



- **generic programming**
- **java generic programming**
  - methods & generic programming
  - classes & generic programming
- **java with “*generics*”**
  - generic methods
  - generic classes
- **java collections framework**
  - collections framework (pre JDK 5)
  - collections framework with generics
- **references**

- **definitions** of generic programming
- generic programming is a **programming style** in which **algorithms** are written at the most **abstract** possible **level independent** of the form of the data on which these algorithms will be carried out
- generic programming is a style of computer **programming** in which algorithms are written in terms of **types to-be-specified-later** that are then instantiated when needed for specific **types** provided as **parameters**

- David Musser and Alexander Stepanov, in the early 1970s
- the term ‘generic programming’ is coined in 1989
  - the generic programming approach was pioneered by **ML** in 1973 (?)
  - the generic programming approach was pioneered by **ADA** in 1983 (?)
- different terms (& implementation) → *similar concept*
  - *generics*
    - Ada, Eiffel, Java, C#, VisualBasic.NET
  - *parametric polymorphism*
    - ML, Scala, Haskell
  - *templates*
    - C++

- **functions** (methods) or **types** (classes) that **differ** only in the set of **types** on which they operate
- generic programming is a way to make a language **more expressive**, while still maintaining **full static type-safety**
- **reduce duplication** of code
- algorithms are written in terms of **generic types**
- types are passed as **parameters** later when needed

- ***generic function***
  - performs the same operation on different data types
- ***generic type (class)***
  - store values and perform operation on different data types
- ***java***
  - generics
- ***c++***
  - templates
  - (concepts)



- generics add a way to specify ***concrete types*** to ***general purpose classes*** and methods that operated on Object before
- Java Specification Request 14 (2004)
  - add generic types to the java programming language
- generics in JDK 5 (originally numbered 1.5) (2005)
- *“This long-awaited enhancement to the type system allows a type or method to operate on objects of various types while providing compile-time type safety. It adds compile-time type safety to the Collections Framework and eliminates the drudgery of casting”* [docs.oracle.com]

- ***type variable***
  - is an unqualified identifier
- ***generic class***
  - if it declares one or more type variables (type parameters of the class)
- ***generic interface***
  - if it declares one or more type variables (type parameters of the interface)
- ***generic method***
  - if it declares one or more type variables (formal type parameters of the method)



- *step back*
  - *overloading*
  - *inheritance & polymorphism*
- *step forward*
  - *generics*



- a first possible solution: *overloading*
- overloading
  - set of methods all having the *same name*, but with a *different* arguments list (*signature*)
- first example:
  - *get the central element of array*

```
/**
 * Generic method - Overloading
 * @author SoWIDE lab
 */
public class ArrayUtil {
    /**
     Get the central element of array
     @param a String array
     @return central element
     */
    public static String getCentral(String[] a) {
        if (a == null || a.length == 0)
            return null;
        return (a[a.length/2]);
    }

    public static Character getCentral(Character[] a) {
        if (a == null || a.length == 0)
            return null;
        return (a[a.length/2]);
    }

    public static Integer getCentral(Integer[] a) {
        if (a == null || a.length == 0)
            return null;
        return (a[a.length/2]);
    }
}
```

```
public class Main {
    public static void main(String[] args) {

        String[] s = {"alpha", "beta", "charlie"};
        Character[] c = {'h', 'a', 'l'};           // autoboxing
        Integer[] i = {4, 8, 15, 16, 23, 42};

        String sc = ArrayUtil.getCentral(s);
        assert sc.equals("beta");
        Character cc = ArrayUtil.getCentral(c);
        assert cc == 'a';
        int ic = ArrayUtil.getCentral(i);         // unboxing
        assert ic == 16;

        Double[] d = {1.1, 2.3, 5.8, 13.21};
        Double dc = ArrayUtil.getCentral(d); // compile time error:
                                           // no suitable method found for getCentral(Double[])
        assert dc == 5.8;
    }
}
```

- ***assertion***

- an assertion is a statement in the Java programming language that enables you to ***test*** you're assumptions about your program

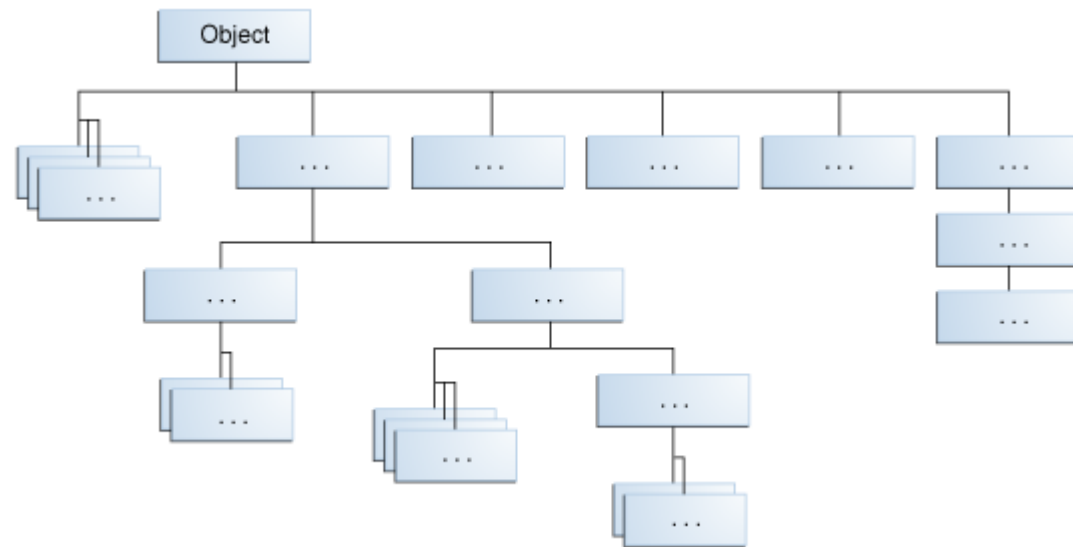
- ***autoboxing***

- autoboxing is the automatic ***conversion*** that the Java compiler makes between the ***primitive types*** and their corresponding object ***wrapper classes***

- ***unboxing***

- unboxing is the conversion of an object of a ***wrapper*** type to its ***corresponding primitive*** value
- the Java compiler applies unboxing when an object of a wrapper class is:
  - passed as a parameter to a method that expects a value of the corresponding primitive type
  - assigned to a variable of the corresponding primitive type

- we can *write* a method that takes a *base class* (or interface) as an argument, and then *use* that method with any *class derived* from that base class
- this method is more general and can be used in more places



```
/**
 * Generic method - Inheritance
 * @author SoWIDE lab
 */
public class ArrayUtil {
    /**
     Get the central element of the array
     @param a Object array
     @return central element
     */
    public static Object getCentral(Object[] a)
    {
        if (a == null || a.length == 0)
            return null;
        return (a[a.length/2]);
    }
}
```

```
public class Main {
    public static void main(String[] args) {

        String[] s = {"alpha", "beta", "charlie"};
        Character[] c = {'h', 'a', 'l'};
        Integer[] i = {4, 8, 15, 16, 23, 42};

        String sc = (String) ArrayUtil.getCentral(s);    //downcast from Object to String
        assert sc.equals("beta");
        Character cc = (Character) ArrayUtil.getCentral(c);
        assert cc == 'a';
        int ic = (int) ArrayUtil.getCentral(i); //downcast & unboxing
        assert ic == 16;
        Double[] d = {1.1, 2.3, 5.8, 13.21};
        Double dc = (Double) ArrayUtil.getCentral(d);
        assert dc == 5.8;

        Integer iVar = (Integer) ArrayUtil.getCentral(c); // no compile-time error
            // run-time exception
            // Exception in thread ... java.lang.ClassCastException: java.lang.Character
            //cannot be cast to java.lang.Integer ...
    }
}
```



- *type safety*
  - the compiler will validate types while compiling
  - throw an error if you try to assign the wrong type to a variable
- *downcasting* from base class can generate *no type-safe* code
  - run-time exception occurs in wrong cast operations

- a generic method (*with generics*) is a method with a *type parameter*
- you can think of it as a *template* for a *set of methods* that differ only by one or more types
- when you call the generic method, you need *not specify* which *type* to use for the type parameter
- you call the method with appropriate parameters, and the *compiler* will match up the type parameters with the parameter types
- *as with generic classes, you cannot replace type parameters with primitive types*

```
modifier <TypeVariable1, TypeVariable2 ...> returnType methodName(parameters) {  
    body  
}
```



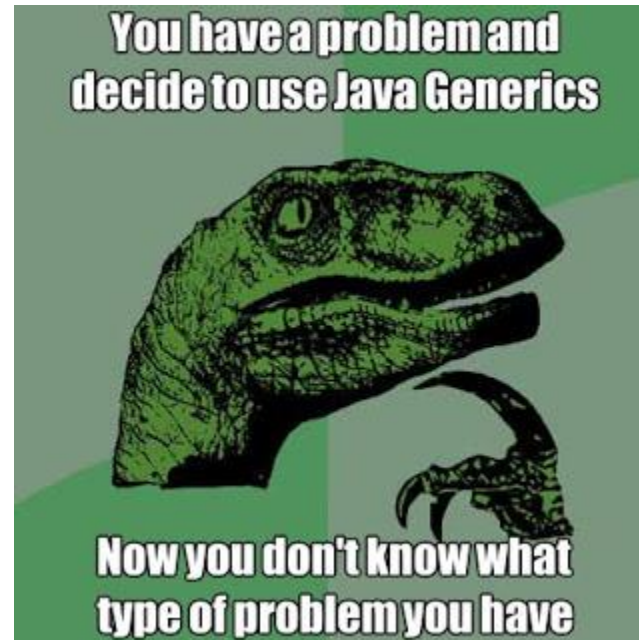
```
public static <T> T getCentral(T[] a) {  
    if (a == null || a.length == 0)  
        return null;  
    return (a[a.length/2]);  
}  
...  
String[] s = { "alpha", "beta", "charlie" };  
String sc = ArrayUtil.getCentral(s); // implicit type (String)parameter
```

```
public class Main {
    public static void main(String[] args) {

        String[] s = { "alpha", "beta", "charlie" };
        Character[] c = { 'h', 'a', 'l' };
        Integer[] i = { 4, 8, 15, 16, 23, 42 };
        Double[] d = { 1.1, 2.3, 5.8, 13.21 };

        String sc = ArrayUtil.getCentral(s); // implicit type (String) parameter
        assert sc.equals("beta");
        Character cc = ArrayUtil.<Character> getCentral(c); // explicit type parameter
        assert cc == 'a';
        int ic = ArrayUtil.getCentral(i); // implicit type parameter & unboxing
        assert ic == 16;
        Double dc = ArrayUtil.getCentral(d);
        assert dc == 5.8;

        Integer iVar = ArrayUtil.getCentral(c); // compile-time error: incompatible types
    }
}
```



- a class that hold elements of various type
- for example a simple generic class Pair that stores pairs of objects, each of which can have an arbitrary type



```
public class Pair {
    private Object first;
    private Object second;
    /**
     * Constructs a pair containing two given elements
     * @param firstElement the first element
     * @param secondElement the second element
     */
    public Pair(Object firstElement, Object secondElement) {
        first = firstElement;
        second = secondElement;
    }
    /**
     * Gets the first element of this pair
     * @return the first element
     */
    public Object getFirst() {
        return first;
    }
    /**
     * Gets the second element of this pair
     * @return the second element
     */
    public Object getSecond() {
        return second;
    }
    public String toString() {
        return "(" + first + ", " + second + ")";
    }
}
```



# test Pair

```
public class Main {  
  
    public static void main(String[] args) {  
        Pair p1 = new Pair("alpha", 1);  
        // String & Integer (autoboxing) - Implicit upcasting to Object  
        String name = (String) p1.getFirst();  
        // explicit downcasting from Object to String  
        Integer value = (Integer) p1.getSecond();  
        System.out.println("Name: "+name+" Value: "+value);  
        Pair p2 = new Pair(3.2,5.5);          // Double & Double (autoboxing)  
        Double x = (Double) p2.getFirst();  
        double y = (double) p2.getSecond();    // unboxing  
        System.out.println("x: "+x+" y: "+y);  
        x = (Double) p1.getFirst();          // run-time error  
        // Exception in thread "main" java.lang.ClassCastException: java.lang.String  
        // cannot be cast to java.lang.Double  
    }  
}
```

```
accessSpecifier class GenericClassName <TypeVariable1 , TypeVariable2 , ...> {  
    instance variables  
    constructors  
    methods  
}
```



# Pair - generic class

```
public class Pair<T, S> {  
    private T first;  
    private S second;  
  
    public Pair(T firstElement, S secondElement) {  
        first = firstElement;  
        second = secondElement;  
    }  
  
    public T getFirst() {  
        return first;  
    }  
  
    public S getSecond() {  
        return second;  
    }  
  
    public String toString() {  
        return "(" + first + ", " + second + ")";  
    }  
}
```

## Pair - test class

```
// explicit actual type parameters
```

```
Pair<String, Integer> p1 = new Pair<String, Integer>("alpha", 1);
```

```
String name = p1.getFirst();
```

```
Integer value = p1.getSecond();
```

```
System.out.println("Name: "+name+" Value: "+value);
```

```
// implicit actual type parameters
```

```
Pair<Double, Double> p2 = new Pair(3.2, 5.5);
```

```
Double x = p2.getFirst();
```

```
double y = p2.getSecond();
```

```
System.out.println("x: "+x+" y: "+y);
```

```
x = p1.getFirst();
```

```
// Compile-time error: Type mismatch: cannot convert from String to Double
```

- type variable meaning
  - **E** Element type in a collection
  - **K** Key type in a map
  - **V** Value type in a map
  - **T** General type
  - **S , U** Additional general types

- type parameters can be *constrained* with *bounds*
- it is often necessary to specify what types *can be used* in a generic class or method



- it is often necessary to formulate *constraints* of type parameters
- there are three kinds of wildcard types:

name	syntax	meaning
wildcard with <b>upper</b> bound	<b>? extends B</b>	any <i>subtype</i> of B
wildcard with <b>lower</b> bound	<b>? super B</b>	any <i>supertype</i> of B
<b>unbounded</b>	<b>?</b>	<i>any type</i>

- *example*: if you want to write a method that works on **List<Integer>**, **List<Double>**, and **List<Number>** you can achieve this by using an upper bounded wildcard
- to write the method that works on lists of Number and the subtypes of Number, such as Integer, Double, and Float, you would specify **List<? extends Number>**
- *the term List<Number> is more restrictive than List<? extends Number>*
  - *the former matches a list of type Number **only***
  - *the latter matches a list of type Number or any of its **subclasses***



## collections & generic

```
public static double sumOfList(List<? extends Number> list) {  
    double s = 0.0;  
    for (Number n : list)  
        s += n.doubleValue();  
    return s;  
}
```

```
public static double productOfList(List<? extends Number> list) {  
    double p = 1.0;  
    for (Number n : list)  
        p *= n.doubleValue();  
    return p;  
}
```

## collections & generic

```
public static void main(String[] args) {
    List<Integer> li = Arrays.asList(1, 2, 3);
    System.out.println("sum = " + sumOfList(li));           // output sum = 6
    System.out.println("product = " + productOfList(li));

    List<String> sli =Arrays.asList("alpha","beta","charlie");
    System.out.println("sum = " + sumOfList(sli));
    // Compile time error: The method sumOfList(List<? extends Number>) ...
    // is not applicable for the arguments (List<String>) ...

    List gli = Arrays.asList("alpha","beta","charlie");;
    System.out.println("sum = " + sumOfList(gli));
    // Exception in thread "main" java.lang.ClassCastException:
    // java.lang.String cannot be cast to java.lang.Number
}
```

- the *unbounded wildcard* type is specified using the wildcard character (?), for example, **List<?>**. this is called a list of unknown type
- there are two scenarios where an unbounded wildcard is a useful approach:
  - if you are writing a method that can be implemented using functionality provided in the *Object* class
  - when the code is using methods in the generic class that *don't depend on the type* parameter



- generics in Java provide **compile-time safety** for type-correctness
  - *but is partially considered as a run-time feature and it is somewhat similar to inheritance-polymorphism in practice*
- there is a process called **type erasure**
  - **type** information is **removed** during **compilation** and there is no way to tell what was the type of a generic when it was instantiated during run-time
- any algorithm that requires to know the original type cannot be implemented through generics



## source code

```
public static <T> T getCentral(T[] a)
{
    if (a == null || a.length == 0)
        return null;
    return (a[a.length/2]);
}
```

## code after erasure

```
public static Object getCentral(Object[] a)
{
    if (a == null || a.length == 0)
        return null;
    return a[a.length / 2];
}
```

- the Java compiler *erases* type parameters, replacing them with their *bounds* or *Objects*
- because the Java compiler erases all type parameters in generic code, you *cannot verify* which parameterized type for a generic type is being used *at runtime*
- the process *erases type* parameters but *adds casts*
- knowing about type erasure helps you understand limitations of Java generics
  - for example, you cannot construct new objects of a generic type

```
public static <E> void fillWithDefaults(E[] a) {  
    private E[] elements;  
    elements = new E[10]; // error  
    for (int i = 0; i < a.length; i++)  
        a[i] = new E(); // error  
}
```



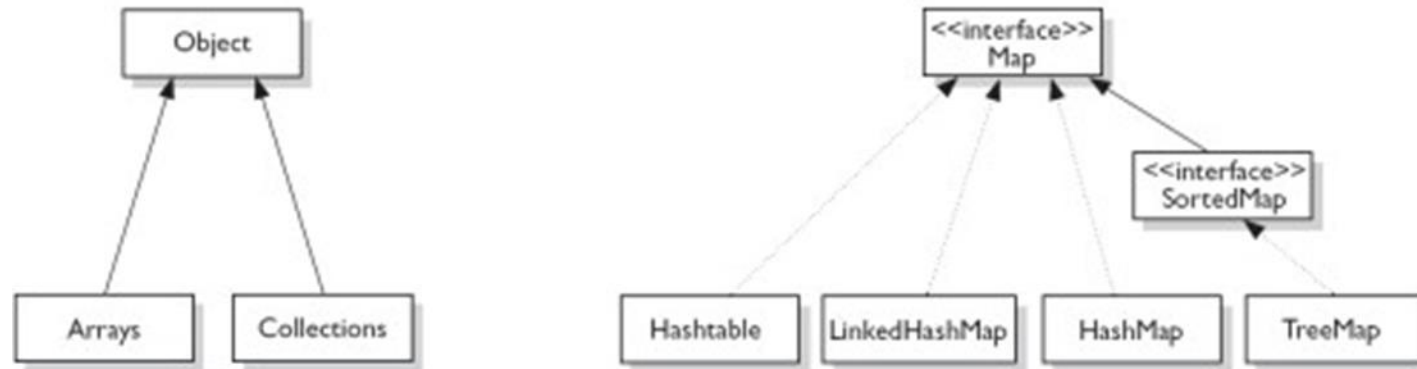
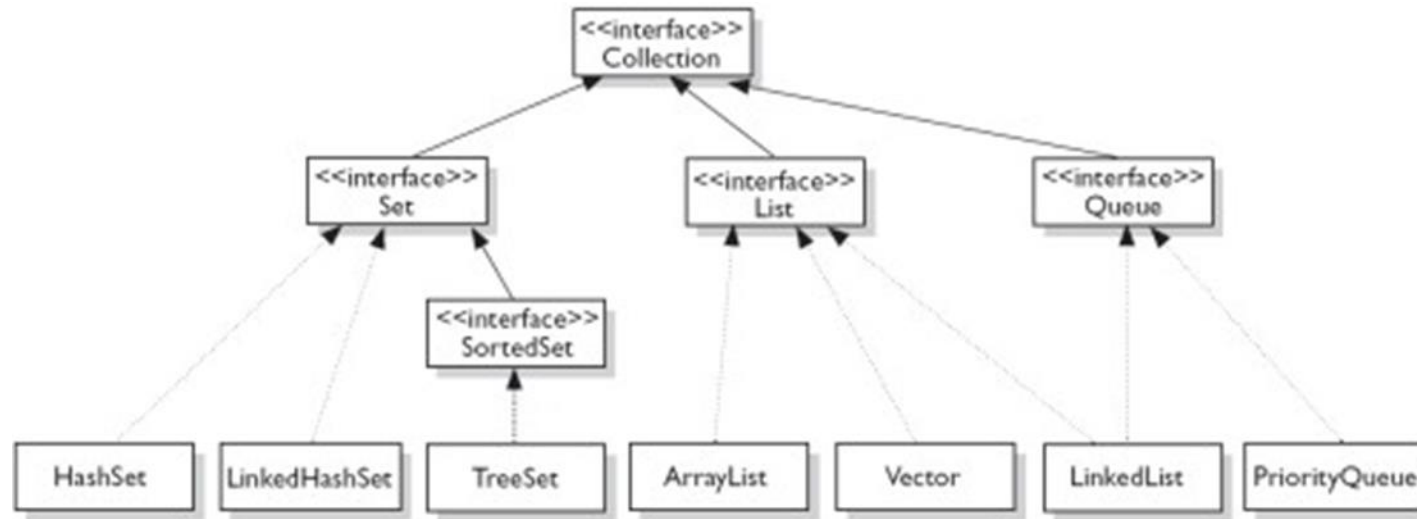
## Main (after erasure)

```
public static void main(String args[]) {
    String s[] = { "alpha", "beta", "charlie" };
    Character c[] = { Character.valueOf('h'), Character.valueOf('a'), ...};
    Integer i[] = { Integer.valueOf(4), Integer.valueOf(8), ...};
    Double d[] = { Double.valueOf(1.10000000000000001D), ...};
    String sc = (String) ArrayUtil.getCentral(s);
    if (!$assertionsDisabled && !sc.equals("beta"))
        throw new AssertionError();
    Character cc = (Character) ArrayUtil.getCentral(c);
    if (!$assertionsDisabled && cc.charValue() != 'a')
        throw new AssertionError();
    int ic = ((Integer) ArrayUtil.getCentral(i)).intValue();
    if (!$assertionsDisabled && ic != 16)
        throw new AssertionError();
    Double dc = (Double) ArrayUtil.getCentral(d);
    if (!$assertionsDisabled && dc.doubleValue() != 5.7999999999999998D)
        throw new AssertionError();
    else
        return;
}
```

- are Java Generics just syntactic sugar?



- it keeps things *simple*, in that generics do not add anything fundamentally new
- it keeps things *small*, in that there is exactly *one implementation* of List, not one version for each type
- it eases *evolution*, since the same library can be accessed in both nongeneric and generic forms
- *cast-iron guarantee*: the implicit casts added by the compilation of generics never fail



- is a unified architecture for representing and manipulating collections, enabling *collections* to be manipulated *independently* of *implementation* details
- **pre-JDK5** collections are not type-safe
  - the *upcasting* to `java.lang.Object` is done implicitly by the compiler
  - the programmer has to *explicitly downcast* the `Object` retrieved back to their original class
  - the *compiler* is *not able* to check whether the downcasting is valid at compile-time
  - incorrect downcasting will show up only at runtime, as a *ClassCastException*
- **post-JDK5**
  - the *compiler* can perform all the *necessary type-check* during compilation to ensure *type-safety at runtime*

## java collections framework (pre JDK 5)

```
import java.util.ArrayList;
import java.util.Iterator;

public class ArrayListPreJDK5Test {
    public static void main(String[] args) {
        ArrayList lst = new ArrayList(); // ArrayList contains instances of Object
        lst.add("alpha"); // add() takes Object. String upcast to Object implicitly
        lst.add("beta");
        lst.add("charlie");
        lst.add(new Integer(10)); //Integer upcast to Object implicitly
        System.out.println(lst); // [alpha, beta, charlie, 10]

        Iterator iter = lst.iterator();
        while (iter.hasNext()) {
            //explicitly downcast from Object to String
            String str = (String)iter.next(); // ERROR
            System.out.println(str);
        }
    }
}
```

## java collections framework (post JDK 5)

```
public class ArrayListPostJDK15Test {
    public static void main(String[] args) {
        ArrayList<String> lst = new ArrayList<String>(); //Inform compiler about type
        lst.add("alpha"); // compiler checks if argument's type is String
        ...
        Iterator<String> iter = lst.iterator(); // Iterator of Strings
        while (iter.hasNext()) {
            String str = iter.next(); // compiler inserts downcast operator
            System.out.println(str);
        }
        lst.add(new Integer(1234)); // ERROR: compiler can detect wrong type
                                   // error: no suitable method found for add(Integer)
        Integer intObj = lst.get(0); // ERROR: compiler can detect wrong type
                                   // error: incompatible types: String cannot be converted to Integer

        // Enhanced for-loop (JDK 1.5)
        for (String str : lst) {
            System.out.println(str);
        }
    }
}
```

- Musser, D.A. and Stepanov, A.A., Generic Programming, Proceeding of International Symposium on Symbolic and Algebraic Computation, vol. 358: Lecture Notes in Computer Science, Rome, Italy, 1988, pp. 13–25
- Giovannelli, D. (2013). Programming in Algorithms: Generic Programming and its Implementation.
- Naftalin, M., Philip Wadler, P.. Java Generics and Collections. Speed Up the Java Development Process. O'Reilly Media (2006)
- Jaime Niño. 2007. The cost of erasure in Java generics type system. J. Comput. Sci. Coll. 22, 5 (May 2007), 2-11