Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

# Programming Techniques in Java

## Generics

I. Salomie, C.Pop
2017



## Introduction

- Generics – type parameterization
- Generic type - A reference type in Java, which accepts one or more type parameters
- Generic type declaration in: classes, interfaces, methods
- Concrete type should be specified when using the classes, the interfaces and the methods
- Improves
  - Reliability
  - Readability
  - Compile-time type safety (no cast is necessary)
- Allows writing polymorphic code that works with any type
- Introduced since
  - Java JDK 1.5 (some classes and interfaces were modified to use generics)
- Generic type – implemented at the compiler level
- JVM has no knowledge of generic type

# Motivation

**Consider the interface Comparable**

- Before JDK 1.5

```
public interface Comparable {
    public int compareTo(Object o);
}
```

- JDK 1.5

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

- <T> – formal generic type
- More formal generic parameters
  - <E, T> or <T1, T2, T3>

**Convention**

- T indicates that parameter is type
- E – element
- K – key
- V – value

**Using the interface Comparable**

- Before JDK 1.5

```
// compiles ok
// generates a run time error
Comparable c = new Date();
int comp = c.compareTo("Cluj");
```

- After JDK 1.5

```
// compile error
// the code is more reliable
Comparable<Date> c = new Date();
int comp = c.compareTo("Cluj");
```



# Declaring generic classes and interfaces
## Classical approach

```
public class ClassicalStack {
   private ArrayList list = new java.util.ArrayList();

   public boolean isEmpty() { return list.isEmpty(); }
   public int getSize() { return list.size();}

   public Object peek() { return list.get(getSize() – 1); }

   public Object push(Object o) { list.add(o); return o; }

   public Object pop() {
      Object o = list.get(getSize() – 1);
      list.remove(getSize() – 1);
      return 0;
   }
}
```

**Examples of usage**

```
ClassicalStack stk1 = new ClassicalStack();
stk1.push("Cluj");
stk1.push("Oradea");
stk1.push("Timisoara");
String s1 = stk1.pop(); // error
String s2 = (String)stk1.pop(); // OK
```

**Example 2**

```
ClassicalStack  stk2 = new ClassicalStack ();
stk2.push(2.5);
stk2.push(0.3);
stk2.push(18.2);
double d1 = stk2.pop(); // error
double d2 = (double)stk2.pop(); // OK
```

# Declaring generic classes and interfaces

## With generics

```
public class GenericStack<E> {
  private java.util.ArrayList<E> list = new java.util.ArrayList<E>();
  public int getSize() { return list.size();}
  public boolean isEmpty() { return list.isEmpty();}
  public E peek() {  return list.get(getSize()-1);}
  public E push(E o) { list.add(o); return(o); }
  public E pop() {
    E o = list.get(getSize() – 1);
    list.remove(getSize() -1);
    return o; }
}
```

**Instantiation**

- Actual concrete type substitutes a formal generic type
- Generic types must be reference types
- Formal generic types cannot be directly instantiated
- !!! Illegal constructs: new T() , new T[SIZE]



**Examples of using generics**

**Example 1**

```
GenericStack<String> stk1 = new
    GenericStack<String>();
stk1.push("Cluj");
stk1.push("Oradea");
stk1.push("Timisoara");
String s = stk1.pop(); // no cast
```

**Example 2**

```
GenericStack<Double> stk2 = new
    GenericStack<> ();
stk2.push(2.5); // auto boxing 2.5 to
    new Double(2.5)
stk2.push(0.3);
stk2.push(18.2);
Double d = stk2.pop(); // no cast
```

# Generic Methods and Constructors

```
public class GenericMethodDemo {
  …
  public static <E> void print(E[ ] list) {
    for (int i = 0; i < list.length; i++)
      System.out.print(list[i] + " ");
    System.out.println();
  }

  public static void main(String[ ] args ) {
     Integer[ ] aInts = {1, 2, 3, 4, 5};
     String[ ] aStrings = {"Cluj", "Oradea", "Turda"};

     // invocation of the generic method
     GenericMethodDemo.<Integer>print(aInts);
     GenericMethodDemo.<String>print(aStrings);
     // …
  }

}
```

# Generic Methods and Constructors

```
public class Wrapper<T> {
  private T ref;
  public Wrapper(T ref) { this.ref = ref;}
  public T get() { return ref; }
  public void set(T a) { this.ref = ref; }
}

// define a new type parameter for methods
// V forces a and b parameters to be of the
     same type
// c must be of type T (the type of class
     instantiation)
public class Test<T> {
  // … other class resources
  public <V> void m1(Wrapper<V> a,
     Wrapper<V> b, T c) {
    // … do something
  }
}
```

```
// Using the defined resources
Test<String> t = new Test<String>();
Wrapper<Integer> iw1 = new Wrapper<Integer>(new
     Integer(100));
Wrapper<Integer> iw2 = new Wrapper<Integer>(new
     Integer(200));

// Below, shows that Integer is the actual type for the type
     parameter for m1()
t.<Integer>m1(iw1, iw2, "hello");
// Let the compiler figure out the actual type parameter for
     the m1() call using types for iw1 and iw2
t.m1(iw1, iw2, "hello"); // OK
```



# Generic Methods and Constructors
## Type parameters for constructors

```
// class T – shows the constructor
public class Test<T> {
  // type parameter U must be the same with T or subtype of T
  public <U extends T> Test(U k) {  // type parameter U
    // … do something
  }
}
```

1. The compiler figures out the actual type parameter passed to the constructor (Integer) with the value you pass

```
Test<Number> t2 = new Test<Number>(new Integer(123));
```

2. The type parameter passed to the constructor is explicitly indicated

```
Test<Number> t1 = new <Double>Test<Number>(new Double(12.89));
```

# Generic Methods and Constructors
# Type inference when creating objects

```
List<String> list = new ArrayList<String>();

// Use of diamond operator (Java 7 and later)
List<String> list = new ArrayList<>();

// Using ArrayList as a raw type, not a generic type generates unchecked warning
List<String> list = new ArrayList();
```

**Note.**

- Sometimes compiler fails to infer correctly the parameter type of a type in an object-creation expression
  - Specify the parameter type instead of using the diamond operator (<>)
  - Otherwise, the compiler will infer a wrong type, which will generate an error.
- Conclusion – use diamond operator only when type inference is obvious

```
List<String> list1 = Arrays.asList("A", "B");
List<Integer> list2 = Arrays.asList(9, 19, 1969);

// Inferred type is String
List<String> list3 = new ArrayList<>(list1);

// Compile-time error
List<String> list4 = new ArrayList<>(list2);

// Inferred type is String
List<String> list5 = new ArrayList<>();
```

**Example**

```
public static void process(List<String> list) {
  // Code goes here
}
// Call method process
process(new ArrayList<>());
```

- The inferred type is Object in Java 7 (generates Error ), and String in Java 8 and later (No error)
- Java 8 is smarter compiler (it is looking at the **process** method signature)



# Generics and arrays

```
public class GenericArrayTest<T> {
  private T[] elements;
  public GenericArrayTest(int nmb) {
    // Compile time error
    elements = new T[nmb];
  }
  // … Other code goes here
}
```

- **new** is a run time operator
- No runtime information about T !!
- It is allowed to create an array of unbounded wild card:

```
Wrapper<?>[] arr = new Wrapper<?>[10];
```

How to create an array of generic type?

**Answer**: Use newInstance() method of the java.lang.reflect.Array

```
Wrapper<String>[] a =
(Wrapper<String>[])Array.newInstance(Wrapper.class, 10);
```

**Note**. Wrapper.class generates the object of type Class corresponding to the class Wrapper.

**Note**. An unchecked warnings at compile time will be issued due to the cast used in the array creation statement (no type information at runtime)

```
// Populate the array
// OK. Checked by compiler
a[0] = new Wrapper<String>("Hello");
```

# Bounded generic types

**Another example**

- Method **max** (from class **Collections**) to find the maximum element in a nonempty collection

```
public static <T extends Comparable<T>> T max(Collection<T> coll) {
   T candidate = coll.iterator().next();
   for (T elt : coll) {
      if (candidate.compareTo(elt) < 0) candidate = elt;
   }
   return candidate;
}
```

- **max** method declares a bound on the type variable (T is bounded by Comparable<T>)
- **iterator().next()** is used rather than get(0) to get the first element, because get is not defined on collections other than lists.
- Method raises a **NoSuchElement** exception when the collection is empty
- Notes
  - Bounds for type variables - always indicated by the keyword **extends** (even when the bound is an interface)
  - Unlike wildcards, type variables must always be bounded using **extends**, never **super**



# Bounded generic types

- Calling method **max**
  - When calling the method, T may be specified (for example):
  - Integer (since Integer implements Comparable<Integer>) or
  - String (since String implements Comparable<String>):

```
List<Integer> ints = Arrays.asList(0,1,2);
assert Collections.max(ints) == 2;

List<String> strs = Arrays.asList("zero","one","two");
assert Collections.max(strs).equals("zero");
```

- **Note.**
- T cannot be Number (Number does not implement Comparable):

```
List<Number> nums = Arrays.asList(0,1,2,3.14);
assert Collections.max(nums) == 3,14;  // compile-time error
```

# Raw type

Raw type – is classical definition of classes / interfaces

```
GenericStack stack = new GenericStack();
```

Equivalent generic type

```
GenericStack<Object> stack = new
      GenericStack<Object>();
```

**Raw type is unsafe**

Example

```
public class Max {
   public static Comparable max(
                  Comparable o1, Comparable o2) {
   if (o1.compareTo(o2) > 0) return o1;
     else return o2; }
   }
}
```

**// invocation**

```
// compiles ok, runtime error
Max.max("alpha", 3);    // 3 is auto boxed to new Integer(3)
```

When compiled with JDK 1.5 > compiler using –Xlint:unchecked,
a warning will be displayed

**Generic type is safe**

Example

```
public class Max1 {
   public static <E extends Comparable<E>> E max (E
     o1, E o2) {
      if (o1.compareTo(o2) > 0)  return o1;
      else return o2;
   }
}
```

**// invocation**

```
Max1.max("alpha", 3); // compilation error
```

- The arguments must be of the same type (two String or two ints)
- In E o1, E o2, E must be subtype of Comparable<E>
- In other words
  - E must inherit from the same super class and
  - The superclass must implement Comparable



# Raw type

**Example**

```
1  GenericStack stack; // raw type declaration
2  stack = new GenericStack<String>();
3  stack.push("Welcome to Java");
4  stack.push(new Integer(2));
```

- In line 1 a raw type is declared
- It is assigned a generic type in line 2
- Now, compiler gets confused; Line 4 is unsafe because the stack is intended to store strings
- Line 3 should be OK, but the compiler will show warnings on both line 3 and line 4
  - It cannot follow the semantic meaning of the program
  - All the compiler knows is that stack is a raw type and it is unsafe to perform certain operations
  - Therefore, warnings are displayed to alert potential programs

# Wildcards

- Wildcard type - denoted by<?>
  - ? - Denotes unknown type
- **For a generic type, a wildcard type is what an Object type is for a raw type**
- To a generic wildcard type can be assigned any generic type
- Wildcard processing - complex set of rules
- It's good to remember
  - Generics purpose is compile time safety
  - If the compiler is satisfied that the operation will not produce any surprising results in execution, it allows the statement to pass



# Wildcards
## The unbounded

**Example**

```
// Wrapper of String type
Wrapper<String> stringWrapper = new
                Wrapper<String>("Hi");
// You can assign a Wrapper<String> to
// Wrapper<?> type
Wrapper<?> wildCardWrapper = stringWrapper;

// wildCardWrapper has unknown type
Wrapper<?> unknownWrapper;

// to unknownWrapper can be assigned
// any typed reference
Wrapper<?> unknownWrapper = new
              Wrapper<String>("Hello");
```

- Remember that Wrapper **get()** returns a reference of type T

```
String str = unknownWrapper.get();  // compile err
Object obj = unknownWrapper.get(); // OK
```

- Wrapper **set(T a)** takes a T type argument which is unknown to unknownWrapper

```
// compile time error – all 3 below
unknownWrapper.set("Hello");
unknownWrapper.set(new Integer());
unknownWrapper.set(new Object());
```

- Getting type information using reflection

```
public class WrapperUtil {
  public static void printDetails(Wrapper<?> wrapper) {
    // Can assign get() return value to Object
    Object value = wrapper.get();
    String className = null;
    if (value != null) {
      className = value.getClass().getName();
    }
    System.out.println("Class: " + className);
    System.out.println("Value: " + value);
  }
}
```

# Wildcards
## Lower and Upper Bounded

**Upper bounded Wildcards**

Define method add in WrapperUtil class

- Takes two wrapped numbers Integer, Long, Short, Double, Float
- Return their sum

```
public static double sum(Wrapper<?> n1,
                         Wrapper<?> n2) {
  // … Code goes here
}
// Try adding an Integer and a String
double d = sum(
 new Wrapper<Integer>(new Integer(125)),
 new Wrapper<String>("Hello"));
```

- Meaningless computation
- Compiles OK
- => Runtime error

**Correct approach using upper bounded wildcards**

```
public static double sum(Wrapper<? extends Number> n1,
                         Wrapper<? extends Number> n2) {
  Number num1 = n1.get();
  Number num2 = n2.get();
  double sum = num1.doubleValue() + num2.doubleValue();
  return sum;
}
```

- Now, sum statement below will be rejected by the compiler

```
double d = sum(
 new Wrapper<Integer>(new Integer(125)),
 new Wrapper<String>("Hello"));
```

- Consider the code

```
Wrapper<Integer> intWrapper = new
                  Wrapper<Integer>(new Integer(10));
Wrapper<? extends Number> numberWrapper =
                   intWrapper; // Ok
numberWrapper.set(new Integer(1220)); // compile  error
numberWrapper.set(new Double(12.20)); // compile error
```

- set() statements – compiler is unsure at compile time that numberWrapper is a type of Integer or Double, which are subtypes of a Number



# Wildcards
## Lower and Upper Bounded

**Lower bounded Wildcards**

- Copy method of class **Collections**
- Copies into a destination list, all of the elements from a source list

```
public static <T> void copy(List<? super T> dst,
                            List<? extends T> src) {
    for (int i = 0; i < src.size(); i++) {
        dst.set(i, src.get(i));
    }
}
```

- dst is **? super T**
  - the destination list may have elements of any type that is a supertype of T
- src is **? extends T**
  - the source list may have elements of any type that is a subtype of T
- Sample call

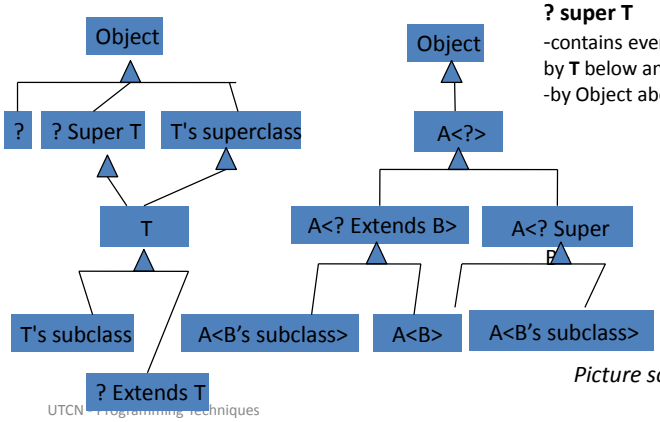
```
List<Object> objs = Arrays.<Object>asList(2,
    3.14, "four");
List<Integer> ints = Arrays.asList(5, 6);
Collections.copy(objs, ints);
assert objs.toString( ).equals("[5, 6, four]");
```

# Wildcards
# Lower and Upper Bounded

- The relationship between generic types and wildcard types
- T is a generic type, A and B are classes or interfaces

**? extends T**
-contains every type in an interval bounded by the type of null below and
- by T above (where the type of null is a subtype of every reference type).

**? super T**
-contains every type in an interval bounded by **T** below and
-by Object above




*Picture source: Liang*



# Get and Put Principle

- Good practice to insert wildcards whenever possible
- How do you decide which wildcard to use?
  - Where should you use **extends**?
  - Where should you use **super?**
  - Where is it inappropriate to use a wildcard at all?
- **The Get and Put Principle:**
  - Use an **extends** wildcard when you only **get** values out of a structure
  - Use a **super** wildcard when you only **put** values into a structure, and
  - Don't use a wildcard when you use both **get** and **put** on the same collection

- This principle was shown in the signature of the copy method copy:

public static <T> void copy(List<? super T> dest, List<? extends T> src)

- The method
  - **gets** values out of the source **src**,
    => it is declared with an **extends** wildcard
  - **puts** values into the destination **dst**
    => it is declared with a **super** wildcard

# Get and Put Principle

*Whenever you use an **iterator** or a **loop** and you **get** values out of a structure, so use an **extends** wildcard*

Method **sum** below

- takes a collection of numbers,
- converts each to a double, and
- sums them up

```
public static double sum(Collection<? extends
    Number> nums) {
 double s = 0.0d;
 for (Number num : nums) s +=
    num.doubleValue();
 return s;
}
```

- Since **extends** is used, all of the following calls are legal:

```
List<Integer> ints = Arrays.asList(1,2,3);
assert sum(ints) == 6.0;

List<Double> doubles = Arrays.asList(2.78,3.14);
assert sum(doubles) == 5.92;

List<Number> nums =
    Arrays.<Number>asList(1,2,2.78,3.14);
assert sum(nums) == 8.92;
```

- The first two calls would not be legal if extends were not used



# Get and Put Principle

*Whenever you use the **add** method, you put values into a structure, so use a **super** wildcard*

Method **addToCol** below

- takes a collection of numbers and an integer n, and
- puts the first n integers, starting from zero, into the collection

```
public static void addToCol(Collection<?
    super Integer> ints, int n) {
 for (int i = 0; i < n; i++) ints.add(i);
}
```

- Since this uses **super**, all of the following calls are legal:

```
List<Integer> ints = new ArrayList<Integer>();
addToCol(ints, 5);
assert ints.toString().equals("[0, 1, 2, 3, 4]");

List<Number> nums = new ArrayList<Number>();
addToCol(nums, 5);  nums.add(5.0);
assert nums.toString().equals("[0, 1, 2, 3, 4, 5.0]");

List<Object> objs = new ArrayList<Object>();
addToCol(objs, 5);  objs.add("five");
assert objs.toString().equals("[0, 1, 2, 3, 4, five]");
```

- The last two calls would not be legal if **super** were not used

# Get and Put Principle

*Whenever you both **put** values into and **get** values out of the same structure, you should not use a wildcard*

```
public static double sumCount(Collection<Number>
    nums, int n) {
  addToCol(nums, n);
  return sum(nums);
}
```

The collection **nums** is passed to both **sum** and **addToCol** and its elements type:

- must both extend Number (as sum requires) and
- be super to Integer (as addToCol requires)

The only two classes that satisfy both of these constraints are Number and Integer, and we have picked the first of these.

Here is a sample call:

```
List<Number> nums = new
    ArrayList<Number>();
double sum = sumCount(nums,5);
assert sum == 10;
```

Since there is no wildcard, the argument must be a collection of Number



# Get and Put Principle

The Get and Put Principle also works in the opposite way:

- If **extends** is present = > all you will be able to do is get but not put values of that type
- If **super** is present => all you will be able to do is put but not get values of that type

Example

- Consider the following code fragment, which uses a list declared with an extends wildcard:

```
List<Integer> ints = Arrays.asList(1,2,3);
List<? extends Number> nums = ints;
double dbl = sum(nums);  // ok
nums.add(3.14);  // compile-time error
```

- The call to sum is **OK**
  - it **gets** values from the list
- The call to add is **not OK**
  - it **puts** a value into the list
  - It is not allowed could add a double to a list of integers!

# Get and Put Principle

Conversely, consider the following code fragment, which uses a list declared with a super wildcard:

```
List<Object> objs = Arrays.<Object>asList(1,"two");
List<? super Integer> ints = objs;
ints.add(3);  // ok
double dbl = sum(ints);  // compile-time error
```

- The call to **add** is OK,
  - It puts a value into the list,
- The call to sum is not OK,
  - It gets a value from the list.
  - The sum of a list containing a string makes no sense!

- Each of these rules has one exception
- One cannot put anything into a type declared with an extends wildcard except for the value null
  - Null belongs to every reference type

```
List<Integer> ints = Arrays.asList(1,2,3);
List<? extends Number> nums = ints;
nums.add(null);  // ok
assert nums.toString().equals("[1,2,3,null]");
```

- One cannot get anything out from a type declared with an extends wildcard except for a value of type Object
  - Object is a supertype of every reference type

```
List<Object> objs = Arrays.<Object>asList(1,"two");
List<? super Integer> ints = objs;
String str = "";
for (Object obj : ints) str += obj.toString();
assert str.equals("1two");
```



# Generic implementations
## Erasure and restrictions

- Generics is implemented using *type erasure*
- The compiler uses the generic type information to compile the code but erases it afterwards
  - => generic information is not available at runtime
  - => enables the generic code to be backwards compatible with legacy code that uses raw types
- The generic code is presented to the compiler
  - Once the compiler confirms that the generic type is safely (correctly) used it is converted to raw type

Example 1

```
ArrayList<String> list = new ArrayList<String>();
list.add("UTCN");
String univ = list.get(0);
```

Translated into raw types

```
ArrayList list = new ArrayList();
list.add("UTCN");
String univ = (String) (list.get(0));
```

# Generic implementations
## Erasure and restrictions

### Example 2

```
public static <E> void print(E[] list) {
  for (int i = 0; i < list.length; i++)
     System.out.print(;ist[i] + " ");
  System.out.println();
}
```

### Translated to

```
public static Object void print(Object[] list) {
  for (int i = 0; i < list.length; i++)
     System.out.print(;ist[i] + " ");
  System.out.println();
}
```

Exampel 3 (bounded generic type)

```
public static <E extends GeometricObject>
     boolean equalArea (E o1, E o2) {
  return o1.getArea() == o2.getArea();
}
```

Translated into

```
public static GeometricObject boolean
     equalArea (GeometricObject o1,
     GeometricObject o2) {
  return o1.getArea() == o2.getArea();
}
```



# Generic implementations
## Erasure and restrictions

### Notes

```
ArrayList<String> list1 = new
     ArrayList<String>();
ArrayList<Integer> list2= new
     ArrayList<Integer>();
```

- ArrayList<String> and ArrayList<Integer> are two types at compile time
- JVM contains at runtime only one ArrayList class
  - list1 and list2 are both instances of ArrayList
- The following statements are true

```
list1 instanceof ArrayList
list2 instanceof ArrayList
```

- The following expression is incorrect because ArrayList<String> and ArrayList<Integer> are not stored as separate JVM classes

```
list1 instanceof ArrayList<String>
list2 instanceof ArrayList<Integer>
```

- Question: What is the class type of the object for a parameterized type?

```
public class GenericsRuntimeClassTest {
  public static void main(String[] args) {
    ArrayList<String> list1 = new ArrayList<String>();
    ArrayList<Integer> list2 = new ArrayList<Integer>();
    Class aClass = list1.getClass();
    Class bClass = list2.getClass();
    System.out.println("Class for list1: " +
      aClass.getName());
    System.out.println("Class for list2: " +
      bClass.getName());
    System.out.println("aClass == bClass: " +
      (aClass == bClass));
  }
}
```

- Answer: ArrayList for both list1 and list2

# Generic implementations
## Erasure and restrictions

- Generic types are erased at runtime => restrictions on how generic types are used
- **Restriction 1** - generic type instances are not allowed

```
// incorrect because new E is executed at
// runtime but E is not available at runtime
E object = new E();
```

**Restriction 2** - generic array creation is not allowed

```
E[] arr = new E[dim];
```

- This can be rewritten as

```
E[] arr = (E[]) new Object[dim];
```

- Casting to E[] causes an unchecked compile warning
  - The compiler could not be sure that the casting will succeed at runtime

- **Restriction 3** - Exception classes cannot be generic

```
public class MyException<T> extends
                Exception { ... }
```

- It this would be allowed, you can do

```
try { ... }
catch (My Exception<T> ex) { ... }
```

- The JVM has to check the exception thrown from the try clause to see if it matches the type specified in the catch clause
- This is impossible because the type information is not present in runtime