – Part 1

- The presented algorithm is a so called lookup-algorithm (lookup).

```
// The lookup algorithm:
int indexOfRed = colorNames.indexOf("Red");
int colorCodeOfRed = htmlColorCodes.get(indexOfRed);
```

  - Lookups are used very often in programming.
  - A problem of lookups is, that separate *List*s (*colorNames* and *htmlColorCode*) need to be evaluated.

- As we know from oo-paradigm: separated data, which belongs together should be encapsulated into one concept!
  - Modern collection frameworks provide dedicated collections to cover lookups in a comfortable manner: associative collections.
  - Associative collections abstract the usage of multiple collections to solve the just presented lookup in an intuitive manner.

- Java's associative collections are called *Map*s, a versatile *Map* implementation is the *HashMap*:

```
// Somewhere in the JDK:
public class HashMap<K, V> implements Map<K, V> { // (simplified)
        // pass
}
```

  - A type parameter named *K* for key and *V* for value.

```
// Associating color names with HTML color codes using a HashMap:
Map<String, Integer> colorNamesToColorCodes = new HashMap<>();

colorNamesToColorCodes.put("Blue", 0x0000FF);
colorNamesToColorCodes.put("Red", 0xFF0000);
colorNamesToColorCodes.put("Green", 0x00FF00);

int colorCodeOfRed = colorNamesToColorCodes.get("Red");
System.out.printf("Color code for Red: %06X%n", colorCodeOfRed);
// >Color code for Red: FF0000
```

20

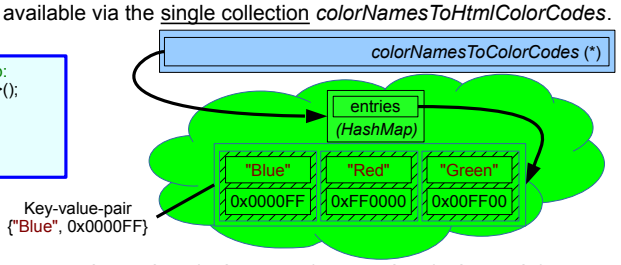
- The separated *ArrayList*s are connected via the index, similar to the way tables in relational databases are connected via primary and foreign keys.

Example – Color Names to HTML Color Codes – Part 2

- With *Map*, the association of color names to color codes is available via the single collection *colorNamesToHtmlColorCodes*.

```
// Associating color names with HTML color codes using a HashMap:
Map<String, Integer> colorNamesToColorCodes = new HashMap<>();

colorNamesToColorCodes.put("Blue", 0x0000FF);
colorNamesToColorCodes.put("Red", 0xFF0000);
colorNamesToColorCodes.put("Green", 0x00FF00);
```



- 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 *ArrayList* is nothing but an associative collection that associates a value with an (int) index.
  - In fact real problems often involve associating values with other values, e.g.: White pages, thesaurus, document index.

- However, from an idiomatic standpoint it is interesting, that *Map* (and also *HashMap*) is a generic.
  - The key type and the value type are separate type parameters in *Map*.
  - => When we use *Map* as raw type, key type and value type default to *Object*, then *Map* is an *Object*-based collection.

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.

Operations on HashMaps – Part 1

- Create the empty *Map colorNamesToColorCodes*:

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

- Then we can add three keys and values, i.e. key-value-pairs like with *Map.put()*:

```
colorNamesToColorCodes.put("Blue", 0x0000FF);
colorNamesToColorCodes.put("Red", 0xFF0000);
colorNamesToColorCodes.put("Green", 0x00FF00);
```

- Get the value of a key, i.e. make a lookup with *Map.get()*:

```
int valueForBlue = colorNamesToColorCodes.get("Blue");
// valueForBlue = 0x0000FF
```

- Calling *Map.get()* for a key that doesn't exist will return null:

```
int valueForBrown = colorNamesToColorCodes.get("Brown");  // Will throw NPE!
```

  - This call will even throw an NPE: *Map.get()* returns null as *Integer*-reference and unboxing this null-reference to int throws an NPE.
  - To avoid NPEs we can get the value boxed in an *Integer* and check it for being null:

```
Integer valueForBrown = colorNamesToColorCodes.get("Brown");
// valueForBrown = null
int unboxedValueForBrown = null != valueForBrown ? valueForBrown : -1;
```

- To avoid such surprises we should check the key for existence (with *Map.containsKey()*), before looking it up.

```
boolean brownAlreadyPresent = colorNamesToColorCodes.containsKey("Brown");
// brownAlreadyPresent = false
```


22

Operations on HashMaps – Part 2

- We can also iterate over the items contained in a *Map*, but not directly, because it doesn't extend *Collection*.
  - However, in opposite to *ArrayList*, the items we get are no single values, but key-value-pairs each.
  - As a *Map* is a collection of pairs, a *List* can be thought of as a collection of index-value-pairs.
- First, we have to get the key-value-pair entries of a *Map*, we do this with *Map.entrySet()*:

```
Set<Map.Entry<String, Integer>> entries = colorNamesToColorCodes.entrySet();
```

  - A *Set* is another generic collection, which is optimized to hold unique items. – Mind, that each key-value-pair must be unique!
  - *Map.Entry* is another generic interface, that is nested in the interface *Map*.

```
// Somewhere in the JDK
public interface Map<K, V> { // (details hidden)
        interface Entry<K, V> {
                K getKey();
                V getValue();
        }
}
```

  - The *Set* interface extends *Collection*, which extends *Iterable*, therefor *Set*s can be used in for each:

```
// Iterate all key-value-pair entries of a Map:
for (Map.Entry<String, Integer> entry : colorNamesToColorCodes.entrySet()) {
        System.out.printf("Color code for %s: %06X%n", entry.getKey(), entry.getValue());
}
```

- And finally, we can get the count of entries stored in the *Map* via *Map.size()*:

```
System.out.println(colorNamesToColorCodes.size());
// >3
```

Sets – a first Glimpse – Part 1

- Let's have a closer look at the generic type *Set*, which proves to be very useful on its own.

- Assume, we want to have a collection of names, in which each name should be unique. Actually we can do this with a *Map*:

```
List<String> names = Arrays.asList("Tina", "Hale", "Tim", "Joe", "Tina", "Tim");
Map<String, String> uniqueNames = new HashMap<>();
for (String name : names) {
        uniqueNames.put(name, null);
}
```

  - The "trick" we use: when we call *Map.put()* we just add name as a key and null as value.
  - And then, we'll request the *Map.keySet()*, which only contains the unique keys:

```
for (String name : uniqueNames.keySet()) {
        System.out.println(name);
}
```

Terminal

```
NicosMBP:src nico$ java Program
Hale
Joe
Tim
Tina
NicosMBP:src nico$
```

  - We "exploited" *Map*'s functionality to only keep one unique key per entry.

- We do not need to use *Map* to "exploit" stripping a collection down only to unique items, we can use a *Set* directly.

Sets – a first Glimpse – Part 2

- *Set* as a collection combines functionality of a *Map* with functionality of a *List*, think of a *Set* as a collection with uniqueness.
  - We get the same set of unique items, if we construct a *Set* from the *List* directly:

```
List<String> names = Arrays.asList("Tina", "Hale", "Tim", "Joe", "Tina", "Tim");
Set<String> uniqueNames = new HashSet<>(names);
```

Terminal

```
NicosMBP:src nico$ java Program
Hale
Joe
Tim
Tina
NicosMBP:src nico$
```

- After the *Set* was created, we can add more items using *Set.add()*:

```
uniqueNames.add("Ralph");
```

  - We can also remove items using *Set.remove()*.

- Contrary to *Map*s, *Set*s offer no way to get a value by a key, because in a *Set* the value is the key at the same time.
  - Rather when working with *Set*s we can ask, if a *Set* contains a value with *Set.contains()*:

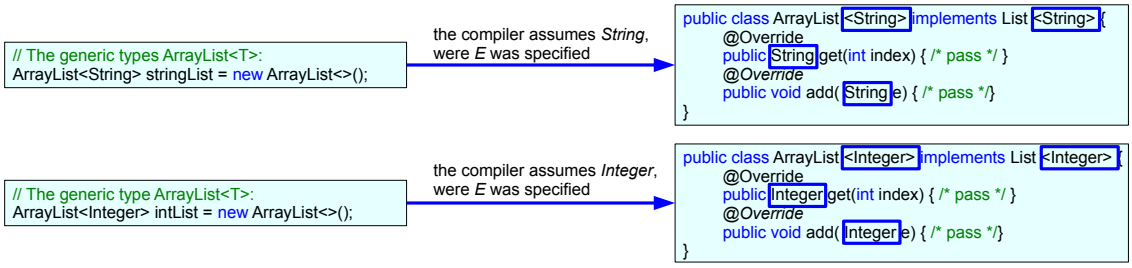
```
if (uniqueNames.contains("Tim")) {
        System.out.println("We have Tim!");
}
```

- Like *List*, *Set* supports means of iteration (implements *Collection*/*Iterable*) and offers the method *Set.size()*.

- We won't discuss *Set*s in more depth, for the topic of collections I offer a dedicated course.

Generic Type Erasure – Part 1

- As as soon as we specify all type arguments, we get a "specific" "filled in" generic type. Here, two cases for *ArrayList*:



- But, this situation is only the compiler's view, it "sees" the type arguments and can fill in what was left away.
- Actually at run time, the types *ArrayList<String>* and *ArrayList<Integer>* are the same!
  - We can prove this with a run time analysis of the both lists:

```
System.out.println(stringList.getClass());
System.out.println(intList.getClass());
System.out.println(stringList.getClass() == intList.getClass());
```

Terminal

```
NicosMBP:src nico$ java Program
class java.util.ArrayList
class java.util.ArrayList
true
NicosMBP:src nico$
```

  - Effectively this means, that the type argument is not longer available at run time.
  - Instead, at run time, *ArrayList* is always used as raw type, i.e. an *Object*-based *ArrayList*.
  - What the compiler does with filled in generics is therefore called type erasure (TE).

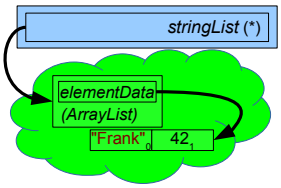
26

- 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 were .NET, without type erasure, but constructed types does shine.
- The expression *o instanceof ArrayList<String>* would not compile.

Generic Type Erasure – Part 2

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

```
public static void main(String… args) {
        List<String> stringList = new ArrayList<>();
        stringList.add("Frank"); // OK!
        ((ArrayList)stringList).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.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
}
```



  - This is a kind of pathological case, but demonstrates the potential problem.

- At least it is most often not a problem for users of generic types, but for implementors.
  - Problems with TE get visible, if we have overloads of *Object*- and type parameters:

```
// <MyGenericClass.java> (1)
public class MyGenericClass<T> {
        public void f(T t) { /* pass */ }
        public void f(Object o) { /* pass */ } // Invalid!
}
```

```
// <MyClass.java> (2)
public class MyClass {
        public void set(List<String> strList) {/* pass */ }
        public void set(List<Integer> intList) { /* pass */ } // Invalid!
}
```

  - (1) The compiler message unleashes the conflict: "'f(T)' clashes with 'f(Object)'; both methods have same erasure".
  - (2) The compiler message unleashes the conflict: "set(List<Integer>) and set(List<String>) have the same erasure".

27

Generic Type Erasure – Part 3

- The Java compiler can warn, if we use a raw type, instead of a filled in generic type.
  - To have the compiler considering this, we have to compile with the option **-Xlint:unchecked**:

```
public static void main(String... args) {
        List<String> stringList = new ArrayList<>();
        stringList.add("Frank");
        ((ArrayList)stringList).add(42);
}
```

Terminal

```
NicosMBP:src nico$ javac System.java -Xlint:unchecked
System.java: warning: [unchecked] unchecked call to add(E) as a member of the raw type ArrayList
        ((ArrayList)stringList).add(42);
                                   ^
  where E is a type-variable:
    E extends Object declared in class ArrayList
1 warning
NicosMBP:src nico$
```

  - So, the compiler calls using a generic type as a raw type "unchecked" usage, hence the warning message.
  - The unchecked warning can occur when interfacing with legacy code written before the advent of generics.

- The best way to deal with such a warning is of course just to use generics in favor to raw types.
  - However, sometimes, this is not possible, e.g. if we have to use pre Java 5 code without generic types in their interfaces.
  - Java allows to suppress such warnings for the scope of a method or full class with the *@SuppressWarnings("unchecked")* annotation.

```
@SuppressWarnings("unchecked")
public static void main(String… args) {
        List<String> stringList = new ArrayList<>();
        stringList.add("Frank");
        ((ArrayList)stringList).add(42);
}
```


28

Generic Type Erasure – Part 4

- Due to TE it is not allowed to inherit from a type parameter or to "implement a type parameter".

```
public class MyClass<T> extends T { // Invalid! Class cannot inherit from its
        // pass                          // typt parameter
}
```

```
public class MyClass<T> implements T { // Invalid! Interface expected here
        // pass
}
```

- But Java allows to extend or implement generic super classes or interfaces and forwarding type parameters as arguments:

```
public class MyClass<T> extends Box<T> { // OK!
        // pass
}
```

```
public class MyClass<T> implements GenericInterface<T> { // OK!
        // pass
}
```

- Generic classes cannot extend *Throwable*:

```
public class MyException<T> extends Throwable { // Invalid! A generic class may not extend java.lang.Throwable
        // pass
}
```

  - The reason: catch-clauses are selected at run time, a catch-handler like *catch(MyException<String> exc)* can't work.
  - Because *MyException<String>* doesn't exist at run time.

29

Generic Type Erasure – Part 5

- TE also implies that any kind of run time type checks won't work on generic types:

```
// <Box.java>
public class Box<T> {
	public void doSomeChecks(Box<T> t) {
		if (t instanceof Box<String>) { // Illegal! illegal generic type for instanceof
			// pass
		}
	}
}
```

- Also reflection of generics or type parameters doesn't work:

```
// <Box.java>
public class Box<T> {
	public void doSomeReflection() {
		Class boxTClass = Box<T>.class; // Illegal!
		Class Tclass = T.class; // Illegal!
	}
}
```

- The idea behind TE it that byte code using generics is still compatible to pre-generics byte code.
  - This binary backwards compatibility is important for generic collections, which are equivalent to *Object*-based collections in Java < 5.

Generic Type Erasure – Part 6

- When we disassemble Java code using a generic type, we see how the compiler handles TE:

```
// Java code:
List<String> names = new ArrayList<>();
names.add("Sally");
String name = names.get(0);
```

```
// Byte code mnemonics:
0:  new          #7                  // class java/util/ArrayList
3:  dup
4:  invokespecial #9                 // Method java/util/ArrayList."<init>":()V
7:  astore_1
8:  aload_1
9:  ldc          #10                 // String Sally
11: invokeinterface #12, 2           // InterfaceMethod java/util/List.add:(Ljava/lang/Object;)Z
16: pop
17: aload_1
18: iconst_0
19: invokeinterface #18, 2           // InterfaceMethod java/util/List.get:(I)Ljava/lang/Object;
24: checkcast    #22                 // class java/lang/String
27: astore_2
```

  - (The class-file was disassembled with the command j**avap -c**. **javap** is a command-line tool that is bundled with the JDK.)
  - In the metadata (byte code comments) of the method calls *List.add()* and *List.get()* we see after TE the signatures show *Object*.
  - We also see that the compiler adds a cast, which brings run time type safety, but adds the burden of a run time checked cast.
    - *List.get()* only returns *Object*, which still needs to be casted to *String*, however this cast is added by the compiler.
    - => Generic types only add type safety, that is forced at compile time and checked at run time, but no improved performance!

31

- In the byte code:
  - The leading numbers on each line are byte-offsets from the start address of the enclosing method (i.e. they are no line numbers). They appear to have "gaps", because each opcode (**new**, **invokespecial** etc.) is represented by only one byte, but it may have zero or more arguments, which also occupy different numbers of bytes between the opcodes.
  - The #-symbols are references to indexes of the constant pool.
  - It is required to call **dub** right after **new** was called, because **invokespecial** will consume and remove the value created by **new** from the operand stack, but we want to continue using this value.
  - In the comments:
    - Method java/util/ArrayList."<init>":()V – The V means that the ctor is actually a method returning void.
    - InterfaceMethod java/util/List.add:(Ljava/lang/Object;)Z – The Z means that List.add() is actually a method returning boolean.
    - InterfaceMethod java/util/List.get:(I)Ljava/lang/Object; – The I means that List.get() accepts an int.

Reified Types

- Types, which are not subject to TE in Java are called reifiable types, others are non-reifiable type.
  - => Reification from lat. res "thing" und facere "create", another word for "solidification".

- Reifiable types
  - Primitive types: int, long
  - Non parameterized class or interface: *String*, *Car*
  - All type arguments are unbound wildcards: *List<?>*, *Map<?, ?>*
  - Raw types: *List*, *Map*
  - Arrays of reifiable types: int[][], *List<?>[]*

- Java's generic types are non reifiable types:
  - Generic type parameters: *T*
  - Generic types with parameters: *List<String>*, *Map<Integer, String>*
  - Generic types with bounds: *List<? extends Number>*, *Consumer<? super String>*

32

- The topic of non reifiable types involves some other complex subjects in Java, which will not be discussed in this course, esp. heap pollution and safe varargs.
  - See: https://docs.oracle.com/javase/tutorial/java/generics/nonReifiableVarargsType.html

Excursus: Reified Generics

- There is a lot of critique on the way Java provides parameterized typing via generics as non-reifiable types:
  - Another perspective on non-reifiable types: they are not known at run time.
  - This implies, that due to TE we have no type safety at run time.
  - TE leads to many casts at run time (we don't see them in code), which reduces performance.
  - Generics cannot be used with primitive types, instead wrapper types must be used, which leads to many boxing/unboxing operations.
- To address these critique points it is discussed to extend Java with so called reified generics.
  - Reified generics should allow to build real constructed types from parameterized types with solidification at run time.
  - So in reified generics the type argument would be known at run time: we have no TE.
  - The idea of constructed types is present in .NET.
  - If and when reified generics will be added to Java is not clear. Kotlin supports reified generics.
- Run time type checked wrappers (*Collections.checkedList()*) get some solidification for generics, but not for all cases.

33

- 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.

Implementing a generic interface – Part 1

- Many interfaces of the JDK have been upgraded to be generics, e.g. *Comparable<T>*.
  - In past we implemented *Comparable* on the class *GasCar*. – Let's upgrade *GasCar* to implement *Comparable<GasCar>*:

```
// <GasCar.java>
public class GasCar extends Car implements Comparable {
	private double fuelCapacity;

	public GasCar(double fuelCapacity) {
		this.fuelCapacity = fuelCapacity;
	}
	@Override
	public int compareTo(Object otherCar) {
		double fuelCapacityOfOtherCar = ((GasCar) otherCar).fuelCapacity;
		if (this.fuelCapacity < fuelCapacityOfOtherCar) {
			return -1;
		} else if (this.fuelCapacity > fuelCapacityOfOtherCar) {
			return 1;
		} else {
			return 0;
		}
	}
}
```

```
// <GasCar.java>
public class GasCar extends Car implements Comparable<GasCar> { // (1)
	private double fuelCapacity;

	public GasCar(double fuelCapacity) {
		this.fuelCapacity = fuelCapacity;
	}
	@Override
	public int compareTo(GasCar otherCar) { // (2)
		double fuelCapacityOfOtherCar = otherCar.fuelCapacity; // (3)
		if (this.fuelCapacity < fuelCapacityOfOtherCar) {
			return -1;
		} else if (this.fuelCapacity > fuelCapacityOfOtherCar) {
			return 1;
		} else {
			return 0;
		}
	}
}
```

  - (1) When implementing *Comparable* "the generic way", we have to write a strange looking "*implements Comparable<GasCar>*".
  - (2) *Comparable<GasCar>* mandates to *@Override* the method *compareTo(GasCar)*, i.e. not just *compareTo(Object)*.
  - (3) Because *compareTo()*'s parameter is of type *GasCar* instead of the more general *Object*, we need no downcast!
  - => Esp. (2) and (3) lead to much better type safety at compile time and more comfort for programmers.

34

Implementing a generic interface – Part 2

- If we implement *Comparable<GasCar>* in favor to the *Comparable*, we get compile time type safety as just mentioned:

```
GasCar lhs = new GasCar(40);
Car rhs = new Car();

// Invalid! Incompatible types: Car cannot be converted to GasCar:
int result = lhs.compareTo(rhs);
```

```
GasCar lhs = new GasCar(40);
GasCar rhs = new GasCar(45.7);

// OK!
int result = lhs.compareTo(rhs);
```

Excursus: Simple Comparison for primitive Types

- Implementing *Comparable<GasCar>* means to *@Override Comparable<GasCar>.compareTo()*.
  - Overriding *Comparable<GasCar>.compareTo()* is not complicated, but requires a certain amount of code.
  - Different negative or positive values or 0 must be returned, depending on the relative order we want to express:

```
@Override
public int compareTo(GasCar otherCar) {
        double fuelCapacityOfOtherCar = otherCar.fuelCapacity;
        if (this.fuelCapacity < fuelCapacityOfOtherCar) {
                return -1;
        } else if (this.fuelCapacity > fuelCapacityOfOtherCar) {
                return 1;
        } else {
                return 0;
        }
}
```

  - Mind, that we need a lot of code only to express equivalence using one field, namely *GasCar.fuelCapacity*.

- Luckily, Java provides helper methods in the companion classes for primitive numeric types, e.g. *Double* and *Integer*.
  - E.g. the helper method *Double.compare()* allows expressing *Comparable<GasCar>.compareTo()* much simpler:

```
@Override
public int compareTo(GasCar otherCar) {
        return Double.compare(this.fuelCapacity, otherCar.fuelCapacity);
}
```

  - Of course *Double.compare()* can also be used independently from generics or overrides of *Comparable.compareTo()*.

Where have we used Type Parameters up to here?

- Alas, TE puts a lot of serious restrictions on how we can use type parameters in method implementations.
  - Mind, that we only have discussed type parameters used in the interface of a generic: as fields, method parameter- and return-types.

- We used type parameters only:
  - (1) To parameterize field types
    - (1.1) Also as array field
    - (1.2) Also as field of generic type using the type parameter as type argument
  - (2) To parameterize methods parameter- and return-types.
  - (3) To parameterize (1) and (2) in nested classes.

- Mind, that esp. (1) and (2) are already sufficient to implement generic collections such as *ArrayList<T>* and *HashMap<T>*.

- But, now it's time to discuss some restrictions.

```
// <Generic.java>
public class Generic<T> {
	private T field; // (1) "scalar" field
	private T[] arrayAsField; // (1.1) array field
	private List<T> genericField; // (1.2) field of generic type with T type argument

	public T get() { // (2) return type
		return this.field;
	}

	public void set(T value) { // (2) parameter type
		this.field = value;
	}

	public class NestedClass {
		private T nestedField; // (3.1) field in nested class method

		public T get() { // (3.2) return type in nested class method
			return this.nestedField;
		}

		public void set(T value) { // (3.2) parameter type in nested class
			this.nestedField = value;
		}
	}
}
```

37

Restrictions – Part 1

- The major restriction is on the operations we can do on objects of the type parameter: we can only use it like *Object*!
  - We can only call methods derived from *Object*.
  - We can only do assignments, reference comparisons and pass them to other methods.
  - Because of TE, we only know, that it must be an *Object* at least.

```
// <Generic.java>
public class Generic<T>  {
        private final T field;

        public void accept(T t) {
            // pass
        }

        public Generic(T t) {
            this.field = t; // OK! Assign reference of parameter type to another reference.
            if (null == t) { // OK! Compare reference of parameter type.
                throw new IllegalArgumentException("Argument may not be null");
            }

            accept(t); // OK! Pass reference of parameter type to a method.

            System.out.println("As String: "+t); // OK! Use operator + on a reference of parameter type to another reference.
            System.out.println("Also as String: "+t.toString()); // OK! Call Object-methods on a reference of parameter.
        }
}
```


38

Restrictions – Part 2

- A painful restriction is, that we can neither call ctors nor any arbitrary method on references of the parameter type:

```
// <InvalidGeneric.java>
public class InvalidGeneric<T>  {
        public InvalidGeneric() {
            T t = new T(); // Invalid! Cannot call ctor of the parameter type!
            t.aMethod();  // Invalid! Cannot call method not derived from Object on a reference of parameter type
        }
}
```

```
// <MyClass.java>
public class MyClass  {
        public void aMethod() {
            // pass
        }
}
```

```
// InvalidGeneric with MyClass as type argument:
InvalidGeneric<MyClass> instance = new InvalidGeneric<>();
```

- The generic type cannot access the interface of the type argument, therefor we can't call arbitrary methods.
  - Although we know, that *MyClass.aMethod()* exists, it can't be called in the generic type, because the generic doesn't know it (TE).

- The generic type cannot create instances of the type parameter: due to TE the type argument is unknown.

Restrictions – Part 3

- Similar to the restriction that we cannot create new instances from the type parameter, we cannot create arrays thereof:

```
// <Generic.java>
public class Generic<T> {
        private T[] arrayAsField;

        public void createArray(int capacity) {
              arrayAsField = new T[capacity]; // Illegal!
        }
}
```

- This restriction can be circumvented by creating an Object[] the "ordinary way" and cast it to the desired array-type:

```
// <Generic.java>
public class Generic<T> {
        private T[] arrayAsField;

        public void createArray(int capacity) {
              arrayAsField = (T[]) new Object[capacity]; // OK!
        }
}
```

Type Bounds

- When methods on references of the type parameter must be called, a bound must be put on the type parameter:

```
// <BoundGeneric.java>
public class BoundGeneric<T extends MyClass> {
        public BoundGeneric(T t) {
            t.aMethod();  // OK!
        }
}
```

- So, Java allows to put bounds on type parameters in the type parameter list to make them bounded type parameters.
  - The bound notated with the contextual extends keyword is an upper bound, the type parameter is now bounded.

```
BoundGeneric<MyClass> instance = new BoundGeneric<>(new MyClass());
```

- Only methods of the upper bound's type can be called on references of the type parameter. There are still restrictions:
  - We still cannot call ctors to create new instances of the type parameter. – Due to TE the actual type is unknown.
    - This is usually no problem for users of generics (they can create new instances), but implementors: They can't create instance of type argument.
  - We cannot use operators on references of the type parameter, except '+' to call *Object.toString()* or an *@Override* thereof.
    - This is bad, because we cannot call operators on numeric types (but this is also the case, because they are value types).

- It is called upper bound, because the type argument can be equal to or lower in the type hierarchy than the type bound.
  - E.g. *MyClass.aMethod()* is safe to called in *BoundGeneric*'s dctor for *MyClass* instances or instances of more special types.

- => Type bounds restrict the types/type arguments we can use for the sake of type safety!

Generic Methods – Part 1

- Let's ask the other way around: When and why is it required to call methods on type parameter references?

- The answer is, strictly speaking from a generic standpoint: We must call methods, if we implement algorithms! Consider:
  - Partitioning splits a *List* into two *List*s, of which one contains larger elements, the other one smaller elements seen from a pivot.

```
public class Algorithms<T extends Comparable<T>> {
        public Pair<List<T>, List<T>> partition(List<T> in, T pivot) {
                Pair<List<T>, List<T>> out = new Pair<>();
                List<T> lessThanOrEqual = new ArrayList<>();
                List<T> greaterThan = new ArrayList<>();
                for (T item : in) {
                        if (0 >= item.compareTo(pivot)) {
                                lessThanOrEqual.add(item);
                        } else {
                                greaterThan.add(item);
                        }
                }
                out.setFirst(lessThanOrEqual);
                out.setSecond(greaterThan);
                return out;
        }
}
```

  - For *Algorithms.partition()* to work, the input *List*'s items must be compared.
  - To enable this algorithm to call *Comparable.compareTo()*, we must set an upper bound to *T*, which constraints to *Comparable<T>*.
  - Another takeaway: despite the upper bound is defined with the extends keyword, we can also use it with interfaces. 42
  - Another restriction should be mentioned here: we cannot call/access static methods/fields although the constraint was set.

Generic Methods – Part 2

- If we want to use *Algorithms.partition()*, we have to create an instance of *Algorithms* and pass the arguments:

```
List<Integer> input = Arrays.asList(15, 97, 52, 29, 7, 92, 12);
Algorithms<Integer> algorithms = new Algorithms<>();
Pair<List<Integer>, List<Integer>> partitions = algorithms.partition(input, 56);
```

- The need to create an *Algorithms* object only to call *partition()* is virtually not required, why not make the method static:

```
public class Algorithms<T extends Comparable> {
        public static Pair<List<T>, List<T>> partition(List<T> in, T pivot) {
            // Invalid! Non-static type variable T cannot be referenced from a static context
            // pass
        }
}
```

  - It doesn't work: due to TE Java disallows static methods and fields to refer the type parameter T of the enclosing class or interface.

- Instead we can make *Algorithms.partition()* itself a generic method:

```
public class Algorithms {
        public static <T extends Comparable> Pair<List<T>, List<T>> partition(List<T> in, T pivot) {
            // pass
        }
}
```

  - Yes, the syntax is strange, we have to put the list of type parameters between the method's modifiers and the return type/void.
  - Also instance methods and ctors can be generic! They can even be itself generic and use generic parameters of the enclosing type.

- Now we can call *Algorithms.partition()* as a static method.

Generic Methods – Part 3

- Generic methods enable an important feature for Java's algorithms: type inference. Consider the code we've just written:

```
Pair<List<Integer>, List<Integer>> partitions = Algorithms.<Integer>partition(input, 56);
```

  - Type inference allows us to write this instead:

```
Pair<List<Integer>, List<Integer>> partitions = Algorithms.partition(input, 56);
```

  - It isn't needed to explicitly write type args when calling generic methods, if types can be undoubtedly inferred from the method args.
  - Actually, we have already used this form of type inference when calling *Arrays.asList()*. Originally, it is defined as:

```
// Somewhere in the JDK:
public class Arrays {
        public static <T> List<T> asList(T... array) { // (details hidden)
            return new InternalArrayList<>(array);
        }
}
```

  - So, we can write either form:

```
List<Integer> input = Arrays.<Integer>asList(15, 97, 52, 29, 7, 92, 12);
```

```
List<Integer> input = Arrays.asList(15, 97, 52, 29, 7, 92, 12);
```

- If a method returns a generic and has params depending on type params, type args can be inferred from the return type:
  - A classic example is calling *Arrays.asList()* with objects of mixed types: Which type argument should the resulting *List* have?
  - Here, the type argument *Object* makes sense to be inferred, because *Object* is a common super type of *String*, *Integer* and *Boolean*:

```
List<Object> input = Arrays.asList("Hello", 97, 52, 29, false, 92, 12);
```

  - This form of control is excellent, because we can explicitly select another common type, e.g. an interface:

```
List<Comparable> input = Arrays.asList("Hello", 97, 52, 29, false, 92, 12);
```


44

Generic Methods – Part 4

- A generic type parameter can be specified with more than one bound:

```
// <Car.java>
public class Car { // (members hidden)
        public void setSpareTyre(Tyre sTyre) {
                this.spareTyre = sTyre;
        }
        public Tyre getSpareTyre() {
                return this.spareTyre;
        }
}
```

```
// <Algorithms.java>
public class Algorithms {
        public static <T extends Car & Comparable<T>> Pair<List<T>, List<T>> partitionBySpareTyrePivot(List<T> in, T pivot) {
                Pair<List<T>, List<T>> out = new Pair<>();
                List<T> lessThanOrEqual = new ArrayList<>();
                List<T> greaterThan = new ArrayList<>();
                for (T car : in) {
                        if (null == car.getSpareTyre() && 0 >= car.compareTo(pivot) ) {
                                lessThanOrEqual.add(car);
                        } else {
                                greaterThan.add(car);
                        }
                }
                out.setFirst(lessThanOrEqual);
                out.setSecond(greaterThan);
                return out;
        }
}
```

- As can be seen, multiple type bounds for specific type parameters are chained with &-symbols.
  - Mind, that interfaces must be listed after a class, other order-details of the list don't matter to Java.
  - However, the class *GasCar* is a perfect fit for the bound *"T extends Car & Comparable<T>"*:

```
GasCar gasCar1 = new GasCar(45);
gasCar1.setSpareTyre(new Tyre());
List<GasCar> gasCars = Arrays.asList(gasCar1, new GasCar(33), new GasCar(86));
Pair<List<GasCar>, List<GasCar>> partitions = Algorithms.partitionBySpareTyrePivot(gasCars, new GasCar(50));
```

  - In "isolation" a type declared as "combination" of other types like "*Car & Comparable*" is called intersection type.

45

- There is one restriction on the bounds we can specify: the bounds are not allowed to list multiple "instances" of the same generic interface. Following definition is invalid:

```
// <SomeType.java>
public class SomeType<T extends Comparable<String> & Comparable<StringBuffer>> {
        // pass
}
```

- Intersection types also play a role in more advanced applications in Java, which are beyond this course.

Excursus: parameterized Polymorphism – Part 1

- Up to now, we discussed two forms of polymorphism:
  - Alongside with procedural programming we discussed overloaded methods, which enable a kind of compile time polymorphism.
  - Alongside with object oriented programming we discussed overridden methods, which enable run time polymorphism.
- This presentation talks about generic programming, which enable parametrized polymorphism at compile time.
- Generic programming is about expressing algorithms to work for multiple datatypes.
  - What is the difference to other forms of polymorphism? – Mainly type inference.
  - Generics leave a part of the type open, accessible by a type parameter, that must be filled by a type argument at compile time.
  - Type inference allows to infer type arguments only from the used expressions.
- Generics have many applications in Java:
  - The Java Collections Framework (JCF) with generic type likes *Collection<E>*, *List<E>*, *ArrayList<E>*, *Map<K, V>* and *HashMap<K, V>*.
  - Generic methods in companion classes such as *Arrays* and *Collections*.
  - Meaningful infrastructure types like *Class<T>*, *Comparator<T>*, *Iterable<T>* and *Iterator<T>*.
  - Java's support for functional programming with types like *Function<T, R>*, *Predicate<T>*, *Supplier<T>*, *Consumer<T>* and *Optional<T>*.
  - Java's support for functional programming with *Streams*, mainly featuring the types *Stream<T>* and *Collector<T, A, R>*.

46

- Java is basically a statically typed language. Esp. in statically typed language it is difficult to write UDTs or methods which handle different types of arguments. We have to use any kind of polymorphism to make it work.

Excursus: parameterized Polymorphism – Part 2

- But … with run time polymorphism I can also make my code working with other types! – Where is the difference?
- Consider the method *mostDrivenCar()*, that returns the *Car* which drove the longest distance from the passed *List*:

```
public static Car mostDrivenCar(List cars) {
        return Collections.max(cars, new CarDrivenDistanceComparator());
}
```

  - We can call the method with a *List<Bus>*:

```
List<Bus> busses = Arrays.asList(new Bus(), new Bus(), new Bus());
busses.get(0).drive(15_000);
busses.get(1).drive(5_000);
busses.get(2).drive(2_500);
Bus mostDrivenBus = (Bus)mostDrivenCar(busses); // OK, but a risky downcast is required!
```

  - Sure, this works, but *mostDrivenCar()* returns only a *Car*, we have to cast it down to the *Bus* we want to work with.
- The risk of a run time polymorphism-based algorithm is, that often we have to leave the ground of type safety!
  - The reason is simple: run time polymorphism means to take a decision about a common most general type, which is still sufficient!
  - The invention of interfaces made this more versatile: types can have more common denominators apart from inheritance!
- Without further ado, when we make *mostDrivenCar()* generic, it'll be much type safer for callers:

```
public static <T extends Car> T mostDrivenCar(List<T> cars) {
        return Collections.max(cars, new CarDrivenDistanceComparator());
}
```

```
Bus mostDrivenBus = mostDrivenCar(busses); // Excellent, no cast required!
```

47

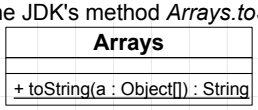
- We can even say, that the idea of interfaces is only to have the same type being able to take part in different run time polymorphism-based algorithms, because it has many common denominators with other types to offer.

Looking back at Array Covariance

- When we discussed arrays containing instances of polymorphic types, also array covariance was presented. Consider:

```
Car[] cars = { new Car(), new Car(), new Car() };
Vehicle[] vehicles = cars; // OK! Arrays are covariant in Java.
```

- The JDK's method *Arrays.toString()* perfectly exploits array covariance. Consider:




```
String carArrayStringRepresentation = Arrays.toString(cars);
System.out.println(carArrayStringRepresentation);
// >[vehicleID: W0L000051T2123456, vehicleID: W0L000041G2153456, vehicleID: F0Z000030H2153972] // Ok!
```

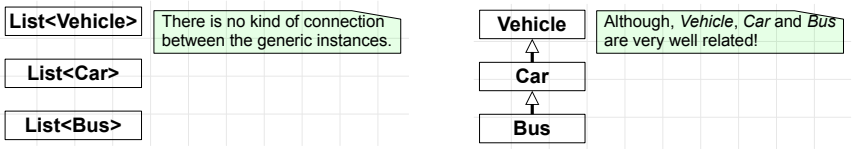
- Once again remember the deeper truth is the special substitutability of arrays in Java known as array covariance:
  - Java assumes that if *B* can be a substitute of *A*, also *B[]* can be a substitute of *A[]*. This is called covariance.
    - *Car* can be a substitute for *Object*, hence *Car[]* can substitute *Object[]*.
  - Although covariance of arrays is intuitive, it can become a problem, when code needs to set elements the array.
    - Mind that *Arrays.toString()* only needs to read elements in the array, which is no problem!

Generic Variance – Wildcards – Part 1

- On arrays, we have covariance, but on generic collections, such as *List*, covariance is not implied:

```
List<Car> cars = new ArrayList<>(Arrays.asList(new Car(), new Car(), new Car()));
List<Vehicle> vehicles = cars; // Invalid! Incompatible types: List<Car> cannot be converted to List<Vehicle>
```

  - From an oo-standpoint this makes sense: *List<Car>* and *List<Vehicle>* are just no related types, esp. no substitutability is given!



- Java allows to get a level of variance of generics against the type *Object* with so called unbound wildcards:

```
List<?> vehicles = cars; // OK!
```

  - Syntactically, we use the ?-wildcard instead of a specific type argument.

- On the following slides, we will discus Java's (rather complex) idiom of wildcards on generic references.

Generic Variance – Wildcards – Part 2

- Unbounded wildcards are useful for algorithms, that should be able to work with generics, but not with the parameter types.

```
List<Vehicle> vehicles = new ArrayList<>(Arrays.asList(new Vehicle(), new Vehicle(), new Vehicle()));
List<Car> cars = new ArrayList<>(Arrays.asList(new Car(), new Car(), new Car()));
List<Bus> busses = new ArrayList<>(Arrays.asList(new Bus(), new Bus(), new Bus()));
List strings = new ArrayList(Arrays.asList("A", "B", "C"));
```

- Consider this code, which uses the raw type *List* for the parameter, in this case we could pass any *List<T>* or a raw *List*:

```
public static boolean hasEvenCountRaw(List items) {
        return 0 == items.size() % 2;
}
```

  - We can pass any kind of *List*, it can be a generic one or a *List* of raw type:

```
boolean result1 = hasEvenCountRaw(vehicles);
boolean result2 = hasEvenCountRaw(cars);
boolean result3 = hasEvenCountRaw(busses);
boolean result4 = hasEvenCountRaw(strings);
```

- Using raw types instead of generics allows us to use the *List* covariantly, but with all the risks, consider:

```
public static void addCar(int index, List items) {
        items.add(index, new Car());
}
```

  - Due to TE, *addCar()* also accepts a *List<Bus>*, but, due to the lack of type safety, a new *Car* instance will be added into *items*:

```
addCar(0, busses);
```

  - But the caller of *addCar()*, does still assume *Bus*ses in the *List<Bus>*! This is still OK for the compiler, but fails at run time:

```
for (Bus bus : busses){ // ClassCastException: Car cannot be cast to Bus
        System.out.println(bus);
}
```


50

Generic Variance – Wildcards – Part 3

- Now we use *List<?>* as parameter type, then we can also pass any *List<T>* and raw *List* as argument:

```
public static boolean hasEvenCountWildcard(List<?> items) {
        return 0 == items.size() % 2;
}
```

```
boolean result1 = hasEvenCountWildcard(vehicles);
boolean result2 = hasEvenCountWildcard(cars);
boolean result3 = hasEvenCountWildcard(busses);
boolean result4 = hasEvenCountWildcard(string);
```

- Using unbounded wildcard in favor of raw types disallows potentially dangerous operations on the generic reference:

```
public static void addCar(int index, List<?> items) {
        items.add(index, new Car()); // Invalid! Incompatible types: Car cannot be converted to capture#1 of ?
}
```

  - The (wildcard-) captured type is the type, that replaces ? at compile time: in case of *vehicles*, the captured type is *Vehicle*.

- On an unbounded wildcard generic reference we can:

```
List<?> items = cars;
```

  - (1) Always call methods, which are independent of the captured type.

```
int itemCount = items.size();  // (1)
```

  - (2) Call methods, which return the captured type, but only store the result in an *Object*.

```
Object item1 = items.get(0);  // (2)
```

    - This variance is safe: the captured type must at least be an *Object*, because we can only hold reference types in generics.
  - (3) Call methods having the captured type as parameter type, but only pass null to these parameters.

```
items.add(0, null); // (3)
```

    - Variance is safe: we can only hold reference types in generics, and null is the least common denominator.

51

List<?> is no List<Object>

- An important take away is that a *List<?>* is not a *List<Object>* and it is no raw type *List*, consider following examples.
  - If we have a *List<Object>*, we can only pass arguments implementing *List<Object>*:

```
public static void acceptObjectList(List<Object> items) {
        // pass
}
```

```
acceptObjectList(new ArrayList<Object>());
```

```
acceptObjectList(new ArrayList<String>());
```

Some phrases, which may help:
- *List<?>* is a short form of *List<? extends Object>*.
- *List<?>* means a *List* of any class.
- *List<Object>* means one a *List* of only *Objects*.

  - If we have a raw *List*, we can pass arguments implementing any *List*:

```
public static void acceptRawList(List items) {
        // We could add instances of unexpected types into items.
}
```

```
acceptRawList(new ArrayList<Object>());
acceptRawList(new ArrayList<String>());
```

    - The problem is, however, that in *acceptRawList()* we could add objects of unexpected type into *items* at run time.
  - If we have a *List<?>*, we can pass all arguments implementing *List<T>*:

```
public static void acceptUnboundedWildcardList(List<?> items) {
        // pass
}
```

```
acceptUnboundedWildcardList(new ArrayList<Object>());
acceptUnboundedWildcardList(new ArrayList<String>());
```

    - But this time, the compiler stops us from programming type-unsafe operations within *acceptUnboundedWildcardList()*!

52

- In a sense *List<?>* and *List<Object>* are extremes, but Java allows using generics between these extremes.

Generic Covariance – Part 1

- Remember, that when we discussed covariant arrays, there also was a problem with type safety, consider:

```
public static void addVehicle(int index, Vehicle[] vehicles) { // Yes, this methods looks harmless!
        vehicles[index] = new Vehicle();
}
```

```
Car[] cars = { new Car(), new Car(), new Car() };
// OK! Arrays are covariant in Java:
addVehicle(0, cars); // java.lang.ArrayStoreException: Vehicle
```

- By default, generics are not covariant, so the code above expressed with *List<T>* instead of arrays won't compile at all:

```
public static void addVehicle(int index, List<Vehicle> vehicles) { // OK! just using List<Vehicle> now!
        vehicles.add(index, new Vehicle());
}
```

```
List<Car> cars = new ArrayList<>(Arrays.asList(new Car(), new Car(), new Car()));
addVehicle(0, cars); // Invalid! Incompatible types: List<Car> cannot be converted to List<Vehicle>
```

- When we change the parameter *vehicles* to an upper bound wildcard, calling *addVehicle()* will successfully compile:

```
public static void addVehicle(int index, List<? extends Vehicle> vehicles) {
        vehicles.add(index, new Vehicle());    // Invalid! Incompatible types: Vehicle cannot be
}                                              // converted to capture#1 of ? extends Vehicle
```

```
List<Car> cars = new ArrayList<>(Arrays.asList(new Car(), new Car(), new Car()));
addVehicle(0, cars); // OK! We can pass List<Car> to List<? extends Vehicle>
```

  - But the definition of *addVehicle()* will no longer compile!

- In a sense, using generics with upper bound wildcards turns "potential *ArrayStoreExceptions*" into a compile time errors.

Generic Covariance – Part 2

- In fact, the Java designer team learned from the problems with covariant arrays and designed generics accordingly.
- (1) By default, generics are not covariant:

```
Car[] cars = { new Car(), new Car(), new Car() };
Vehicle[] vehicles = cars;
```

```
List<Car> cars = new ArrayList<>(Arrays.asList(new Car(), new Car(), new Car()));
List<Vehicle> vehicles = cars;
```

- (2) We have to explicitly declare at the reference type (variables or esp. parameters), if it should accept a covariant type:

```
List<? extends Vehicle> vehicles = cars;
```

- (3) On upper bound wildcard references, we can safely call methods, that return an instance of the captured type:

```
Vehicle vehicle1 = vehicles.get(0); // Safe!
```

  - Assigning the returned value is safe, because *vehicles.get(0)* does for sure return an object, of a substitute type of *Vehicle*.
  - Therefor we call it the upper bound wildcard: must at least have *Vehicle* as "upper" type in the type hierarchy.

- (4) On upper bound wildcard references, we cannot call methods, that have the captured type as parameter:

```
vehicles.add(5, new Car());      // (a) Invalid! (The run time type of vehicles is List<Car>!)
vehicles.add(6, new Bus());      // (b) Invalid!
vehicles.add(4, new Vehicle());  // (c) Invalid!
```

  - Arguments of captured types or its sub types can't be safely passed: run time types of captured types can be more special than the arguments' types.
  - In other words: the argument types are not guaranteed to be substitute types of the run time types of the captured types.
  - Therefor passing instances to parameters of the captured type is strictly forbidden by the compiler in this case. 54
  - Java always allows to pass null:

```
vehicles.add(5, null); // OK! Always safe!
```

Generic Contravariance – Part 1

- There is more to variance. With generics, Java 5 also introduced contravariance.
  - Contravariance is slightly more difficult to understand than covariance.

- Let's assume we have the method *addBus()*, that adds a *Bus* to a *List<Bus>*:

```
public static void addBus(int index, List<Bus> busses) {
        busses.add(index, new Bus());
}
```

- By default, generics are not contravariant, so the code above expressed with *List<T>* instead of arrays won't compile at all:

```
List<Vehicle> vehicles = new ArrayList<>(Arrays.asList(new Vehicle(), new Vehicle(), new Vehicle()));
addBus(0, vehicles); // Invalid! Incompatible types: List<Vehicle> cannot be converted to List<Bus>
```

- Contravariance would mean, that a reference of a specific type could be substituted by a reference of a more general type.

- Hm, why can contravariance be of value? – Accepting a more general type, were a specific type is awaited sounds weird!
  - If we concentrate on the statement `busses.add(index, new Bus());` the question can be: which types can be used for *busses*?
  - Ideally, the answer of this question will be: a *List*, which accepts a type *Bus* or something being more general than *Bus*.
  - Why is this ideal? Because in such a case, we can pass any *List* holding more general types than *Bus* to *addBus()* as type argument.
  - And why is this ideal? – In *addBus()* we can then add *Bus*ses safely, because *busses* accepts at least *Bus*ses or more general types.
  - … i.e. we can safely add a *Bus* to a *List<Vehicle>* or *List<Car>* or *List<Bus>*.

Generic Contravariance – Part 2

- When we change the parameter of *addBus()* to a lower bound wildcard, calling *addBus()* will successfully compile:

```
public static void addBus(int index, List<? super Bus> busses) { // OK! just using List<Vehicle> now!
        busses.add(index, new Bus());
}
```

  - We can pass *List<Vehicle>*, *List<Car>* or *List<Bus>* to *addBus()*:

```
List<Vehicle> vehicles = new ArrayList<>(Arrays.asList(new Vehicle(), new Vehicle(), new Vehicle()));
addBus(0, vehicles); // OK!
List<Car> cars = new ArrayList<>(Arrays.asList(new Car(), new Car(), new Car()));
addBus(0, cars); // OK
List<Bus> busses = new ArrayList<>(Arrays.asList(new Bus(), new Bus(), new Bus()));
addBus(0, busses); // OK
```

  - The calls work, because *List<Vehicle>*, *List<Car>* and *List<Bus>* are contravariant in their parameter types to *List<? super Bus>*.

- Lower bound wildcard references also add restrictions: we cannot call methods, that return the captured type:

```
List<? super Bus> vehiclesAsBusses = busses;
Vehicle vehicle1 = vehiclesAsBusses.get(0); // Invalid! Incompatible types: capture#1 of ? super Bus cannot be converted to Vehicle
Car car1 = vehiclesAsBusses.get(0); // Invalid! Incompatible types: capture#1 of ? super Bus cannot be converted to Car
Bus bus1 = vehiclesAsBusses.get(0); // Invalid! Incompatible types: capture#1 of ? super Bus cannot be converted to Bus
```

  - Args of captured types or its sub types can't be safely returned: run time types of captured types can be more general than the arguments' types.
  - In other words: the argument types are not guaranteed to be substitute types of the run time types of the captured types.
  - Therefor assigning returned instances of the captured type is forbidden by the compiler in this case.

- Java always allows to assign the returned value to *Object*:

```
Object object = vehiclesAsBusses.get(0); // OK! Always safe!
```

Where to use Generic Variance – Part 1

- So, Java allows to hold instances of generic types in covariant or contravariant references in a type safe manner.
  - Java's generic types are invariant. But we can define upper and lower bound wildcards to get variances on generic references.

- Let's go with these *List*s:

```
List<Vehicle> vehicles = new ArrayList<>(Arrays.asList(new Vehicle(), new Vehicle(), new Vehicle()));
List<Car> cars = new ArrayList<>(Arrays.asList(new Car(), new Car(), new Car()));
List<Bus> busses = new ArrayList<>(Arrays.asList(new Bus(), new Bus(), new Bus()));
```

- Upper bound wildcards allow to hold a generic's instance, that is covariant on captured return types of its methods.
  - Syntactically, we use the ?-wildcard, the contextual extends keyword and then the upper bound type argument:

```
List<? extends Vehicle> covariantVehicles = vehicles;
Vehicle vehicle1 = covariantVehicles.get(0); // Safe!
covariantVehicles = cars;
Vehicle vehicle2 = covariantVehicles.get(0);
covariantVehicles = busses;
Vehicle vehicle3 = covariantVehicles.get(0);
```

- We cannot call methods, that have the captured type as parameter type passing a bounded type's reference.

```
vehicles.add(4, new Vehicle());  // Invalid! No suitable method found for add(Car) method add(capture#1 of ? extends Vehicle)
                                 // is not applicable (argument mismatch; Car cannot be converted to capture#1 of ? extends Vehicle)
```

  - The rationale and safety rule is clear: setting of the captured type is not safe (lesson learned from arrays) but getting is safe.
  - When a generic reference has an upper bound to a wildcard, we can only get values of the parameter type, but never set them.
  - The honorable exception is null: passing null is always type safe: `vehicles.add(4, null); // OK!`

Where to use Generic Variance – Part 2

- Lower bound wildcards allow to hold an instance, that is contravariant on captured parameter types of its methods.
  - Syntactically, we use the ?-wildcard, the contextual super keyword and then the lower bound type argument:

```
List<? super Bus> contravariantBusses = vehicles; // OK! Vehicle is more general than Bus, and busses is contravariant on Bus.
contravariantBusses.add(4, new Bus()); // OK! It's type safe, because the captured type Vehicle is more general than Bus.
contravariantBusses = cars; // OK! Car is more general than Bus, and busses is contravariant on Bus.
contravariantBusses.add(4, new Bus()); // OK! It's type safe, because the captured type Car is more general than Bus.
contravariantBusses = busses; // OK! busses is contravariant on Bus.
contravariantBusses.add(4, new Bus()); // OK! It's type safe, because the captured type is Bus.
```

- We cannot call methods, that have the captured type as return type assigning to a bounded type's reference.

```
Vehicle vehicle4 = contravariantBusses.get(0); // Invalid.
```

  - The rationale and safety rule is this: setting of the captured type is safe (lesson learned on arrays) but getting is not safe.
  - When a generic reference has a lower bound to a wildcard, we can only set values of the parameter type, but never get them.
  - The honorable exception is *Object*: if the assigned-to reference type of the got value is *Object*, this is type safe:

```
Object object = contravariantBusses.get(0); // OK!
```

  - Actually, we can override all the type safety we got using contravariance and just make an explicit downcast:

```
Vehicle vehicle5 = (Vehicle) contravariantBusses.get(0); // OK!
```

- Wildcard bounds allow writing type safe, reusable algorithms, but can be tricky to use.

The Type Class<T> reviewed with Covariance – Part 1

- When we discussed dynamic type checks, we also came across the method *Object.getClass()*.
  - We can call *Object.getClass()* to get type information about the dynamic type of an object:

```
Car car = new Bus();
Class dynamicType = car.getClass();
System.out.println(dynamicType);
// > class Bus
```

- Actually, the class *Class* is a generic class and *Object.getClass()* returns a *Class*-object with an unbounded wildcard:

```
// Somewhere in the JDK:
public class Class { // (declaration simplified)
}
```

actually

```
// Somewhere in the JDK:
public final class Class<T> { // (declaration simplified)
}
```

```
// Somewhere in the JDK:
public class Object { // (declaration simplified)
        public Class getClass() { /* pass */ }
}
```

actually

```
// Somewhere in the JDK:
public class Object { // (declaration simplified)
        public final Class<?> getClass() { /* pass */ }
}
```

- When we program new classes like *Car*, Java implies the presence of a method with following return type:

```
// <Car.java>
public class Car { // (members hidden)
        public final Class<? extends Car> getClass() { /* pass */ }
}
```

59

The Type Class<T> reviewed with Covariance – Part 2

- When we inherit the class *Bus* from *Car*, the implied method *Bus.getClass()* has a respective "evolving" return type:

```
// <Car.java>
public class Car { // (members hidden)
        public final Class<? extends Car> getClass() { /* pass */ }
}
```

```
// <Bus.java>
public class Bus extends Car { // (members hidden)
        public final Class<? extends Bus> getClass() { /* pass */ }
}
```

- The basic idea behind *Class<?>* being sufficient in most cases is simple:
  - Most algorithms are not interested in the captured type, but only in the metadata of the captured type.
  - *Class<?>* is often sufficient, because most of the methods in *Class<T>* do not depend on T.

- A notable exception to the statement "unbound wildcard is sufficient" is the method *Class.newInstance()*:

```
// Somewhere in the JDK:
public final class Class<T> { // (declaration simplified)
        public T newInstance() { /* pass */ }
}
```

```
Bus bus = new Bus();
Class<? extends Car> busClass = bus.getClass();
Car car = busClass.newInstance();
System.out.println(car.getClass());
// >class Bus
```

  - If we use an upper-bound wildcard, we exploit the covariant return type of *Bus.getClass()*.
  - Calling *newInstance()* on a *Class<? extends Car>* safely returns a *Car*, but it could be more special (it is a *Bus* in this case).

The Comparator Interface – Part 1

- In one of the prior slides, we implemented *Comparable<GasCar>* in *GasCar* comparing *GasCar*s against their *fuelCapacity*.
  - What we can do with this is sorting a *GasCar[]* in the order of the contained *GasCar*s' *fuelCapacity*s:

```
GasCar[] gasCars = new GasCar[3];
gasCars[0] = new GasCar(45);
gasCars[1] = new GasCar(33);
gasCars[2] = new GasCar(86);
Arrays.sort(gasCars);
// gasCars = {GasCar(fuelCapacity = 33), GasCar(fuelCapacity = 45), GasCar(fuelCapacity = 86)}
```

```
@Override
public int compareTo(GasCar otherCar) {
        return Double.compare(this.fuelCapacity, otherCar.fuelCapacity);
}
```

- But … what, if we want to sort *gasCars* after another criterion, e.g. the *GasCar*s' driven distance in km?
  - We assume, that *GasCar*'s super class *Car* provides *Car.drive()* to increase the driven distance and *Car.getDrivenDistance()*:

```
// <Car.java>
public class Car {
        private double drivenkm;

        public void drive(double distancekm) {
                this.drivenkm += distancekm;
        }
        public double getDrivenDistance() {
                return this.drivenkm;
        }
}
```

  - Of course, we could modify the code of *GasCar.compareTo()* to compare driven distances, but it might break others' code!
  - And we'd hurt the OCP, which tells us not to modify present classes (but rather to extend them).

61

The Comparator Interface – Part 2

- As you might have guessed, Java provides a way to define alternative ways to compare objects of predefined UDTs.

- The JDK provides another interface for this, namely *Comparator*:

```
// (somewhere in the JDK)
public interface Comparator<T> {
        int compare(T o1, T o2);
}
```

  - The idea is, that we have to program a class, which offers a specific way to compare *T*'s by implementing *Comparator<T>*.
  - Let's implement the *CarDrivenDistanceComparator*:

```
// <CarDrivenDistanceComparator.java>
public class CarDrivenDistanceComparator implements Comparator<Car> {
        @Override
        public int compare(Car lhs, Car rhs) {
                return Double.compare(lhs.getDrivenDistance(), rhs.getDrivenDistance());
        }
}
```

- *Arrays.sort()* provides a generic overload, that accepts an arrays and a *Comparator<T>*:

```
// (somewhere in the JDK)
public class Arrays {
        public static <T> void sort(T[] a, Comparator<? super T> c) {
                // pass
        }
}
```


62

The Comparator Interface – Part 3

- Now its time to bring *CarDrivenDistanceComparator* and *Arrays.sort()* together:

```
GasCar[] gasCars = new GasCar[3];
gasCars[0] = new GasCar(45);
gasCars[0].drive(15_000);
gasCars[1] = new GasCar(33);
gasCars[1].drive(5_000);
gasCars[2] = new GasCar(86);
gasCars[2].drive(2_500);
Arrays.sort(gasCars, new CarDrivenDistanceComparator() );
// gasCars = {GasCar(fuelCapacity = 86, drivenkm = 2500), GasCar(fuelCapacity = 33, drivenkm = 5000), GasCar(fuelCapacity = 45, drivenkm =15000)}
```

- The design principle we see here is delegation. We delegate the responsibility to do something to another object.
  - Delegation is often used to make an oo hierarchy robust, while keeping it simple. It adds layers of indirection, but keeps hierarchies.

- *Comparator* shows, how this layer makes sense to be delegated:
  - Sometimes, we cannot change the present *@Override* of *Comparable.compareTo()*, e.g. because we can't access the classes code.
  - From a SOLID standpoint, classes should not be blindly modified by changing *Comparable.compareTo()*. – It hurts the OCP!
  - From a SOLID standpoint a specific class, which has a certain responsibility supports the SRP!
  - Modifying a present *@Override* of *Comparable.compareTo()* might break others' code!
  - Delegation allows adding functionality at run time! – We'll look at this aspect in a minute!

63

- There is also a pretty clear restriction on implementing *Comparator*: the data to be compared must be accessible!

The Comparator Interface – Part 4

- Think about following task: *gasGars* should be sorted for the driven distance in a descending manner. – How to do that?
  - It is simple! We just need another *Comparator*!

```
Arrays.sort(gasCars, new CarDrivenDistanceComparator().reversed());
// gasCars = {GasCar(fuelCapacity = 45, drivenkm =15000), GasCar(fuelCapacity = 33, drivenkm = 5000), GasCar(fuelCapacity = 86, drivenkm = 2500)}
```

  - The default method *Comparator.reversed()* creates a new reversed *Comparator*, decorating our *CarDrivenDistanceComparator*.
  - Reversing means, that the new *Comparator* passes the two objects to be compared to the original *Comparator* as *rhs* then *lhs*.
  - Mind, that this is an awesome application of the decorator pattern, showing its usefulness and power!

- The JDK provides also predefined *Comparator*s for *String*s.
  - We can sort a *String[]* independent of the case of the individual elements with the *Comparator String.CASE_INSENSITIVE_ORDER*:

```
String[] cities = {"Rome", "loNdon", "PariS", "athens", "BerlIn", "ViEnna"};
Arrays.sort(cities, String.CASE_INSENSITIVE_ORDER);
// cities = {"athens", "BerlIn", "loNdon", "PariS", "Rome", "ViEnna"}
```

  - We can also combine *String.CASE_INSENSITIVE_ORDER* with *Comparator.reversed()* for descending case independent sorting:

```
String[] cities = {"Rome", "loNdon", "PariS", "athens", "BerlIn", "ViEnna"};
Arrays.sort(cities, String.CASE_INSENSITIVE_ORDER.reversed());
// cities = {"ViEnna", "Rome", "PariS", "loNdon", "BerlIn", "athens"}
```

Generic Variance in Algorithms – Part 1

- Up to now, we discussed bound wildcards being bound to concrete types, e.g. on a reference.
- We can also bind wildcards to type parameters of a generic type.
  - This application of wildcards is even more complex, than the stuff we have discussed up to here.
- … but this way of using wildcards also allows writing even more powerful algorithms, consider this code:

```
public <T> T append(List<T> dest, List<T> src) {
        T last = null;
        for (T item : src) {
                dest.add(item);
                last = item;
        }
        return last;
}
```

  - *append()* simply appends the elements of *src* to *dest* and returns the last appended element to the caller.
- Following code should reasonably work, but its doesn't:

```
List<String> in = Arrays.asList("Hercules", "Agamemnon", "Aphrodite", "Paris", "Hector");
List<Object> out = new ArrayList<>();
String lastCopied = append(out, in);   // Invalid! Cannot be applied to given types; required: List<T>, List<T> found: List<String>, List<Object>
                                       // reason: inferred type does not conform to equality constraint(s) inferred: Object equality constraints(s): Object, String
```

  - With our current knowledge, this can be explained easily: *append()* requires the same type argument of *T* for *dest* and *src*. 65
  - *String* and *Object* are related by inheritance, but *List<String>* and *List<Object>* are not. – The compiler cannot infer a discrete type.

Generic Variance in Algorithms – Part 2

- But we can tell the compiler, that *dest* and *src* can be *List*s, that contain related types using an upper bound wildcard:

```
public <T> T append(List<T> dest, List<? extends T> src) {
        T last = null;
        for (T item : src) {
                dest.add(item);
                last = item;
        }
        return last;
}
```

  - Actually, with this change, calling *append()* works "somewhat", we can call it with the arguments as before correctly:

```
List<String> in = Arrays.asList("Hercules", "Agamemnon", "Aphrodite", "Paris", "Hector");
List<Object> out = new ArrayList<>();
append(out, in);        // OK!
```

  - It works, because the inferred type of *T* will be discretized to *Object*; the upper bound wildcards makes this work.

- Alas, it doesn't work for the returned value being assigned to a *String*:

```
String lastAppended = append(in, out);      // Invalid! Incompatible types: inference variable T has incompatible bounds
                                            // equality constraints: Object upper bounds: String, Object
```

  - The reason is, that *T* was inferred to *Object* and not *String*! – The assignment from *Object* to *String* doesn't work!

66

Generic Variance in Algorithms – Part 3

- The "full" solution to make *append()* work as intended from the start is the application of an lower bound wildcard on *dest*:

```
public <T> T append(List<? super T> dest, List<T> src) {
        T last = null;
        for (T item : src) {
                dest.add(item);
                last = item;
        }
        return last;
}
```

```
List<String> in = Arrays.asList("Hercules", "Agamemnon", "Aphrodite", "Paris", "Hector");
List<Object> out = new ArrayList<>();
String lastAppended = append(out, in); // OK!
```

  - With the lower bound wildcard on *dest*, the complier rather infers the type of *T* to the more special type, namely *String*.

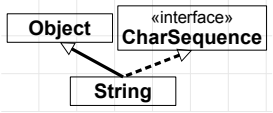
- From a pragmatical standpoint, that's all we need to do, but there's still potential for improvements!

Generic Variance in Algorithms – Part 4

- There is still one case to handle, namely, when we call *append()* with an explicit type argument, not relying on inference:

```
public <T> T append(List<? super T> dest, List<T> src) {
        // pass
}
```

```
List<String> in = Arrays.asList("Hercules", "Agamemnon", "Aphrodite", "Paris", "Hector");
List<Object> out = new ArrayList<>();
CharSequence lastAppended = this.<CharSequence>append(out, in); // Invalid! Incompatible types: List<String>
                                                                // cannot be converted to List<CharSequence>
```



  - The compiler captures *dest*'s *T* to *String*, but *src*'s *T* is set to *CharSequence* explicitly: this doesn't match for the call.
  - It looks contrived, but we as implementors should give to users/callers of *append()* a maximum of compatibility and simplicity.

- To make this way of calling *append()* work
  - (1) we use a lower bound wildcard on *T* at *dest*
  - (2) and an upper bound wildcard on *T* at *src*.

```
// Excellent!
public <T> T append(List<? super T> dest, List<? extends T> src) {
        T last = null;
        for (T item : src) {
                dest.add(item);
                last = item;
        }
        return last;
}
```

```
CharSequence lastAppended = this.<CharSequence>append(out, in); // OK!
```

  - The compiler's work is easy now: in both parameters, *T* is captured as *CharSequence*.

Generic Variance in Algorithms – Part 5

- Effectively, we get along with following signature to cover most cases:

```
public <T> T append(List<? super T> dest, List<? extends T> src) {
        // pass
}
```

  - Downside: signature is complex.
  - Benefit: "reasonable code also compiles".

- Due to the complexity of matters, we can memorize a basic rule as PECS: Producer extends Consumer super.

- Assume, we use a generic with the type parameter *T* in an algorithm.
  - When *T*'s are read (produced) in an algorithm, the generic in question should have an upper bound on *T* (extends).
  - When *T*'s are passed as arguments (consumed) in an algorithm, the generic in question should have a lower bound on *T* (super).

```
public <T> T append(List<? super T> dest, List<? extends T> src) {
        T last = null;
        for (T item : src) { // reading T from src: producer extends
                dest.add(item);  // passing T to dest: consumer super
                last = item;
        }
        return last;
}
```

PECS in the JDK – Part 1

- Lower bound wildcards can be found in the JDK, where generics accept type parameter instances.
  - E.g. *Collections.sort()*:

```
// somewhere in the JDK:
public class Collections {
        public static <T> void sort(List<T> list, Comparator<? super T> c) {
                // pass
        }
}
```

  - Using *T* as lower bound makes sense, if we have a *List<Bus>*, we should be allowed to sort it with a *Comparator<Car>*!
  - The passed *Comparator* will accept/consume Ts → "Consumer super".

- Upper bound wildcards can be found in the JDK, where generics return type parameter instances.
  - E.g. *List.addAll()*:

```
// somewhere in the JDK:
public interface List<E> {
        boolean addAll(Collection<? extends E> c) {
                // pass
        }
}
```

  - Using *E* as upper bound makes sense: if we have a *List<Car>*, we should be allowed *addAll()* of a *List<Bus>*!
  - The passed *Collection* will return/provide/produce Ts → "Producer extends".

PECS in the JDK – Part 2

- And the JDK also shows combinations of bounds to get maximal compatibility and reusability.
  - E.g. *Collections.copy()*:

```
// somewhere in the JDK:
public class Collections {
        public static <T> void copy(List<? super T> dest, List<? extends T> src) {
                // pass
        }
}
```

  - It makes sense: we can copy a *List<Object>* into a *List<String>*, even if the type parameter was explicitly set to *CharSequence*.
  - *dest* should accept/consume *T*s → "Consumer super"; *src* should return/provide/produce *T*s → "Producer extends".

- Most often generics with wildcard bounds are rather used in method parameters than as return types.
  - If returned generics with wildcard bounds must be handled explicitly it is a challenge even for seasoned Java programmers!
  - Therefor it is recommended to strive to return generics without any bounds. → Don't be too clever!
  - Btw: The wildcard can be casted away!

Peculiarities of Generics

- We can also use generic parameters for the throws specification list of a method:

```
// <MyInterface.java>
public interface MyInterface<X extends Throwable> {
        void add(int item) throws X;
}
```

  - *X* is often used as type parameter name for *Exception* and *Throwable* bound types.
  - Then an implementing class just passes an exception type (mind the *Throwable* bound) respectively:

```
// <MyClass.java>
public class MyClass implements MyInterface<RuntimeException> {
        public void add(int item) throws RuntimeException {
            throw new RuntimeException();
        }
}
```

- Using type parameters for exception types basically only makes sense on interfaces, just consider:

```
// <MyInvalidClass.java>
public class MyInvalidClass<X extends Throwable> {
        public void add(int item) throws X {
            throw new X(); // Invalid! Type parameter X cannot be instantiated directly
        }
}
```

  - It boils down to one remaining restriction with generics here: we cannot create instances of the type parameter's type.

Thank you!