Generics

Object Oriented Programming

http://softeng.polito.it/courses/09CBI

Version 2.9.0 - April 2018

Motivation

- Often the same operations has to be performed on objects of unrelated classes
  - Typical solution is to use Object references to accommodate any object type
- Object references bring cumbersome code
  - Several explicit casts are required
  - Compiler checks can be limited
  - Down-cast 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`** for any value 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` allows usage with different 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();
```

- That cannot be checked at compile time

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

`ClassCastException` at run-time

Example

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

```
Pair mixp = new Pair(1,"Two");
Pair ximp = new Pair("One”,2);
```

- Extra code is required for safety:

```
Object o = mixp.second();
if(o instanceof Integer){ … }
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 setFirst(T a){ this.a=a; }
  public void setSecond(T b){ this.b=b; }
}
```


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: type mismatch

- 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. P_1:
  - Represent classes or interfaces
  - Conventionally uppercase letter
  - Usually: T(ype), 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);
}
```


Iterable example

```
class Random implements Iterable<Integer> {
  private int[] values;
  public Random(int n, int min, int max){ … }
  public Iterator<Integer> iterator() {
    return new Iterator<Integer>(){
        private int position=0;
        public boolean hasNext() {
           return position < values.length;
        }
        public Integer next() {
           return values[position++];
        }
     };
}}
```

`Iterable` example

- Without generics:

```
Random seq = new Random(10,5,10);
for(Object e : seq){
  int v = ((Integer)e).intValue();
  System.out.println(v);
}
```

- With generics:

```
Random seq = new Random(10,5,10);
for(int v : seq){
  System.out.println(v);
}
```


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

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

Unbounded example

- A point with varying precision

```
public class Point<T> {
  T x; T y;
  public Point(T x, T y){
    this.x = x; this.y = y;
  }
  public double length(){
    return Math.sqrt(
     Math.pow(x.doubleValue(),2)
     + Math.pow(y.doubleValue(),2) );
  }
}
```

method undefined for type T

Bounded types

- Express constraints on type parameters

  <T extends B1 { & B2 } >

  - class **T** can be replaced only with types extending from **B1** (and **B2**, etc.) including **B1**
    - **B1** is an upper bound

  <T super D >

  - class **T** can be replaced only with types that are super classes of **D**, including **D**
    - **D** is a lower bound

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;
} } }
```


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,E extends Comparator<T>>
void sort(T v[], E 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;
 } } }
```

Bounded example

- T must be bounded to allow the compiler know which methods are available

```
public class Point<T extends Number> {
  T x; T y;
  public Point(T x, T y){
    this.x = x; this.y = y;
  }
  public double length(){
    return Math.sqrt(
     Math.pow(x.doubleValue(),2)
     + Math.pow(y.doubleValue(),2) );
  }
}
```


Generics subtyping

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

```
Pair<Integer> pi = new Pair<>(0,1);
Pair<Object> pn = pi;
pn.setFirst("0.5");
Integer i = pi.first();
```

if this were legal then...

.. 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_B**
  - Non-safe assumption!
- Invariance: elements inheritance does not imply container inheritance
  - Type safe assumption

Array covariance

- 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

Type 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; // if it were correct
pn.setFirst("0.5");// this would be possible
```

Type mismatch

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

- 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

Wildcards

- 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

Cannot be invoked with `Pair<Integer>`

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

Defines an upper bound for the type parameter

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

Unknown with upper bound
Equivalent but more compact

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


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

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

```
class MasterStudent extends Student{
  private String degree;
}
```

```
Pair<MasterStudent> pm={...};
sort(pm);
```

Method is not applicable for the argument: `MasterStudent` does not implement `Comparable<MasterStudent>`

Sorting a pair

```
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);
  }
}
```

Sort generic

```
 T extends Comparable<? super T>
MasterStudent        Student    MasterStudent
```

- Why <? super T> instead of just <T> ?
  - Suppose you define
    - **MasterStudent extends Student { }**
  - Intending to inherit the Student ordering
    - Does not implement **Comparable<MasterStudent>**
    - But **MasterStudent** extends (indirectly) **Comparable<Student>**

Sort method

- On Comparable objects:

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

  - For backward compatibility, actually 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)
```

TYPE ERASURE

Generics classes

- The compiler generates only one class for each generic type declaration
  - Compilation erases the types

```
Person<Integer> a = new Person<Integer>
          ("Al","A",new Integer(123));
Person<String> b = new Person<String>
                         ("Pat","B","s32");
boolean same=(a.getClass()==b.getClass());
```

believe it or not
same is *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
  - Erasure of a parameter is the erasure of its first constraint
    - If no constraint then erasure is `Object`
  - The erasure of a non-generic type is the type itself

Type erasure - examples

- In: `<T>`
  - `T` → `Object`
- In: `<T extends Number>`
  - `T` → `Number`
- In: `<T extends Number & Comparable>`
  - `T` → `Number`

Type erasure – consequences I

- Compiler applies checks only when a generic type is used, not within it.
- Whenever a generic or a parameter is used a cast is added to its erasure
- To avoid inconsistencies and wrong expectations
  - **`instanceof`** and **`.class`** cannot be used on generic types
  - valid for G<?> equivalent to the raw type

Type erasure – consequences II

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

```
class G<T> {
    T[] toArray(){
    T[] res = new T[n];
    T t = new T();
  }}
```

The compiler cannot instantiate these objects

  - It is not possible to substitute the erasure in an instantiation statement

Type erasure – consequences II

- 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 substituted in an instantiation statement

SOftEng http://softeng.polito.it

Type erasure– consequences III

- Overload and ovverride must be considered after type erasure

```
class Base<T> {
  void m(int x){}
  void m(T t){}          // T -> Object
  void m(String s){}
  <N extends Number> void m(N x){}   // <N extends Number> struck out, N -> Number
  void m(List<?> l){}    // <?> struck out
}
```

SOftEng http://softeng.polito.it

Type erasure– consequences IV

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

```
class Value implements Comparable<Value>
class ExtValue extends Value
           implements Comparable<ExtValue>
```


FUNCTIONAL INTERFACES WITH GENERICS

Functional Interfaces

- An interface with exactly one method
- The semantics is purely functional
  - The result of the method depends solely on the arguments
  - There are no side-effects on attributes
- Can be implemented as lambda expressions
- Predefined interfaces are defined in
  - **`java.util.function`**

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
  - **To*Type*Function**
  - **Type1ToType2Function**
- Suppliers: **TypeSupplier**
- Predicate: **TypePredicate**
- Consumer: **TypeConsumer**

Generic Comparator

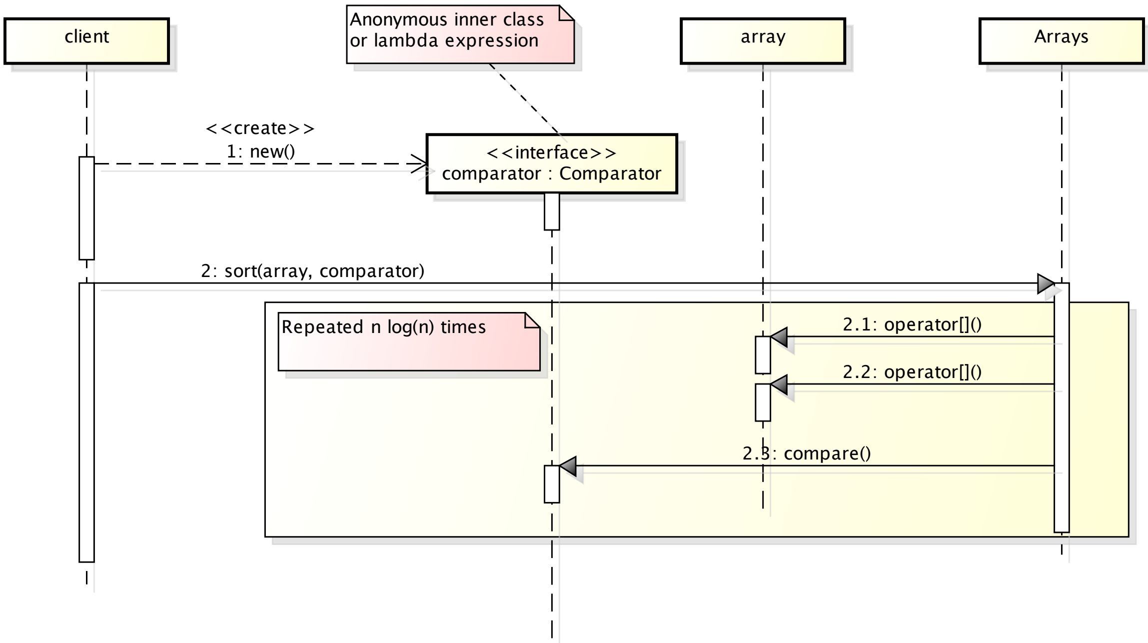
- Interface **java.util.Comparator**

```
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 the natural order (i.e. implement `Comparable`)

```
static <T,U extends Comparable<U>>
Comparator<T>
   comparing(Function<T,U> keyGetter)
```

Comparator.comparing()

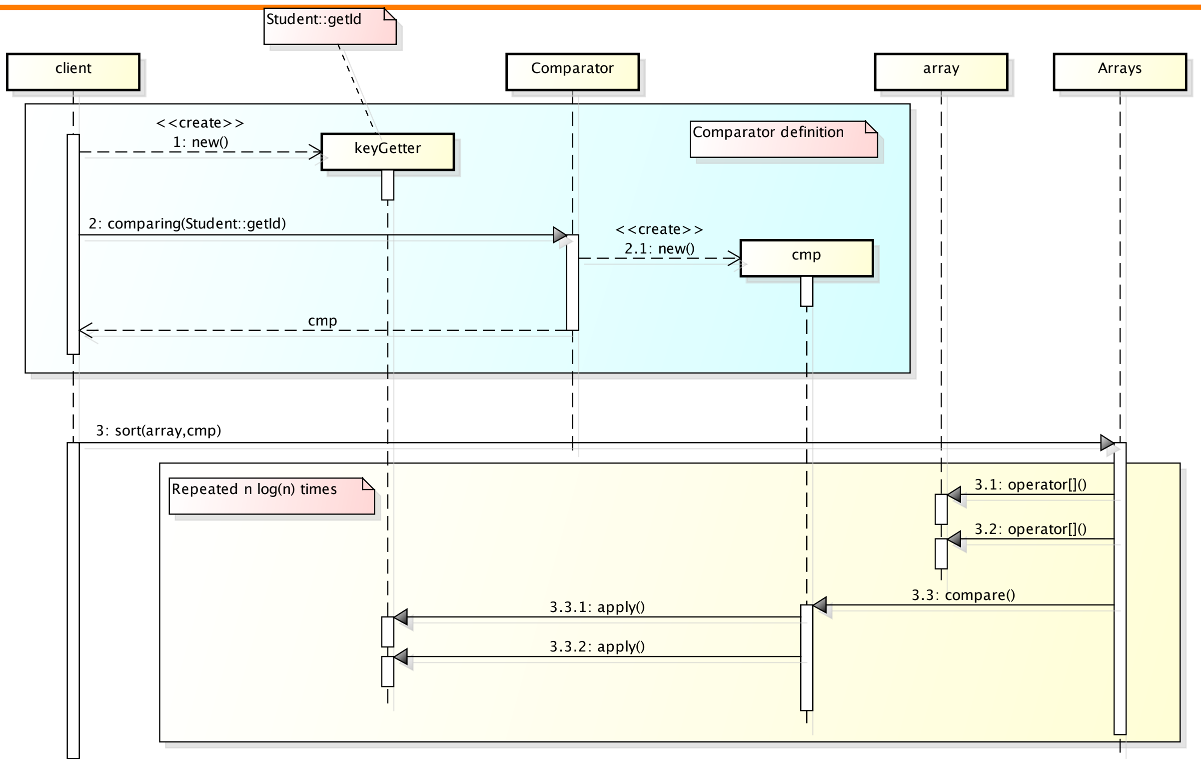
Comparator.comparing

```
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 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(){
 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

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

```
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(a,b);
}
```

```
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));
}
```

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


Performance

- Comparing
  - Anonymous Inner Class or Lambda Expression

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

  - Comparator.comparing + reversed

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

    - Requires 50% to 60% more time

Functional Interfaces Composition

- Predicate
  - `default Predicate<T> and(Predicate<T> o)`
  - `default Predicate<T> or(Predicate<T> o)`
  - `default Predicate<T> negate()`
- Function
  - `default Function<V,R> compose(Predicate<V,T> b)`

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