

Generics



Object Oriented Programming

<https://softeng.polito.it/courses/09CBI>



SoftEng
<http://softeng.polito.it>

Version 2.11.1

© Marco Torchiano, 2021



Motivation

- Often the same operations have to be performed on objects of unrelated classes
 - ◆ A typical solution is to use `Object` references to accommodate objects of any type
- `Object` references bring cumbersome code
 - ◆ Several explicit casts are required
 - ◆ Compile-time checks are limited
 - ◆ Down-casts can be checked at run-time only
- Solution
 - ◆ Use **generic** classes and methods

Example

- We may need to represent pairs or values different types (e.g. `int`, `String`, etc.)
- Use of `Object` to allow any type

```
public class Pair {  
    Object a, b;  
  
    public Pair(Object a, Object b)  
    { this.a=a; this.b=b; }  
  
    Object first(){ return a; }  
  
    Object second(){ return b; }  
}
```

Note: No primitive types,
only wrappers allowed

Example: Object-based use

- Object allows usage with diverse types

```
Pair sp = new Pair("One", "Two");  
Pair ip = new Pair(1,2);
```

- Though you need explicit down-casts:

```
String a = (String) sp.second();  
int i = (Integer) ip.first();
```

- Down-casts are checked at run-time

```
String b = (String) ip.second();
```



ClassCastException
at run-time

Example: Object-based use

- No check is possible at compile time about homogeneity of elements:

```
Pair mixpair = new Pair(1, "Two");  
Pair pairmix = new Pair("One", 2);
```

- Extra code is required for safety:

```
Object o = mixpair.second();  
if(o instanceof Integer){  
    i = (Integer)o;  
}else { ... }
```

Generic class

```
public class Pair<T> {  
    T a, b;  
    public Pair(T a, T b) {  
        this.a = a;    this.b = b;  
    }  
    public T first(){ return a; }  
    public T second(){ return b; }  
    public void set1st(T x){ a = x; }  
    public void set2nd(T x){ b = x; }  
}
```

Generics use

- Declaration is slightly longer:

```
Pair<String> sp = new Pair<>("One", "Two");  
Pair<Integer> ip = new Pair<>(1, 2);  
Pair<String> mixp = new Pair<>(1, "Two");
```

Compiler error

- Use is more compact and safer:

```
String a = sp.second();  
int b = ip.first();  
String bs = ip.second();
```

No down-cast
is required

Integer can be
auto-unboxed

Compiler error:
type mismatch

Generic type declaration

- Syntax:

(class | interface) Name <P₁ , P₂>

- Type parameters, e.g. P₁:

- ◆ Represent classes or interfaces
- ◆ Conventionally uppercase letter
- ◆ Usually:
T(ype), **R**(eturn), **E**(lement), **K**(ey), **V**(alue)

Generic Interfaces

- All standard interfaces and classes have been defined as generics
 - ◆ since Java 5
- Use of generics leads to code that is
 - ◆ safer
 - ◆ more compact
 - ◆ easier to understand
 - ◆ equally performing

Generic Comparable

- Interface `java.lang.Comparable`

```
public interface Comparable<T>{  
    int compareTo(T obj);  
}
```

- Semantics: returns
 - ◆ a negative integer if this precedes obj
 - ◆ 0, if this equals obj
 - ◆ a positive integer if this succeeds obj

Generic Comparable

- Without generics:

```
public class Student implements Comparable{  
    int id;  
    public int compareTo(Object o) {  
        Student other = (Student)o;  
        return this.id - other.id;  
    } }
```

- With generics:

```
public class Student  
    implements Comparable<Student> {  
    int id;  
    public int compareTo(Student other) {  
        return this.id - other.id;  
    } }
```

Generic Iterable and Iterator

```
public interface List<E>{  
    void add(E x);  
    Iterator<E> iterator();  
}
```

```
public interface Iterator<E>{  
    E next();  
    boolean hasNext();  
}
```

Iterable example

```
class Letters implements Iterable<Character> {  
    private char[] chars;  
    public Letters(String s) {  
        chars = s.toCharArray(); }  
    public Iterator<Character> iterator() {  
        return new Iterator<Character>() {  
            private int i=0;  
            public boolean hasNext() {  
                return i < chars.length;  
            }  
            public Character next() {  
                return chars[i++];  
            }  
        };  
    } }
```

Iterable example

- Without generics

```
Letters l = new Letters("Sequence");  
for(Object e : l){  
    char v = ((Character)e);  
    System.out.println(v);  
}
```

- With generics

```
Letters l = new Letters("Sequence");  
for(char ch : l){  
    System.out.println(ch);  
}
```

Diamond operator

- Reference type parameter must match the class parameter used in instantiation

- ◆ E.g.

```
List<String> l=new LinkedList<String>();
```

- The Java compiler can infer the type when the diamond operator is used:

```
List<String> l = new LinkedList<>();
```

- ◆ Since Java 7

Generic method

- Syntax:

modifiers <T> type name(pars)

- pars can be:

- ◆ as usual
- ◆ T
- ◆ type<T>

Generic Method Example

element must have same type as array type

```
public static <T>
    boolean contains(T[] ary, T element) {
        for(T current : ary) {
            if(current.equals(element))
                return true;
        }
        return false;
    }
```

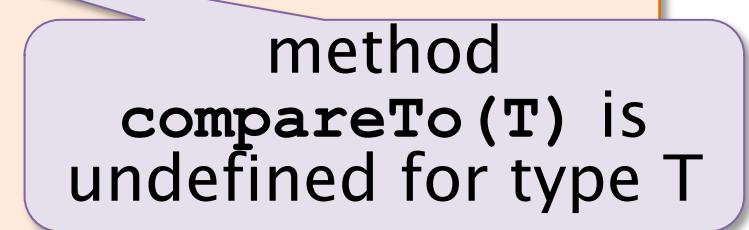
```
String[] words = { ... };
boolean found = contains(words, "fox");
```

Unbounded type

- The type parameters used in generics are unbounded by default
 - ◆ I.e. there are no constraints on the types that can be substituted to the type parameters
- The safe assumption for any type parameter **T** is that **T extends Object**
 - ◆ References of a type parameter **T** at least provide members that are defined in class **Object** (e.g., **equals()**)

Unbounded generic sorting

```
public static <T>
void sort(T v[]) {
    for(int i=1; i<v.length; ++i)
        for(int j=1; j<v.length; ++j) {
            if(v[j-1].compareTo(v[j])>0) {
                T o=v[j];
                v[j]=v[j-1];
                v[j-1]=o;
            }
        }
}
```



method
compareTo(T) is
undefined for type T

Bounded types

- Express constraints on type parameters

`< T extends B >`

- class **T** can be replaced only with types extending **B** including **B** itself
 - ◆ **B** is called an **upper bound**
- It is possible to specify multiple upper bounds

`<T extends B1 { & B2 } >`

Bounded generic sorting

```
public static <T extends Comparable>
void sort(T v[]) {
    for(int i=1; i<v.length; ++i)
        for(int j=1; j<v.length; ++j) {
            if(v[j-1].compareTo(v[j])>0) {
                T o=v[j];
                v[j]=v[j-1];
                v[j-1]=o;
            }
        }
}
```

Ok: method
compareTo(T) is
defined in Comparable

Bounded generic sorting

Since Comparable is a generic interface itself

```
public static <T extends Comparable<T>>
void sort(T v[]) {
    for(int i=1; i<v.length; ++i)
        for(int j=1; j<v.length; ++j) {
            if(v[j-1].compareTo(v[j])>0) {
                T o=v[j];
                v[j]=v[j-1];
                v[j-1]=o;
            }
        }
}
```

Bounded Comparator

```
public static <T,C extends Comparator<T>>
void sort(T v[], C cmp) {
    for(int i=1; i<v.length; ++i)
        for(int j=1; j<v.length; ++j) {
            if(cmp.compare(v[j-1],v[j])>0) {
                T o=v[j];
                v[j]=v[j-1];
                v[j-1]=o;
            }
        }
}
```

Bounded Comparator

```
public static <T>
void sort(T v[], Comparator<T> cmp) {
    for(int i=1; i<v.length; ++i)
        for(int j=1; j<v.length; ++j) {
            if(cmp.compare(v[j-1], v[j])>0) {
                T o=v[j];
                v[j]=v[j-1];
                v[j-1]=o;
            }
        }
}
```

Java Generics

SUBTYPING AND Co-VARIANCE

Generics subtyping

- We must be careful about inheritance when generic types are involved
 - ◆ `Integer` is a subtype of `Object`
 - ◆ `Pair<Integer>` is **NOT** subtype of `Pair<Object>`

Attention!!

```
Pair<Integer> pi = new Pair<>(0,1);
```

```
Pair<Object> pn = pi;
```

if this were legal then...

```
pn.set1st("0.5");
```

```
Integer i = pi.first();
```

.. we could end up assigning a
String to an Integer reference

Containers and elements

- Containers can be co-variant or invariant.
- **Co-variance**: elements inheritance implies containers inheritance
 - ◆ If **A** *extends* **B**
 - ◆ Then **Container<A>** *extends* **Container**
 - ◆ In Java this brings to unsafe assumptions
- **Invariance**: elements inheritance does not imply container inheritance
 - ◆ Type safe assumption

Generics invariance

- Generics types are invariant
- The elements type are the type arguments
 - ◆ The fact `Integer` extends `Object` does not imply `Pair<Integer>` extends `Pair<Object>`
- Co-variance would lead to type clashes

```
Pair<Integer> pi;
```

```
Pair<Object> pn ≠ pi;
```

Type mismatch

Arrays co-variance

- Arrays are type co-variant containers
 - ◆ If **A** extends **B**
 - ◆ Then **A[]** extends **B[]**
- Co-variance make type clashes possible

```
String[] as = new String[10];  
Object[] ao;  
ao = as; // this is ok!!!  
ao[1] = new Integer(1);
```

java.lang.ArrayStoreException

Invariance limitations

- An attempt to have a universal method:

```
void printPair(Pair<Object> p) {  
    System.out.println(p.first() + "-" +  
                       p.second());  
}
```

- Won't work with e.g. `Pair<Integer>`

```
Pair<Integer> p = new Pair<>(7,4);  
printPair(p);
```

Method is not applicable
for the argument

Invariance limitations

- Universal method must be generic

```
<T> void printPair(Pair<T> p) {  
    System.out.println(p.first() + "-" +  
                       p.second());  
}
```

- Even if declared as generic, the method in itself is not generic
 - ◆ Type T is never mentioned in the method

Wildcards

- Allow to express (lack of) constraints when *using* generic types
- `<?>`
 - ◆ unknown, unbounded
- `<? extends B>`
 - ◆ upper bound: only sub-types of B
 - Including B
- `<? super D>`
 - ◆ lower bound: only super-types of D
 - Including D

Invariance limitations

The unknown must be treated in the most generic way possible

- Universal method must be generic

```
void printPair(Pair<?> p) {  
    System.out.println(p.first() + "-" +  
                       p.second());  
}
```

Pair of unknown

- Compiler treats unknowns conservatively

```
void clearFirst(Pair<?> p) {  
    p.setFirst("");  
}
```

Method is not applicable
for the argument

Wildcard constraints

- The **?** (unknown) type is literally unknown therefore the compiler treats it in the safest possible way:
 - ◆ Only method from `Object` are allowed
 - ◆ Assignment to an unknown reference is illegal

Bounded wildcard – example

```
double sum(Pair<Number> p) {  
    return p.a.doubleValue() + p.b.doubleValue();  
}
```

Cannot be invoked
with `Pair<Integer>`

```
<T extends Number> double sumB(Pair<T> p)  
{ ... }
```

Defines an upper bound for
the type parameter

```
double sumUB(Pair<? extends Number> p)  
{ ... }
```

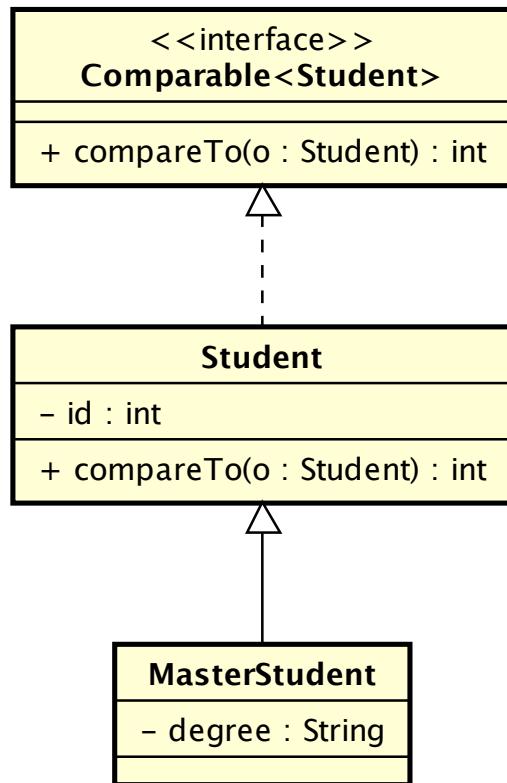
Unknown with upper bound
Equivalent but more compact

Sorting a pair

```
void <T extends Comparable<T>>
sortPair(Pair<T> p) {
    if(p.first().compareTo(p.second()) > 0) {
        T tmp = p.first();
        p.setFirst(p.second());
        p.setSecond(tmp);
    }
}
```

Sorting a pair example

```
Pair<MasterStudent> pm = new Pair<>(...);  
sortPair(pm);
```



Method is not applicable
for the argument:
(Pair<MasterStudent>)

MasterStudent **MasterStudent**
~~void <T extends Comparable<T>>~~ ~~Comparable<T>~~
sortPair(Pair<T> p)
MasterStudent

MasterStudent
does not implement
Comparable<MasterStudent>

Sorting a pair

T must implement Comparable
of any superclass of T
(including T itself)

```
static <T extends Comparable<? super T>>
void sortPair(Pair<T> p) {
    if(p.first().compareTo(p.second()) > 0) {
        T tmp = p.first();
        p.setFirst(p.second());
        p.setSecond(tmp);
    }
}
```

MasterStudent implements
Comparable<Student super MasterStudent>

Sort method

- On Comparable objects:

```
static <T extends Comparable<? super T>>  
void sort(T[] list)
```

- For backward compatibility, in class Array sort is defined as:
 - `public static void sort(Object[] a)`
 - No compile time check is performed.

- Using a Comparator object:

```
static <T> void  
sort(T[] a, Comparator<? super T> cmp)
```

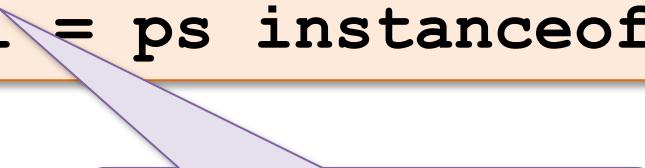
Java Generics

TYPE ERASURE

Generics classes

- The compiler generates only one class for each generic type declaration
 - ◆ Compilation **erases** the type parameters

```
Pair<Integer> pi = new Pair<>(1,2);  
Pair<String> ps = new Pair<>("one", "two");  
  
boolean is = pi instanceof Pair;  
boolean si = ps instanceof Pair;
```



Both are **true**

Type erasure

- Classes corresponding to generic types are generated by **type erasure**
- The erasure of a generic class is a **raw type**
 - ◆ where any reference to the parameters is substituted with the parameter erasure
 - ◆ Raw type of generic class $G<T>$ is G
- Erasure of a parameter is the erasure of its first boundary
 - ◆ If no boundary is provided then it is **Object**

Type erasure – examples

- For: <T>
 - ◆ T → Object
- For: <T extends Number>
 - ◆ T → Number
- For: <T extends Number & Comparable>
 - ◆ T → Number

(first boundary --> Number)

Type erasure – consequences I

- Since there is only one class, within the class no information is available about the type parameters
 - ◆ They have been erased
 - The actual value of type parameter is known only to the user of a generic type
 - Compiler applies checks only when a generic type is used, not within it.
-

Type erasure – consequences II

- Whenever a generic or a parameter is used a cast is added to its erasure
- To avoid inconsistencies and wrong expectations
 - ◆ `instanceof` cannot be used on generic types
 - ◆ `instanceof` is valid for `G` or `G<?>`

Type erasure – consequences III

- It is not possible to instantiate an object of the type parameter from within the class

```
class Triplet<T> {  
    private T[] triplet;  
    Triplet(T a, T b, T c) {  
        triplet = new T[]{a,b,c};  
    }  
}
```

Compiler cannot
create a generic
array of T

- ◆ The erasure cannot be used in any way inside the raw type

Type erasure- consequences IV

- Overloads and overrides are checked by compiler after type erasure

```
class Base<T> {  
    void m(int x) {}  
    void m(T t) {}  
    void m(String s) {}  
<N extends String> void m(N x) {}  
    void m(Pair<?> l) {}  
}
```

Object

String

Duplicate method

Type erasure- consequences V

- Inheritance together with generic types leads to several possibilities
- It is not possible to implement twice the same generic interface with different types

```
class Student implements Comparable<Student>
```

```
class MasterStudent extends Student  
    implements Comparable<MasterStudent>
```

The interface **Comparable** cannot be implemented more than once with different arguments

Type inference

- Upon generic method invocation, compiler infers the type argument (**capture**)
- It is possible to use a type witness, although often useless
 - ◆ `Arrays.<Student>sort(sv);`
- Inference is based on
 - ◆ Target type
 - ◆ Argument type

USE OF GENERICS IN PRACTICE

Standard Functional Interfaces

Interface	Method
Function <T ,R>	R apply(T t)
BiFunction <T ,U,R>	R apply(T t, U u)
BinaryOperator <T>	T apply(T t, T u)
UnaryOperator <T>	T apply(T t)
Predicate <T>	boolean test(T t)
Consumer <T>	void accept(T t)
BiConsumer <T ,U>	void accept(T t, U u)
Supplier <T>	T get()

Primitive specializations

- Functional interfaces handle references
 - Specialized versions are defined for primitive types (`int`, `long`, `double`, `boolean`)
 - Functions: `ToTypeFunction`
`Type1ToType2Function`
 - Suppliers: `TypeSupplier`
 - Predicate: `TypePredicate`
 - Consumer: `TypeConsumer`
-

Instance method of object

- Method is invoked on the object
 - Parameters are those of the method

v -> hexDigits.charAt(v)

```
String hexDigits = "0123456789ABCDEF";
```

```
IntFunction<Character>
```

```
IntToCharFun hex = hexDigits::charAt;
```

```
System.out.println("Hex for 10 : "
```

```
+ hex.apply(10) );
```

```
interface IntToCharFun  
    char apply(int value)  
}
```

```
interface IntFunction<R> {  
    R apply(int value);  
}
```

Instance method reference

- The first argument is the object on which the method is invoked
 - ◆ The remaining arguments are mapped to the method arguments

```
ToIntFunction<String>
```

s -> s.length()

```
StringToIntFunction f = String::length;
```

```
for(String s : words) {
```

```
    System.out.println(f.apply(s));
```

```
,
```



```
interface ToIntFunction<T> {
```

```
    int applyAsInt(T x);
```

```
}
```

```
StringToIntFunction {
```

```
    int apply(String s);
```

int

Constructor reference

- The return type is a new object
 - ◆ Parameters are the constructor's parameters

i -> new Integer(i);

IntFunction<Integer>

```
IntegerBuilder builder = Integer::new;  
Integer i = builder.apply.build(1);
```

```
interface IntFunction<R> {  
    R apply(int value);  
}
```

}

Integer

```
IntegerBuilder{  
    Integer build(int value);
```

Generic Comparator

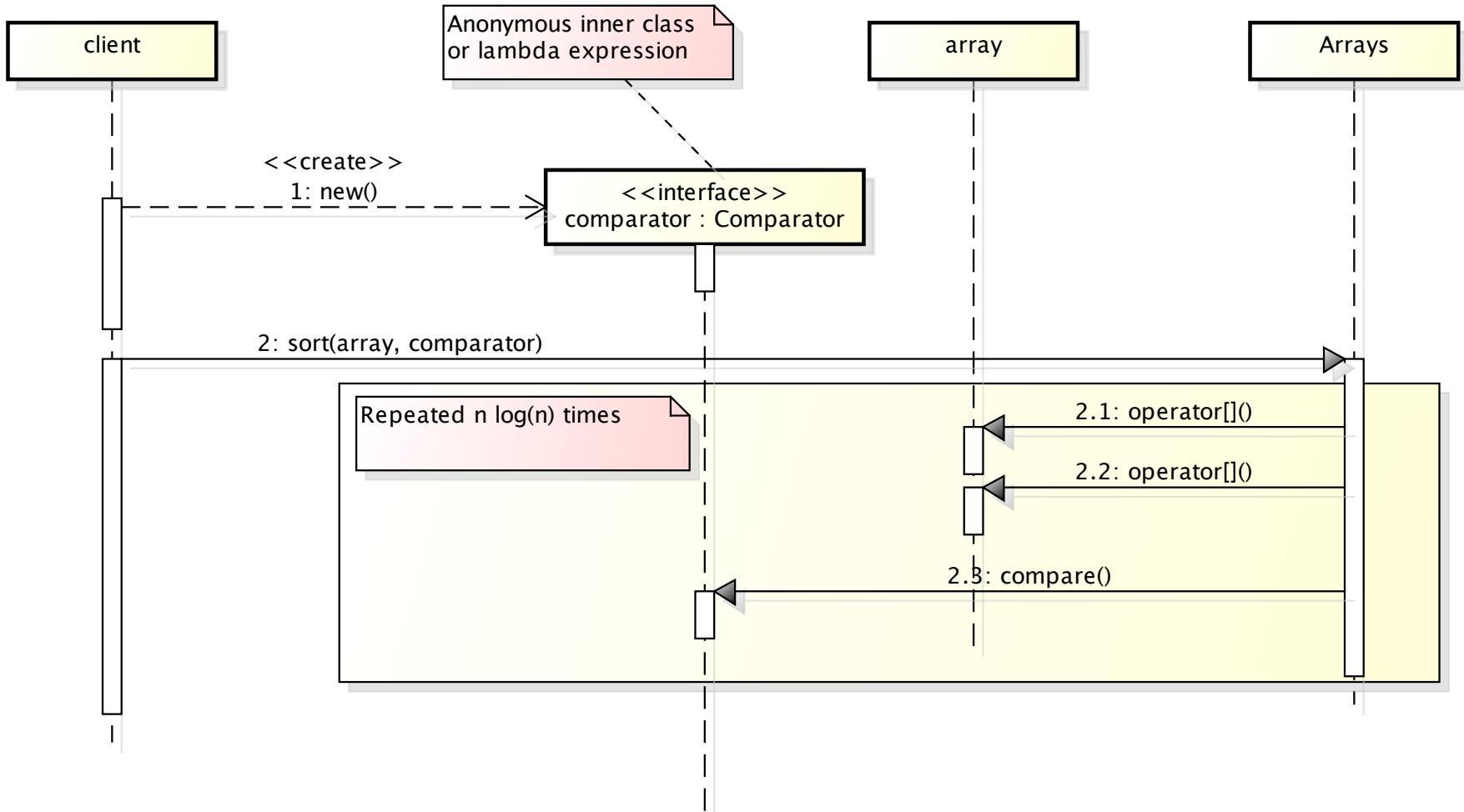
Interface `java.util.Comparator`

```
public interface Comparator<T>{  
    int compare(T a, T b);  
}
```

```
Arrays.sort(sv, (a,b) -> a.id - b.id );
```

```
Arrays.sort(sv, new Comparator<Student>(){  
    public void compare(Student a, Student b){  
        return a.id - b.id  
    } );
```

Comparator behavior



Comparator factory

- Most comparators take some information out of the objects to be compared
 - ◆ Typically through a getter
 - ◆ Such values are primitive or are comparable using their natural order (i.e. Comparable)
- Such comparator can be generated starting from a key getter functional object:

```
static <T, U extends Comparable<U>>  
Comparator<T>
```

comparing (Function<T, U> keyGetter)

Comparator.comparing()

Comparator.comparing

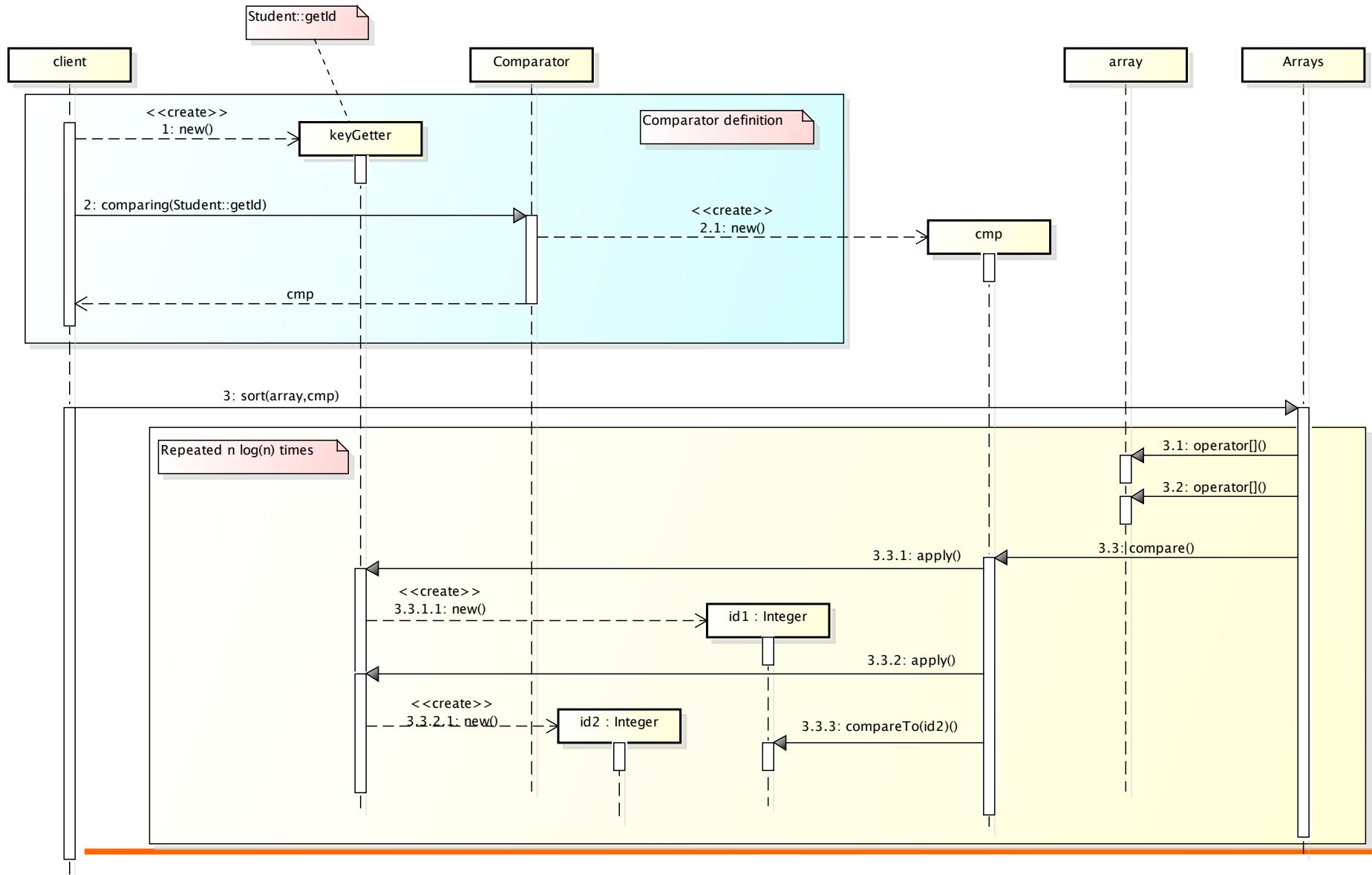
```
Arrays.sort(sv, comparing(Student::getId));
```

Requires:

```
import static java.util.Comparator.*
```

```
static <T,U extends Comparable<? super U>>
Comparator<T>
comparing(Function<T,U> keyGetter) {
    return (a,b) -> keyGetter.apply(a).
        compareTo(keyGetter.apply(b));
}
```

Comparator factory behavior



Comparator.comparingInt

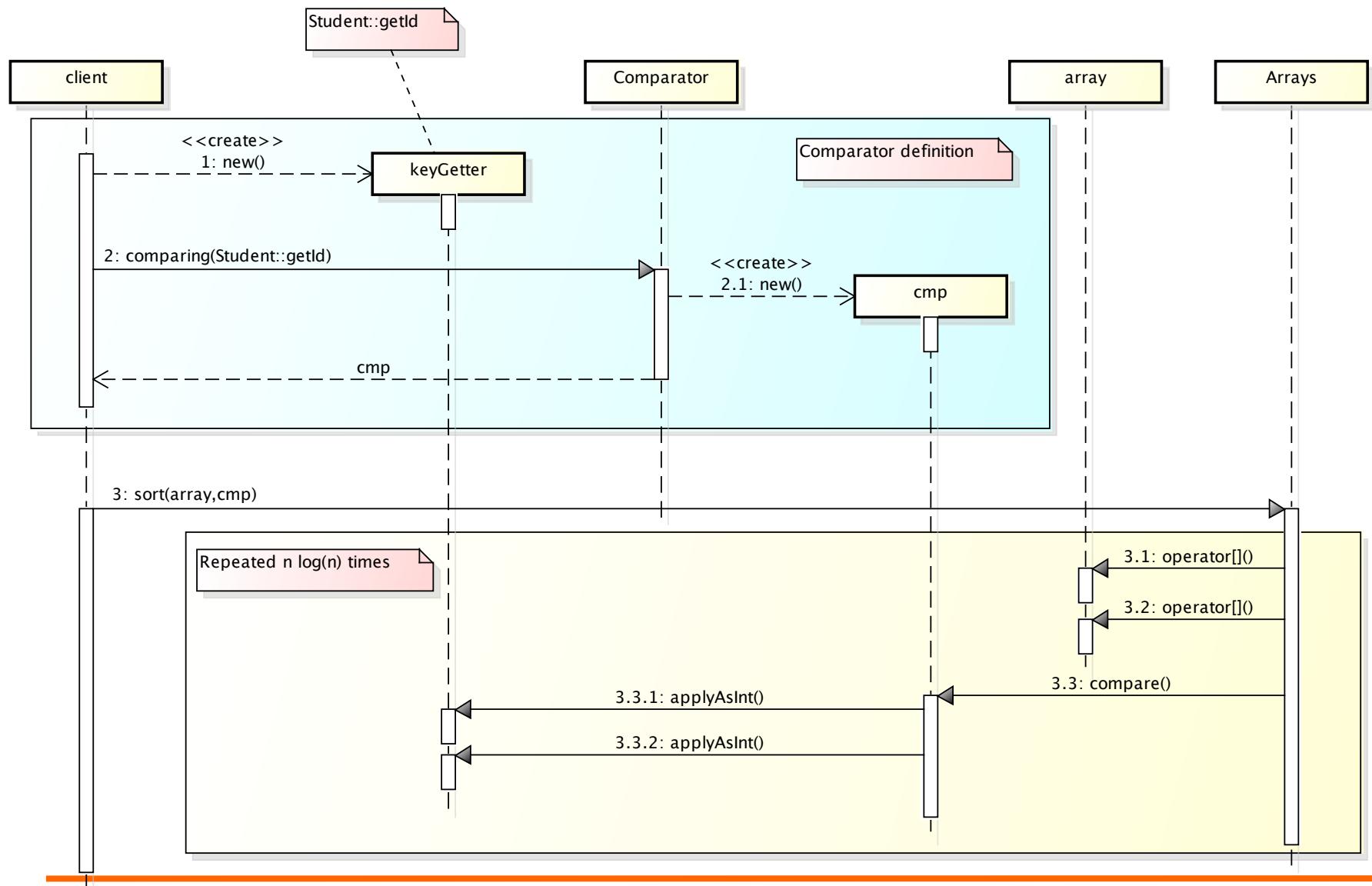
```
Arrays.sort(sv, comparingInt(Student::getId))
```

Requires:

```
import static java.util.Comparator.*
```

```
static <T, U extends Comparable<? super U>>
Comparator<T>
    comparing(ToIntFunction<T, U> keyGetter) {
        return (a, b) -> keyGetter.applyAsInt(a) -
            keyGetter.applyAsInt(b);
    }
```

IntComparator factory behavior



Performance

Comparator	Time
<code>(a,b) -> a.id - b.id</code> only if id field is public	100
<code>(a,b) -> a.getId() - b.getId()</code> if id field is private: getter	120
<code>comparingInt(Student::getId)</code> more efficient than the generic version	135
<code>comparing(Student::getId)</code>	185

Also use a lot of memory due
to wrapper objects creation

Relative performance

Comparator historical perspective

```
Arrays.sort(sv, new Comparator() {  
    public int compare(Object a, Object b) {  
        return ((Student)a).id - ((Student)b).id;  
    } } );
```

Java ≥ 2

```
Arrays.sort(sv, new Comparator<Student>() {  
    public int compare(Student a, Student b) {  
        return a.getId() - b.getId();  
    } } );
```

Java ≥ 5, Generics

Java ≥ 8, Lambda

```
Arrays.sort(sv, (a,b) ->a.getId() -b.getId());
```

```
Arrays.sort(sv, comparing(Student::getId));
```

Java ≥ 8, Method reference

Functional interface composition

- Reverse order method
 - ◆ Not a Comparator method!

```
static <T> Comparator<T>
    reverse(Comparator<T> cmp) {
        return (a,b) -> cmp.compare(b,a) ;
    }
```

```
Arrays.sort(sv, reverse(
    comparing(Student::getId))) ;
```

Comparator composition

- Reverse order
 - ◆ Default method `Comparator.reversed()`

```
default <T> Comparator<T> reversed() {  
    return (a,b) -> this.compare(b,a);  
}
```

```
Arrays.sort(sv,  
            comparing(Student::getId).reversed());
```

Comparator composition

- Multiple criteria
 - ◆ Default method
`Comparator.thenComparing()`

```
default <T> Comparator<T>
    thenComparing(Comparator<T> other) {
    return (a,b) -> {
        int r = this.compare(a,b);
        if(r!=0) return r;
        else return other.compare(a,b);
    }
}
```

Comparator composition

- Multiple criteria

```
default <U extends Comparable<U>
Comparator<T> thenComparing(Function<T,U> ke) {
    return (a,b) -> {
        int r = this.compare(a,b);
        if(r!=0) return r;
        return ke.apply(a).compareTo(ke.apply(b));
    }
}
```

Comparator composition

```
Arrays.sort(sv, (a,b)-> {
    int l = a.last.compareTo(b.last);
    if(l!=0) return l;
    return a.first.compareTo(b.first);
}));
```

```
Arrays.sort(sv,
    comparing(Student::getLast).
    thenComparing(Student::getFirst));
```

Functional Interface Composition

- Predicate
 - ◆ `default Predicate<T> and(Predicate<T> o)`
 - ◆ `default Predicate<T> or(Predicate<T> o)`
 - ◆ `default Predicate<T> negate()`
- Function
 - ◆ `default <V>`
`Function<T,V> andThen(Function<R,V> after)`
 - ◆ `default <V>`
`Function<V,R> compose(Function<V,T> before)`

Wrap-up

- Generics allow defining type parameter for methods and classes
- The same code can work with several different types
 - ◆ Primitive types must be replaced by wrappers
- Generics containers are type invariant
 - ◆ Wildcard, ? (read as unknown)
- Generics are implemented by type erasure
 - ◆ Checks are performed at compile time