Technische Hochschule Rosenheim
Technical University of Applied Sciences

Object-oriented programming
Chapter 3 – Generics

Prof. Dr Kai Höfig

Motivation

*"**Reusability** is one of the great promises of **object-oriented** technology. Unfortunately, it's a promise that often goes unrealised. The problem is that reuse isn't free; it isn't something you get simply because you're using object-oriented development tools. Instead, it's something **you must work hard** at if you want to be successful."*
(Scott Ambler)

- **Generic, reusable code** through *interfaces, specialisation, polymorphism*, etc.
  - Conflict: flexibility vs. type safety.
  - Type check sometimes only at runtime.
- Which generic data structures and algorithms does the **Java Standard Library** already offer?

Without generics ("raw types")

- **Aim:**
  - Generic class `Bag` that can include any object as content.
- **Attempt:**

```
public class Bag {
 private Object content;
 public Bag(Object content) {
   this.content = content;}
 public Object getContent() {
   return content; }
 public void setContent(Object c){
   this.content = c;}
}
```

Using the class `Bag`:

```
Long bigNumber = 11111111111L;
Bag b1 = new Bag(bigNumber);
Bag b2 = new Bag("Hello");

// later on
Long val = (Long)b1.getContent();
String s = (String)b2.getContent();
```

No compiler error if b2 contains a Long!

- **Potential improvements**
  - When initialising, notify the compiler about which content type the instance of Bag is to be used for.
  - The compiler can then monitor this so that only the desired content type is really added.
  - When reading/accessing the content, we can be sure that the desired data type is in the Bag.

Generic classes

- **Declaration** of a *generic type T* for a class
  - "Parameterisation of a data type"
  - Always replace `Object` with `T`

- **Using** a generic data type
  - There are 2 *parameterised types* created with the *type parameters* Long and String.
  - No type casting is necessary!

```
public class Bag<T> {
 private T content;
 public Bag(T content) {
    this.content = content;
  };
 public T getContent() {
    return content;
  }
 public void setContent(T c) {
    this.content = c;}
}
```

```
Long bigNumber = 1111111111L;
Bag<Long> b1 = new Bag<Long>(bigNumber);
Bag<String> b2 = new Bag<String>("Hello");

// later on
Long val = b1.getContent();
String s = b2.getContent();
```

Note for the compiler, that here the placeholder T is assigned a type, which is fixed from then on.

Motivation for generics

- In the last two chapters we learned about the two data structures List and Set (which we implemented as a binary tree). While doing so, we chose the interfaces in such a way that they were fixed as specific data types:

Can we not implement this more generally so that a special data structure doesn't need to be implemented for each data type?

```
public interface IntList {
  // according to the [] operator:
  int get(int i);
  void put(int i, int v);

  // regarding the list length
  void add(int v);
  int remove(int i);

  int length();
}
```

```
public interface CharSet {
  void add(char c);            // add element
  boolean contains(char c);   // check if already included
  char remove(char c);         // remove element
  int size();                  // not "length", as there is no sequence
}
```

Example list

- Obviously, the structure of a list is independent of which specific elements are stored in it. Looking back: with the `IntList` we used a container element that had stored precisely one `int` value:

```
public class IntElement {
    int value;
    IntElement next;
    IntElement(int v, IntElement e) {
        value = v;
        next = e;
    }
}
```

The classes that do the same with `char`, `double`, `string` and all other classes would also look the same. Isn't there a shorter and better way?

- Now in Java, it is the case that all objects -- no matter what class -- are always also a `java.lang.Object`, which is the common base class. This means that if we use `Object` everywhere instead of the specific `int` type, then the list should work for all data types.

Attempted “generic” implementation

- We could simply just work with `Object` instead of a specific type.
- Why isn't this an optimal implementation?

```
public interface GenericList {
  void add(Object o);
    Object get(Object o);
  int remove(Object o);
  int length();
}
```

```
public class GenericListImpl implements GenericList {
  class Element {
        Object value;
        Element next;

        Element(Object o, Element e) {
      value = o;
      next = e;
        }
    }

    Element head;

  public void add(Object o) {
    if (head == null) {
      head = new Element(o, null);
      return;
        }

        Element it = head;
    while (it.next != null)
            it = it.next;

    it.next = new Element(o, null);
    }
…
}
```

Problems when implementing with `Object`: loss of type safety

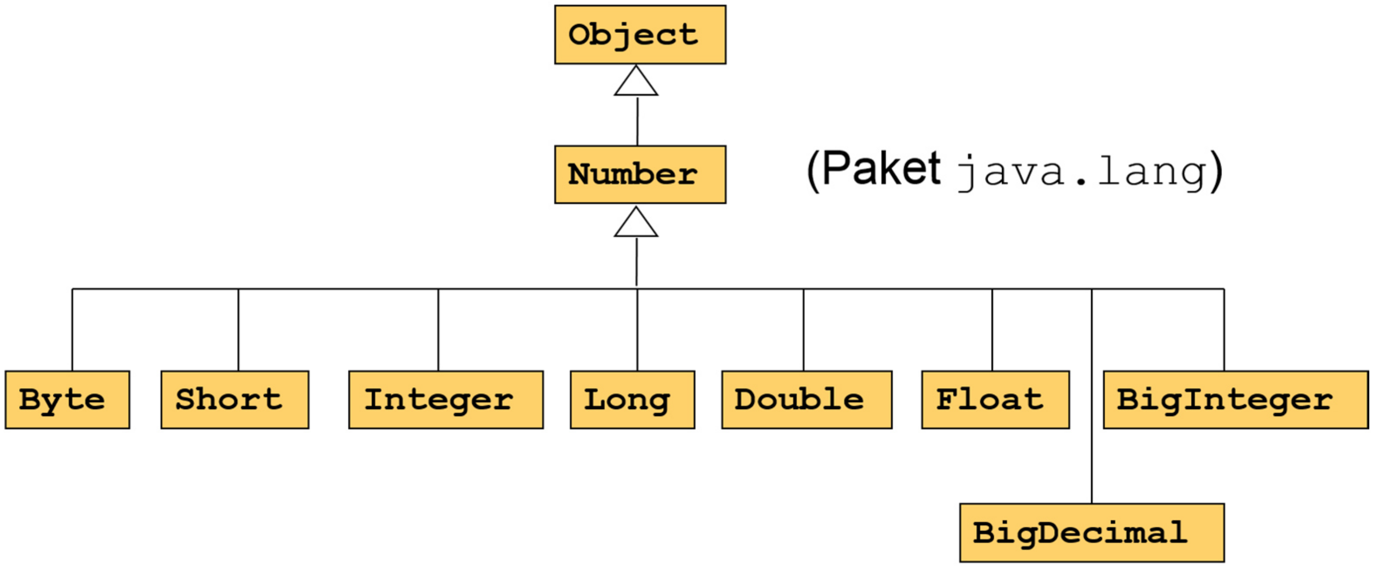
```
public class startupGenerics {
  public static void main(String[] args) {
    GenericList li = new GenericListImpl();

    li.add("Hello");  // OK: every string is also an object
    li.add("world");
    li.add(4); // also works. Do we want this? (type safety)
    li.add(4+""); // good, now it’s at least a  string again :/

    for (int i = 0; i < li.length(); i++)
      System.out.println(li.get(i));  // OK: every object can .toString()
        //string s = li.get(0);  // compiler error: Object is not string

    String hw = (String) li.get(0);  // OK: forced type conversion
    int i=(int) li.get(2);  // also works, but can also easily
                                  // go wrong, if there is a string there.
  }
}
```

- … this even works with primitive data types, if we use the **wrapper classes**

Wrapper classes

- For each primitive data type, there is a ***wrapper class***
  - Encapsulates primitive data types in an object.

- Why wrapper classes?
  - Provide static methods for conversion to strings and back.
  - Necessary for data structures of the class library (collections), which can only store objects.
  - Generics (see later) are only available with wrapper classes.



(Paket `java.lang`)

Wrapper classes

- Creating wrapper objects
  - with constructors
  - static ***valueOf*** methods
  - with ***boxing***
- All wrapper classes overwrite *equals()*
- Wrapper classes are **immutable**!
- **Autoboxing**
  - Automatic conversion between primitive data types and wrapper objects
- Operations without a wrapper class are sometimes more powerful!

```java
// Creation through constructors
Boolean    b = new Boolean(true);
 Character c = new Character('X');
 Byte        y = new Byte(1);
 Short       s = new Short(2);
 Integer     i = new Integer(3);
 Long        l = new Long(4);
 Float       f = new Float(3.14f);
 Double      d = new Double(3.14);

// Creation with valueOf
Long l1 = Long.valueOf(1000L)

// Creation with boxing
Integer i1 = 42;

int i2 = 4711;
// boxing -> j = Integer.valueOf(i)
Integer j = i2;
// unboxing -> k = j.intValue()
int k = j;
```

`java.math.BigInteger` class

- **Advantages**
  - Representation of numbers of any size
  - Numerous additional methods such as modular arithmetic

- Objects of the class are **immutable!**
  - `public BigInteger(String val)`
  - `public BigInteger(String val, int radix)`
  - `static BigInteger valueOf(long val)`

- ***Class-specific constants***
  - `BigInteger.ZERO`
  - `BigInteger.ONE`
  - `BigInteger.TEN`

- Numerous methods for arithmetic calculations, comparisons, conversion to primitive data types
  - https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html

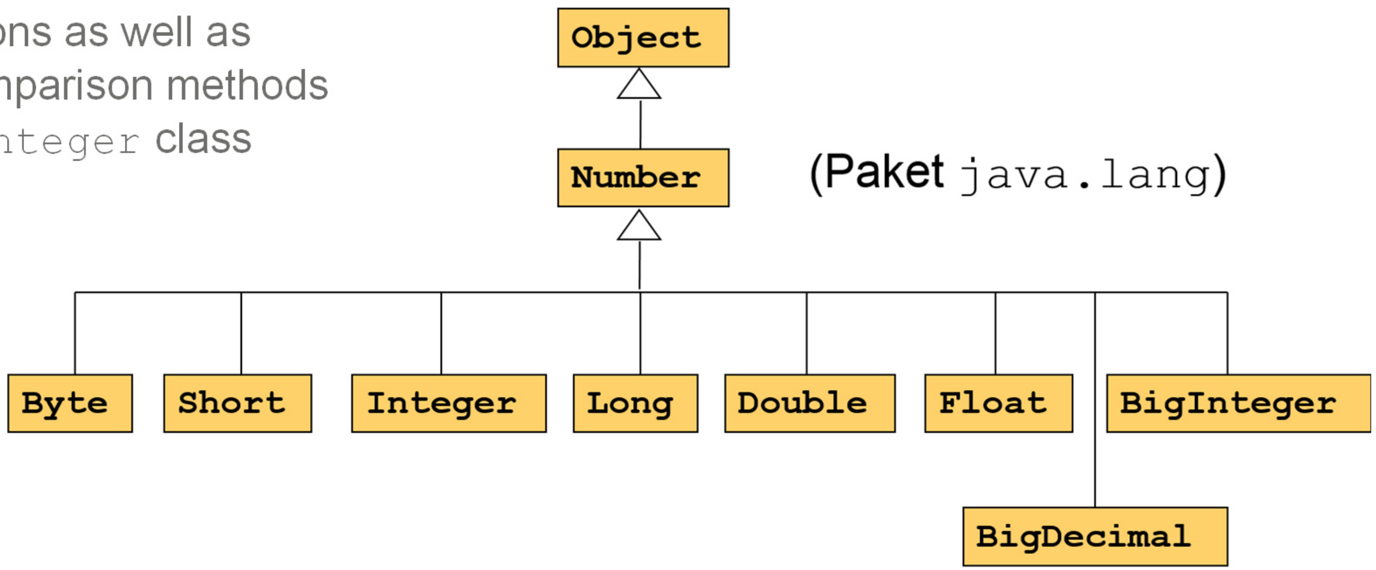
`java.lang.BigDecimal` class

- Representation of **any exact floating-point numbers**
  - https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html

- Consist of
  - a sequence of numbers (object of type `BigInteger`) and
  - a scaling (number of decimal places)

- Objects of the class are **immutable.**

- Methods for carrying out
  - arithmetic calculations as well as
  - conversion and comparison methods
  - similar to the `BigInteger` class



What's wrong here?

```
BigInteger  big   = new BigInteger("12345678901234567890 12");
BigInteger  small = BigInteger.valueOf(25000);

String  s  = small.toString();  // "25000"
String  t  = small.toString(7); // to base 7: "132613"

big = big.add (small);          // adds small "to" big

BigDecimal bd1 = new BigDecimal(big);
BigDecimal bd2 = new BigDecimal(3.14);
BigDecimal bd3 = new BigDecimal("3.14");

bd2.add(bd3);
```

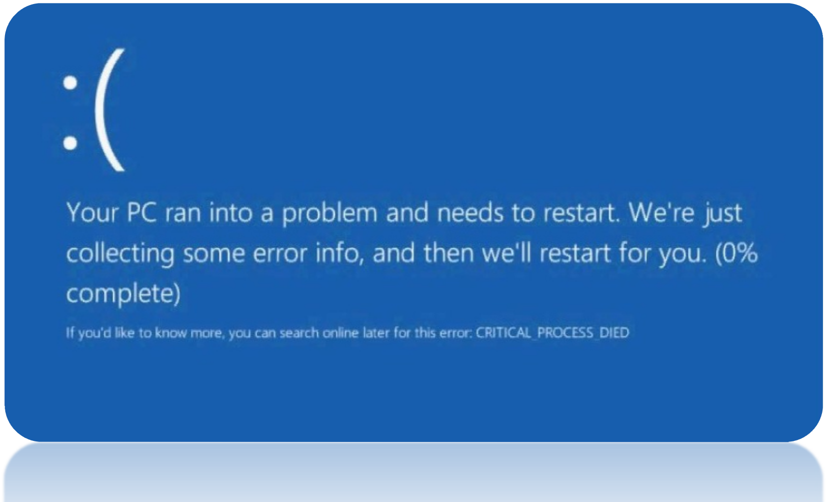
What's wrong here?

```
BigInteger  big   = new BigInteger("12345678901234567890 12");
BigInteger  small = BigInteger.valueOf(25000);

String  s   = small.toString();  // "25000"
String  t   = small.toString(7); // to base 7: "132613"

big = big.add (small);           // adds small "to" big

BigDecimal bd1 = new BigDecimal(big);
BigDecimal bd2 = new BigDecimal(3.14);
BigDecimal bd3 = new BigDecimal("3.14");

bd2.add(bd3);
```

bd2 remains unchanged by addition. A new result object is created, which must be assigned. Correct would be, for example
bd2 = bd2.add(bd3)

Errors at runtime and errors at translation time

- We have seen that it is possible to implement generic classes and methods by using `Object`.
  - All classes are derived from `Object`
  - There are **wrappers** for primitive data types
  - **A generic implementation makes our methods and classes (even more) reusable**
- **But**: the type safety is violated, which can lead to **runtime errors**.
- **Runtime errors are particularly critical** because they occur during the runtime, i.e. at the customer. So we want to avoid runtime errors at all costs
- By using ***Java generics*** we eliminate runtime errors, and already notice when compiling a programme whether there are any type errors.



Generics

- Instead of working with `Object`, we now insert a type parameter (a ***generic***)

- For example, we have a parameter list of a method, here the parameter, or placeholder i:

```
public interface IntList {
  int get(int i);
..}
```

- With ***Java generics*** , classes can now also be assigned one or more parameters, often `T` for type, but this is not mandatory:

```
public class myClass<Param1,Param2> {
  private Param1 element1;
  private Param2 element2;
}
```

- These parameters are then passed when objects are created, similar to the parameters of a constructor. But in contrast with a constructor, these parameters are types:

```
public class startupMyClass {
  public static void main(String[] args) {
    MyClass<Integer,String> myObject = new MyClass<Integer, String>();
    }
}
```

- The generics must be of the type `Object` (wrapper classes!)

What are the benefits? Example `List`

- We can only assume that `T` is of the type `Object`, but we can still do a lot with it.
- If other types are now used as an `Integer`, e.g. `String`, there is a compiler error
  → **type safety**
  → **but still a high degree of reusability**
- **Disadvantage**
  In the generic class, i.e. ListImpl<T>, we do not know the subsequent type and cannot do much

```java
public class startupGenerics {
  public static void main(String[] args) {
        List<Integer> li = new ListImpl<Integer>();
    li.add(1);
    int a = li.get(0);  // no cast required anymore

    li.add("Hans");  // Compiler error!
  }
}
```

```java
class ListImpl<T> implements List<T> {
  private class Element {
    T value;  // previously Object
    Element next;
        Element(T o, Element e) {  // !
      value = o;
      next = e;
        }
    }

  private Element head;

  public T get(int i) { … }

  public void add(T v) {  // !
    if (head == null) {
      head = new Element(v, null);
      return;
        }

        Element it = head;
    while (it.next != null)
            it = it.next;
    it.next = new Element(v, null);
    }
}
```

Yes, there are restrictions, but what is possible?

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Since with generics, when developing we can (initially) only assume that it is an `Object`, we are rather restricted to just the methods of `Object`
  https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

- An extract of the methods of `Object`:

- `clone()`
- `equals(Object obj)`
- `getClass()`
- `hashCode()`
- `toString()`

- Now we look at `toString`, `equals` and also

- `Comparable`
  https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html
- `Comparator`
  https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

`toString()` method of the `Object` class

- **`public`** `String toString()`
  - Creates a string representation of the object.
  - Returns a string that describes an object in readable form

- **Implementation in `Object` class:**
  - `getClass().getName() + '@' + Integer.toHexString(hashCode())`
  - <ClassName>@<HashValue/ObjectID>

- **Recommendation:**
  - For own classes: overwrite method!

```java
public class Person {
  private String name;

  public Person(String name) {
    this.name = name;
    }

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

What are the benefits?
Example `List<T>`

- Now we can create a generic output.
- If the type we use to create the list has a useful implementation of toString, this results in a good output.
- Otherwise, we only see class names and IDs

```
class ListImpl<T> implements List<T> {
..
  @Override
  public String toString(){
    if (head == null) {
      return "";
        }

        Element it = head;
        String output=it.value.toString();
    while (it.next != null){
            it = it.next;
            output += " "+it.value.toString();
        }
    return output;
    }
}
```

```
public class startupGenerics {
  public static void main(String[] args) {
        List<Integer> li = new ListImpl<Integer>();
    li.add(1);
    li.add(2);
    li.add(3);
    System.out.println(li); // 1 2 3
  }
}
```

`equals(.)` method of the `Object` class

- Check the **identity**: "==" or "!="
  - In the case of *primitive data types*, this means that the integer values match.
  - In the case of *non-primitive (reference) data types* (objects), this only means that references (= addresses in memory) match.

- **Equality** of objects referred to by non-primitive (reference) types: `equals`

- Standard implementation in `Object` class:

```
@Override
public boolean equals(Object obj) {
  return (this == obj);
}
```

```
public class startupPerson {
  public static void main(String[] args) {
        Person p1= new Person("Klaus");
        Person p2= new Person("Maria");
        Person p3 = new Person("Klaus");

        System.out.println(p1.equals(p2)); // false
    System.out.println(p1.equals(p3)); // true
    System.out.println(p1==p3);         // false
  }
}
```

```
public class Person {
  private String name;
..
  @Override
  public boolean equals(Object obj){
    // (1) compare with 0
    if (obj == null) return false;
    // (2) check identity
    if (obj == this) return true;
    // (3) check if same data type
    if (!this.getClass().equals(obj.getClass())) {
      return false;
        }
    // type cast and compare
    Person p = (Person) obj;
    return this.name==p.name;
    }
}
```

What are the benefits? Example `List<T>`

```
class ListImpl<T> implements List<T> {
..
  public boolean checkPure(){
    if (head == null) {
      return true;
        }

        Element it = head;
    while (it.next != null) {
      if(!(it.value.equals(it.next.value))){
        return false;
            }
            it = it.next;
        }
    return true;
    }
}
```

- Now we can compare in generic classes (provided that `equals` is implemented correctly).

```
public class startupGenerics {
  public static void main(String[] args) {
        List<Integer> li = new ListImpl<Integer>();
    li.add(1);
    li.add(2);
    li.add(3);
    System.out.println(li.checkPure()); // false

    List<Integer> li2 = new ListImpl<Integer>();
        li2.add(1);
        li2.add(1);
    System.out.println(li2.checkPure()); //true
  }
}
```

Interfaces: `Comparable` and `Comparator`

- **Natural order:** `interface Comparable<T>`
  - `public int compareTo(T o)`
  - If the class implements this interface, then objects of the class are comparable.
  - However, those who programme the class determine "how" to compare.
  - Example: a room is compared to another room in terms of number of seats.

- **Further order:** `interface Comparator<T>`
  - `public int compare(T o1, T o2)`
  - Necessary if there are ***multiple different comparison criteria*** for objects.
  - Example: “rooms" should be sorted once by number of seats and once by size in square metres.

- **Result** respectively:
  - <0 if the current or left object is smaller.
  - >0 if the current or left object is larger.
  - 0 if there is "equality."

So objects are in the correct order

```
"A".compareTo("B") = -1
```

- Generic code
  - Example: sorting algorithms work on all classes that implement the `Comparable` interface.

Example Person

```
public class Person implements Comparable<Person>{
..
  @Override
  public int compareTo(Person o) {
    return this.name.compareTo(o.name);
    }
}
```

Comparison without generics:

```
public class PersonOG implements Comparable{
    String name;
  @Override
  public int compareTo(Object o) {
    PersonOG other = (PersonOG)o; // Type safe?
    return this.name.compareTo(other.name);
    }
}
```

```
public class startupPerson {
  public static void main(String[] args) {
        Person p1= new Person("Klaus");
        Person p2= new Person("Maria");
    System.out.println(p1.compareTo(p2)); // -2
  }
}
```

- *We can now determine our own order here. In this case lexicographically.*

Generic bounds

…because we can't know if the type implements the `Comparable` interface

- If we now want to specify that our generic class may only be called with types that implement the `Comparable` interface, we use a ***bound*** in the definition of the generic class.

- Example `List`

```
interface List<T extends Comparable<T>> {
  void add(T o);
  T get(int i);
  boolean checkPure();
}
```

- The Set interface should therefore be generic in `T`, but only for those `T` which implement the `Comparable<T>` interface. If several such restrictions are necessary, you can use `&` (not a comma!) to create a series, e.g. `<T extends Comparable<T> & Serializable>`.

Example Comparator `Person`

```java
public class PhonebookPersonComparator implements Comparator<Person> {
  @Override
  public int compare(Person o1, Person o2) {
    return o1.toString().toLowerCase().compareTo(o2.toString().toLowerCase());
    }
}
```

```java
public class startupPerson {
  public static void main(String[] args) {
        Person p1= new Person("Klaus");
    Person p4 =new Person("klaus");

    System.out.println(p4.compareTo(p1)); // 32

    PhonebookPersonComparator ppc = new PhonebookPersonComparator();
    System.out.println(ppc.compare(p1,p4)); //0

  }
}
```

Why is `compare()` not `static`?
Short answer: because Comparator is old.

https://stackoverflow.com/questions/21817/why-cant-i-declare-static-methods-in-an-interface?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa

- With `Comparator`, different comparison operations can be implemented.
- With `Comparable`, a natural order is implemented, and with `Comparator`, other special cases

What are the benefits? Generic algorithms

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

  public static <T> void sort(T[] a, Comparator<T> comp) {
    for (int i = 0; i < a.length; i++) {
      for (int j = i + 1; j < a.length; j++) {
        int c = comp.compare(a[j], a[i]);
        if (c < 0) {
          T h = a[i];
                a[i] = a[j];
                a[j] = h;
              }
            }
          }
    }
}
```

- Example Sort: the `Sort` class can sort arrays of any type, as long as either a matching `Comparator` is provided, or the elements of the array are `Comparable` themselves. Great, right?

```
public class startupPerson {
  public static void main(String[] args) {
    Person[] pa = {p1,p2,p3,p4};
    Sort.sort(pa); // natural sorting
    Sort.sort(pa,ppc); // like in the telephone book
  }
}
```

Summary

- Generics give us type safety when reusing code for different types
- When using them, either we are restricted to the methods of `Object`, or we restrict the generic implementations for certain types by means of bounds. Seen in the example of `Comparator` and `Comparable`
- Generics allow us to use generic algorithms.