Java Generics

Th. Letschert
Fachhochschule Giessen-Friedberg, Giessen, Germany
University of Applied Sciences
Generics: Generic Classes and Methods
Generic Classes and Methods

**Generic Class : Class with types as parameters**

Form:

```java
class C < ... generic parameter(s) ... > .... { ... }
```

```java
public class Stack<T> {
    private List<T> a = new ArrayList<T>();

    public void push( T x ) {
        a.add(x);
    }

    public T pop () {
        if ( a.isEmpty() )
            throw new IllegalStateException();
        else
            return a.remove(0);
    }
}
```

```java
Stack<String> myStack = new Stack<String>();
```

UML Diagram:

- Stack<T>
  - push(T): void
  - pop(): T

**Definition**

**Use**
Generic Classes and Methods

Generic Method : Method with types as parameters

Form:

\[ \text{attributes} < \ldots \text{generic parameter(s)} \ldots > \text{method-definition} \]

```java
class GenMethod
    public static <T> void swapFirstLast (T[] a) {
        T temp = a[0];
        a[0] = a[a.length-1];
        a[a.length-1] = temp;
    }
}

String[] a = { "Hello", "Java" };
swapFirstLast(a);
```

Definition

```
String[] a = { "Hello", "Java" };
GenMethod.swapFirstLast(a);
```

Use in same class

```
String[] a = { "Hello", "Java" };
GenMethod.<String>swapFirstLast(a);
```

Use in different class (2)
Generic Classes and Methods

Generics and Interfaces

- Interfaces may be defined as generics
- Interfaces may be type parameters

```java
public interface Stack<T> {
    void push(T x);
    T pop();
}
```

```java
public class ListStack<T> implements Stack<T> {
    ...
}
```

```java
Stack<Stack<String>> st = new ListStack<Stack<String>>();
st.push(new ListStack<String>());
```

A generic interface

Implementation of a generic interface

A Stack of Stacks of Strings
Generics: Restrictions

Instantiation
of objects with generic type: Not possible
- Instantiation of generic parameter type is not allowed

Static Methods
of objects with generic type: Not possible
- Static methods of objects with a generic type my not be accessed

Overloading
of methods based on generics: Not possible
- Overloaded methods must differ in more than a generic type

```java
class C<E> {
    ... new E() ... 
}

class C<E> {
    ... E.m() ... 
}

void f(List<String> x) {}
void f(List<Integer> x) {}

class C<T1, T2> {
    void f(T1 x){}
    void f(T2 x){}
}
```
Generics: Restrictions

Generics and Arrays do not live in complete harmony:

- arrays of generic types may not be allocated
- Solution: allocate Object-arrays and cast them

```java
public class ArrayStack<T> implements Stack<T> {
    @SuppressWarnings("unchecked")
    private T[] a = (T[]) new Object[10];

    private int size = 0;

    public T pop() {
        if (size <= 0) throw new IllegalStateException();
        return a[--size];
    }

    public void push(T x) {
        if (size >= 10) throw new IllegalStateException();
        a[++size] = x;
    }
}
```

Type safety of generics is guaranteed by the compiler, as long as there are no casts. With casts there may be type errors at runtime – so this warning. It is suppressed because we know what we do!
Generics: Restrictions

Generics and Arrays
do not live in complete harmony:

- arrays of parametric instantiations may not be allocated
- Solution: allocate Object-arrays and cast them

```java
@ SuppressWarnings("unchecked")
public static void main(String[] args) {
    List<String>[] a = (List<String>[]) new Object[10];
    a[0] = new ArrayList<String>();
    a[1] = new LinkedList<String>();
}
```
Generics

Kinds of Genericity

- **Java Generics ~ Parametric Polymorphism**
  Flexibility and adaption by generating definitions

- **Genericity by Inheritance**
  Flexibility and adaption by generalization

```java
class Cage<T> {
    void lockUp(T x) {
        ....
    }
}
Cage<Tiger> c1 = new Cage<Tiger>();
c.lockUp( new Tiger() );
Cage<Lion> c1 = new Cage<Lion>();
c.lockUp( new Lion() );
```

```java
class Cage {
    void lockUp(Object x) {
        ....
    }
}
Cage c = new Cage();
c.lockUp( new Tiger() );
c.lockUp( new Lion() );
```
Generics

The 2 Dimensions of Genericity

Inheritance:  type ~ subtype
Parametric:  type parameter

Inheritance

Cage<\text{Student}>  Cage<\text{Tiger}>  Cage<\text{Professor}>

Animal

SibirianTiger

parametric
Implementation of Generics: type erasure

**Implementation of generics by type erasure**

- Java generics are realized by mapping parametric genericity to inheritance genericity
- The compiler removes all type parameters / arguments and replaces them by casts / base types
- Raw types (free of type parameters/arguments) appear in class files

```java
class Cage<T> {
    void lockUp(T x) {....}
    T free() {....}
};

Cage<Tiger> cage = new Cage<Tiger>();
Tiger t = new Tiger();
cage.lockUp(t);
...
t = cage.free();
```

```java
class Cage {
    void lockUp(Object x) {....}
    Object free() {....}
};

Cage cage = new Cage();
Tiger t = new Tiger();
cage.lockUp(t);
...
t = (Tiger)cage.free();
```
Implementation of Generics: casts by the compiler

**Type erasure and type errors**

- Casts added by the compiler will never fail
- ... if it doesn't issue a warning

```java
class Cage {
    void lockUp(Object x) {...}
    Object free() {...}
};
...
t = (Tiger)cage.free();
```

*code after type erasure*

```java
@SuppressWarnings("unchecked")
public static void main(String[] args) {
    List<String>[] a = (List<String>[]) new Object[10];
    a[0] = new ArrayList<String>();
    a[1] = new LinkedList<String>();
}
```

fail-proof cast: compiler assures success of its casts

no assurance for success by the compiler
Implementation of Generics: Arrays

**Type erasure and arrays**
- Arrays carry type information at runtime
- Because: arrays are covariant in their element-type
  \[ \text{Cow<Animal} \Rightarrow \text{cow[]<Animal[]}, \]
  but this may be dangerous

```java
static void f(Animal[] l){
    l[0] = new Tiger();
}
public static void main(String[] args){
    Cow[] cows = new Cow[2];
    ...
    f(cows);
}
```

ArrayStoreException

\[ l \] is tagged with **Cow**

Tiger is not compatible with Cow
Implementation of Generics: Arrays

Type erasure and arrays
- Arrays carry type information at runtime
- ... but this can only be a raw type

```java
public class Stack<T> {
    private T[] a = new T[10];
    ...
}
```

The compiler can't associate a raw type with T. Array creation is not possible

```java
public static void main(String[] args) {
    List<String>[] a = new List<String>[10];
}
```

The compiler can't associate a raw type with List<String>. Array creation is not possible
Implementation of Generics: Arrays

Type erasure and arrays
- Arrays carry type information at runtime
- ... but this can only be a raw type

```java
public class ArrayStack<T> implements Stack<T> {

    private final T[] a = (T[]) new Object[10];
    private int size = 0;

    public T pop() {...}
    public void push(final T x) { ... }

    @SuppressWarnings("unchecked")
    public T[] toArray() {
        final T[] res = (T[]) new Object[size];
        for ( int i = 0; i < size; i++ ) { res[i] = a[i]; }
        return a;
    }
}

public static void main(String[] args) {
    ArrayStack<Integer> stack = new ArrayStack<Integer>();
    stack.push(5);
    Integer[] ar = stack.toArray();  // ClassCastException !
}
```
Bounded Type Parameters
Bounded Parameters

Bounded parameters:
A generic parameter may be restricted

```java
public class Stable<T extends Animal> {

    @SuppressWarnings("unchecked")
    T[] box = (T[]) new Object[10];

    public void insert(T t, int i) {
        box[i] = t;
    }
    ...
}
```

Example: stables of different types

```java
public static <T extends Comparable<T>>
    T max(T x, T y) {
        return x.compareTo(y) > 0 ? x : y;
    }
```

Example: generic maximum
Bounded Parameters

Bounded parameters:
A generic parameter may be restricted

```java
public class Stable<T extends Animal> {

    @SuppressWarnings("unchecked")
    T[] box = (T[]) new Object[10];

    public void insert(T t, int i) {
        box[i] = t;
    }

    ...
}
```

Example: stables of different types

```java
public static <T extends Comparable<T>>
T max(T x, T y) {
    return x.compareTo(y) > 0 ? x : y;
}
```

Example: generic maximum
Bounded Parameters

```java
public interface PriorityQueue<E extends Comparable<E>> extends Iterable<E> {
    public void enqueue (E e);
    public E dequeue() throws QueueEmptyException;
}
```

Example: Generic Interface with bounded parameter that implements a generic interface

- The bounded type (E) appears in its bound (Comparable<E>) (F-bound genericity)
- The bounded type appears in the implementation clause
Bounded Parameters

Bounded parameters : &
A generic parameter may be restricted by several bounds

```java
public static <T extends Toy & Quaker>
void goSwimming(T t) {
    ...
}
```

Example: two bounds.
Go swimming with something that is a toy and can quake.

```java
public interface Toy {
    void play();
}
```

```java
public interface Quaker {
    void quake();
}
```

```java
public class RubberDuck implements Toy, Quaker {
    public void play() { ... }
    public void quake() { ... }
}
```
Wildcards

Wildcard Extends

Wildcard Super
**Wildcard Extend**

**Subtype relation**
- $T < T'$ : T-Objects may substitute T'-Objects
- $T < T'$ : T extends / implements T'

**Subtype relation and generics**
- **Covariance on collection type:**
  
  $$\text{ArrayList} < E > < \text{List} < E > < \text{Collection} < E > < \text{Iterable} < E >$$

- $T < \text{Object}$, for every reference type T
- **Collections take Objects of subtypes :**
  
  $$\text{Collection} < E > \ c;$$
  $$c.\text{add}(x)$$ for every $U x$; with $U < E$

- **Generic classes: neither covariant nor contravariant on argument types:**
  
  $$T < U \quad \text{Collection} < T > < \text{Collection} < U >$$
  $$\text{Collection} < T > > \text{Collection} < U >$$
Wildcard Extend

\[
T < U \quad \not\Rightarrow \quad \text{Collection}\langle T \rangle < \text{Collection}\langle U \rangle
\]

```java
static void f(List<Animal> animals){
    animals.add( new Tiger() ); // OK (for the compiler)!
}

public static void main(String[] args){
    List<Cow> cows = new LinkedList<Cows>();
    f(cows); // Compiler Error-Message
    ... assume f contains cows ...
}
```

*No covariance: This restrictions prevents from type errors.*

*however...*
**Wildcard Extend**

\[ T < U \implies \text{Collection}\langle T\rangle < \text{Collection}\langle U\rangle \]

```java
class Animal {
    void feed(Feed feed) {
        // feed the animal
    }
}

class Feed {
    void feed() {
        // functionality
    }
}

class Cow extends Animal {
    // Cow implementation
}

class Horse extends Animal {
    // Horse implementation
}

class AnimalList<T> {
    List<T> animals;
    void feed() {
        for (T animal : animals) {
            animal.feed(new Feed());
        }
    }
}

public static void main(String[] args) {
    List<Cow> cows = new LinkedList<Cow>(){
        AnimalList<Cow> animalList = new AnimalList<Cow>(){
            for (Cow cow : cows) {
                cow.feed();
            }
        };
        animalList.feed();
    };// Compiler Error-Message
}
```

This restrictions prevents lots of correct programs from being compiled.
What exactly might cause typing errors, if covariance were accepted?

```java
static void f (List<Animals> animals ){
    for( Animal A : animals )
        a.feed(new Feed());
    a.add( new Tiger() );
}

public static void main(String[] args){
    List<Cow> c = new LinkedList<Cow>();
    ...
    f(c);
    ... assume c contains cows ...
}
```

Reading elements from the collection and treating them as animals is OK.

Writing to the collection might cause a problem.

So: accept covariance and allow reading, prevent writing.
Wildcard Extend <\ ? extends T>:

- Accept covariance
  \[ T < U \Rightarrow \text{Collection}<T> < \text{Collection}<U> \]
- Accept reading operations
- Do not accept writing operations (except \texttt{null})

```java
static void f (List<\ ? extends Animal> animals ){
    for( Animal a : animals )
        a.feed(\new Feed());    \// OK !
    animals.add(\new Tiger()); \// Compiler Error-Message
    animals.add(\new Animal()); \// Compiler Error-Message
    animals.add(\null );        \// OK !
}
```

```java
public static void main(String[] args){
    List<Cow> c = \new LinkedList<Cow>();
    ....
    f(c); \// OK !
    ... assume c contains cows ...
}
```
Wildcard Extend

Wildcard Extend  

– Example 2

```java
public void f(Stack<? extends Animal> s) {
    s.push(new Cow());  // Compiler Error-Message
    Animal a = s.pop();  // OK !
    Cow c = s.pop();  // Compiler Error-Message
}

public static void main(String[] args) {
    ArrayStack<Cow> stack = new ArrayStack<Cow>();
    stack.push( new Cow() );  // OK !
}
```
Wildcard Extend

The Contract of Wildcard Extend \(< ? \text{ extends } T>\)

– Accept collections of subtypes of T-elements
– Will not add anythings except perhaps \texttt{null}

```java
static void f(List<? extends Animal> a) {
    tread them as animals or objects
}
```

**Precondition:**
Only take arguments of type \(\text{List}<T>\text{ with } T < \text{Animal}\)

**Postcondition:**
Do not insert anything (except perhaps \texttt{null}) tread all elements as animals (or a superclass of Animals)
Wildcard Extend

Wildcard Extend of Object \(<\ ? \text{extends} \ Object> \equiv <\ ?>\)

\(-<\ ? \text{extends} \ Object>\) has a special notation: \(<\ ?>\)

\(-<\ ?>\): tread elements as objects

```java
static void printAll(Collection<?> c){
    for( Object o: c)
        System.out.println(o);
}

public static void main(String[] args){
    List<Cow> cows = new LinkedList<Cow>();
    cows.add( new Cow() );
    printAll(cows);
}
```
Wildcard Extend

Wildcard Extend of Object  

– `<? extends Object>` / `<?>` is not the same as  `<Object>`

– `<?>` : covariance OK, no writing (except for null)

– `<Object>`: treat as Objects

```java
static void printAll_1( List<? > c ){
    for( Object o: c )
        System.out.println(o);
    c.add( new Tiger() );  // ERROR
    c.add( null );         // OK
}
```

```java
List<Cow> cows = new ArrayList<Cow>();
List<Animal> animals = new ArrayList<Animal>();
printAll_1(cows);         // OK
printAll_1(animals);      // OK
```

```java
static void printAll_2( List<Object> c ){
    for( Object o: c )
        System.out.println(o);
    c.add( new Tiger() );  // OK
    c.add( null );         // OK
}
```

```java
printAll_2(cows);         // Error
printAll_2(animals);      // Error
```
Wildcard Extend of Object < ?>
An example from the Java-API

```java
java.util.Interface Collection<E>

boolean containsAll ( Collection<?> c )

Returns true if this collection contains all of the elements in the specified collection.

Checks whether all elements of a different collection are contained in this collection

List<Animal> animals = new LinkedList<Animal>();
List<Cow> cows = new LinkedList<Cow>();
if ( cows.containsAll( animals ) ) {
    ....
} elseif ( cows.containsAll( animals ) ) {
    ....
}

boolean containsAll ( Collection<?> animals )

reading access to argument animals: OK,
no binding E ~> Cow / Animal : all collections are accepted
```
Example: test of containment in own collections

**Version A**: according to `java.util`

**Version B**: better? / more reasonable?

**Version A is more liberal**
(compatible to Java 1 - 1.4)
Some errors are not recognized.

**Version B is more restrictive**
Some correct programs are rejected.
Wildcard Extend

Wildcard Super : `<? extends Animal>`

- **Reading access to the Collection: OK**
  - `Collection<? extends Animal>`
    - A collection of beings that are animals or more specific than animals
    - You can assume any feature that animals have

- **Writing access to the collection is not allowed: except for null**
  - `Collection<? extends Animal>`
    - A collection of beings that are cows or more specific than cows
    - No being should be inserted except null. **null** is allowed because it is of any type, even to most specific subtype of Animal. Nothing else is allowed because nothing can have all features that any subtype of animal might have. (Remember, evolution didn't stop, nature will create new species for ever.)
Java Generics

Wildcard Super
T < U  \not\Rightarrow \text{Collection<T> > Collection<U>}

```java
static void f(List<Cows> cows){
    Milk milk = cows.milk();  // OK (for the compiler)!
}

public static void main(String[] args){
    List<Animals> animals = new LinkedList<Animals>();
    animals.add( new Tiger() );
    f(animals);  // Compiler Error-Message
}
```

No contravariance: This restrictions prevents from type errors.

however...
Wildcard Super

\[ T < U \neq \implies \text{Collection}\langle T \rangle \rightarrow \text{Collection}\langle U \rangle \]

```java
static void f(List<Cows> cows){
    cows.add( new Cow() ); // OK
}

public static void main(String[] args){
    List<Animals> animals = new LinkedList<Animals>();
    ...
    f(animals); // Compiler Error-Message
}
```

No contravariance: This prevents some legal programs from compiling.
What exactly might cause typing errors, if contravariance were accepted?

```java
static void f(List<Cow> cows){
    cows.add( new Cow() );
    Milk milk = cows.get(0).milk();
}

public static void main(String[] args){
    List<Animals> animals = new LinkedList<Animals>();
    animals.add( new Tiger() );
    f(animals);
}
```

So: accept contravariance and allow writing, prevent reading.
Wildcard Super

Wildcard Super <\? super T>
- Accept contravariance
  \( T < U \implies \text{Collection}<T> > \text{Collection}<U> \)
- Accept writing operations
- Do not accept reading operations (except for type \text{Object})

```java
static void f(List<\? super Cow> cows){
cows.add( new Cow() );
Cow cow = cows.get(0);
Object o = cows.get(0);
}

public static void main(String[] args){
  List<Animals> animals = new LinkedList<Animals>();
  animals.add( new Tiger() );
f(animals);
}
```

Writing elements to the collection is OK.
No reading access except for type \text{Object}
Wildcard Extend

The Contract of Wildcard Super \(< \ ? \ super \ T >\)

– Accept collections of supertypes of T
– Will not perform anythings except perhaps as \textbf{Object}

\begin{center}
\textbf{Precondition:}
Only accept lists of type \texttt{List<T>} with \( T > \text{Cow} \)
\end{center}

\begin{center}
\textbf{Postcondition:}
Did nothing but:
- Deletion
- Insertion of Cows or subtypes of \texttt{Cow}, down to \texttt{null}
- treat elements only as Objects
\end{center}
Wildcard Super

Wildcard Super: `<? super Cow>`

- Writing access to the Collection: OK
  - `Collection` `<? super Cow>`
    - A collection of beings that are cows or more general than cows
    - A cow or a subtype of Cow or null may be inserted
    - Deletions within the collections are allowed

- Reading access to the collection is not allowed: except for treating the element that was read as an `Object`
  - `Collection` `<? super Cow>`
    - A collection of beings that are cows or more general than cows
    - Do not assume any feature, except those that all beings (Objects) have
Wildcard Super and Wildcard Extends

- Wildcard Extends  
  `<? extends T>`  
  T and its super-types

- Wildcard Super  
  `<? super T>`  
  T and its sub-types

Everbody has Object's features

T or a Supertype, can be written with T's or subtypes of T. Can be treated as Objects

null belongs to any type
Wildcard Super and Wildcard Extends

Wildcard Extends and Wildcard Super
an example from the Java-API

\[
\text{java.util Class Collections}
\]

\[
\text{static } \langle T \rangle \ \text{void copy(List}\langle? \ \text{super } T \rangle \ \text{dest, List}\langle? \ \text{extends } T \rangle \ \text{src})
\]
\[
\text{Copies all of the elements from one list into another.}
\]

\[
\text{src is read}
\]
\[
\text{dest is written}
\]
Wildcard or Named Type Parameter

Wildcards may replace named type parameters

```java
public class Container<T> {
    T x1;
    T x2;
    ...
}
```

```java
public class Containers {

    public static boolean duplicate_1(Container<?> c) {
        return c.x1.equals(c.x2);
    }

    public static <T> boolean duplicate_2(Container<T> c) {
        return c.x1.equals(c.x2);
    }

}
```

Ok!

Ok!
Wildcards may replace named type parameters

\textit{but not always!}

\begin{verbatim}
public class Containers {

    public <T> void swap_1(Container<T> c) {
        T temp = c.x1;
        c.x1 = c.x2;
        c.x2 = temp;
    }

    public void swap_2(Container<?> c) {
        swap_1(c);
    }

    public void swap_3(Container<?> c) {
        Object temp = c.x1; //OK
        c.x1 = c.x2;      //Add cast to 'capture of ?'
        c.x2 = temp;     //Cannot convert from Object to 'capture of ?'
    }
}
\end{verbatim}

\textbf{Ok:} A swap-methods that takes a container of anything.

\textbf{Ok:} A swap-methods that takes a container of anything.

\textbf{NOT Ok!} wildcard can not be “captured”!
Java Generics

Comparisons
Comparisons

```java
public class Apple extends Fruit implements Comparable<Apple> {
    public Apple(int weight) {
        super(weight);
    }
    public int compareTo(Apple other) {
        return this.weight < other.weight ? -1
            : this.weight == other.weight ? 0
            : 1;
    }
}

public class Orange extends Fruit implements Comparable<Orange> {
    public Orange(int weight) {
        super(weight);
    }
    public int compareTo(Orange other) {
        return this.weight < other.weight ? -1
            : this.weight == other.weight ? 0
            : 1;
    }
}
```

Pure fruits do not exist and can't be compared

Compare apple with apple

Compare orange with orange
Comparisons

```java
public static <T extends Comparable<T>> T max (Collection<T> c) {
    if (c.size() == 0)
        throw new NoSuchElementException();
    T largest = null;
    for (T x: c) {
        if (largest == null || largest.compareTo(x) < 0)
            largest = x;
    }
    return largest;
}

public static void main(String[] args) {
    List<Fruit> l = new LinkedList<Fruit>();
l.add(new Apple(19));
l.add(new Apple(20));
    System.out.println(max(l));  // Type Error Bound mismatch!
}
```

Bound mismatch:
Fruit in List<Fruit>
does not match the bound
<T extends Comparable<T>>
Comparisons

Solution: make all fruits comparable

```java
abstract public class Fruit implements Comparable<Fruit> {
    private int weight;

    Fruit(int weight) {
        this.weight = weight;
    }

    public intcompareTo(Fruit other) {
        return this.weight < other.weight ? -1
                        : this.weight == other.weight ? 0
                        : 1;
    }
}
```

```java
public class Apple extends Fruit {
    ...
    public Apple(int weight) { super(weight); }
}
```

```java
public class Orange extends Fruit {
    ...
    public Orange(int weight) { super(weight); }
}
```
Comparisons

However

List<Fruit> lf = new LinkedList<Fruit>();
lf.add(new Apple(19));
lf.add(new Orange(20));
System.out.println(max(lf)); // OK

List<Apple> la = new LinkedList<Apple>();
la.add(new Apple(21));
la.add(new Apple(22));
System.out.println(max(la)); Type Error Bound mismatch !

public static <T extends Comparable<T>> T max (Collection<T> c) {
   ...
}
Comparisons

Solution completion

```java
public static <T extends Comparable<? super T> > T max (Collection<T> c) {
    if ( c.size() == 0 ) throw new NoSuchElementException();
    T largest = null;
    for ( T x: c) {
        if ( largest == null || largest.compareTo(x) < 0)
            largest = x;
    }
    return largest;
}

public static void main(String[] args) {
    List<Fruit> lf = new LinkedList<Fruit>();
    lf.add(new Apple(19));
    lf.add(new Orange(20));
    System.out.println(max(lf));

    List<Apple> la = new LinkedList<Apple>();
    la.add(new Apple(21));
    la.add(new Apple(22));
    System.out.println(max(la));
}
```

Apple now matches the bound
```
<T extends Comparable<? super T> >
```
Comparisons

Example from the Java-API

**java.util Class Collections**

```java
static <T extends Comparable<? super T>> void sort(List<T> list)
```

*Sorts the specified list into ascending order, according to the natural ordering of its elements.*

**sort Lists**

*T extends Comparable<? super T>*:

- `T extends ...` : Take a list of comparable elements, `extends`: we want to read and use `compareTo`
- `... Comparable<? super T>` : `compareTo` may be defined in a super-class

*in order to be able to sort*

- `T must support compareTo`
- `compareTo must be defined in T`
- `or in a super class of T`
Comparisons

Example: Quicksort 1

```java
/*
 * Quicksort Arrays of Records with Key that are comparable
 */
public static <Key extends Comparable<? super Key>, Value> void quickSortRecordArray(
    Record<Key, Value>[] a, int li, int re) {
    int i = li, j = re;
    if (li >= re) return;
    Record<Key, Value> pivot = a[(li + re) >> 1];
    while (i <= j) {
        while ((a[i].key).compareTo(pivot.key) < 0) i++;
        while ((a[j].key).compareTo(pivot.key) > 0) j--;
        if (i <= j) { swap(a, i, j); i++; j--; }
    }
    quickSortRecordArray(a, li, j);
    quickSortRecordArray(a, i, re);
}
```

```java
class Record<K,V> {
    K key;
    V value;
}
```
Comparisons

Example: Quicksort 2

```java
/*
 * Quicksort Arrays with Elements without natural order
 * (compare with Comparator)
 */

public static <Eintry>
void quickSortArrayUnNatural (  Eintry[] a, int li, int re,
                              Comparator<? super Eintrag> c) {
    int i = li, j = re;
    if (li >= re) return;
    Eintry pivot = a[(li + re) >> 1];
    while (i <= j) {
        while (c.compare(a[i], pivot) < 0) i++;
        while ( c.compare(a[j], pivot) > 0) j--;
        if (i <= j) { swap(a, i, j); i++; j--; }
    }
    quickSortArrayUnNatural(a, li, j, c);
    quickSortArrayUnNatural(a, i, re, c);
}

quickSortArrayUnNatural( a,  0, a.length-1,
                        new Comparator<Integer>(){
                            public int compare(Integer arg0, Integer arg1) {
                                return arg0.compareTo(arg1);
                            }
                        });
```
Java Generics

Bridges
Bridges

```java
public abstract class Fruit implements Comparable<Fruit> {
    protected int weight;
    ...
    public int compareTo(Fruit other) {
        return this.weight < other.weight ? -1
            : this.weight == other.weight ? 0 : 1;
    }
}
```

From this the compiler generates something like

```java
public abstract class Fruit implements Comparable {
    protected int weight;
    ...
    public int compareTo(Object o) {
        compareTo((Fruit)o);
    }
    public int compareTo(Fruit other) {
        return this.weight < other.weight ? -1
            : this.weight == other.weight ? 0 : 1;
    }
}
```

**Bridge method** (no variations in parameters of redefined methods!)

Cast will always succeed because the compiler checks compatibility with `Comparable<Fruit>`
**Bridges**

```java
public class Apple extends Fruit implements Comparable<Apple> {
    private int redness;
    ....
    public int compareTo(Apple other) {
        return this.redness < other.redness ? -1
              : this.redness == other.redness ? 0 : 1;
    }
}
```

**ERROR:** The interface can not be implemented more than once with different arguments

From this the compiler refuses to generate something like

```java
public abstract class Fruit implements Comparable {
    protected int redness;
    ....
    public int compareTo(Object o) {
        compareTo((Apple)o);
    }
    public int compareTo(Apple other) {
        return this.redness < other.redness ? -1
               : this.redness == other.redness ? 0 : 1;
    }
}
```

**Bridge method**
redefined version. May lead to a ClassCastException.
This is not acceptable for generics.
(Comparison with a Fruit)
public class Apple extends Fruit {
    public Apple(int weight) {
        super(weight);
    }
    private int redness;

    public int compareTo(Fruit other) {
        if (other instanceof Apple)
            return compareTo((Apple)other);
        else
            return super.compareTo(other);
    }

    public int compareTo(Apple other) {
        return this.redness < other.redness ? -1 :
            this.redness == other.redness ? 0 : 1;
    }
}

Solution: Use the traditional way to define a special version of compareTo