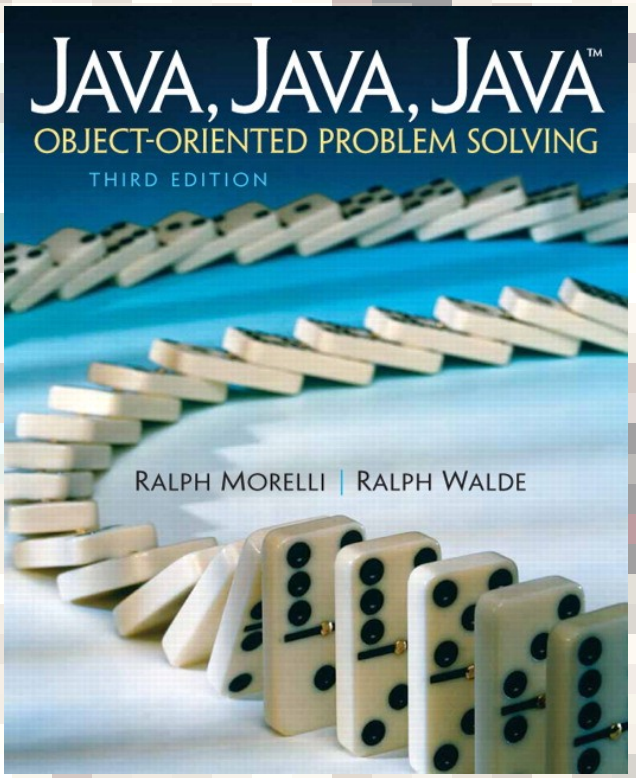

presentation slides for

Object-Oriented Problem Solving

JAVA, JAVA, JAVA

Third Edition

Ralph Morelli | Ralph Walde
Trinity College
Hartford, CT

published by Prentice Hall

Java, Java, Java

Object Oriented Problem Solving

Lecture 07: Generics in Java

Objectives

- Understand the benefits of generics
- Create a generic class
- Apply bounded type parameters
- Use wildcard arguments
- Apply bounded wildcards
- Create a generic method
- Create a generic interface
- Use raw types
- Know generics restrictions

Outline

- Generics Fundamentals
- Generics Types and Subclasses
- Generics Methods
- Bounded Type Generics
- Generics Interfaces
- Wildcards in Generics
- Generics Erasure and Casting

Generics Fundamentals

- **Generics** or **parametric polymorphism** in Java is the ability for a class or method to be written in such a way that it handles values identically without depending on knowledge of their types
  - Such a class or method is calling a generic class or method
  - Part of many high-level languages
- Motivation: Generics allow developers to write flexible, general code for software reuse, type safety, and optimization (fewer type casting)

Old-Fashioned Generics in Java <= 1.4

- **Example:**

```
public class ArrayList {
     void add(Object obj) { ... }
     Object get(int index) { ... }
}
```

- Such an ArrayList class can hold any data references data types that subclass Object class
- If typecasting fails => runtime exception

```
ArrayList names = new ArrayList();
names.add("Marty Stepp"); names.add("Stuart Reges");
String name = (String) names.get(0);
// this will compile but crash at runtime; bad
Point oops = (Point) names.get(1);
```


Modern Generics in Java >= 1.5

- **Allows generic type placeholders, i.e.,**

```
List<Type> name = new ArrayList<Type>();
```

- Support writing general code, declaring generic classes and methods
- *Instantiate* with actual types for placeholders
- Example:

```
List<String> names = new ArrayList<String>();
names.add("James"); names.add("Stuart");
String teacher = names.get(0); //no type cast
Point oops = (Point) names.get(1); //compile error
```

- Similarly, can define List<Integer>, List<Person>, List<Student>, etc.

Generic Class Syntax

```
// a parameterized (generic) class
public class name<Type> { ... }     or
public class name<Type1, Type2, ..., TypeN>{...}
```

- The **Type** in < > must be a legal reference data type such as Integer, String, Person, Student, etc.
  - Multiple type parameters separated by commas.
- The convention is to use a 1-letter name such as: T for Type, E for Element, N for Number, K for Key, or V for Value.
- The type parameter is *instantiated* by the client. (e.g. `E` → `String`)

Generic Class Notes

- Generics Work Only with Objects:
  - Primitive types are not supported, i.e., List<int> intOb = new ArrayList<int>() will cause compile error
  - Replace primitive types with wrapper class, i.e., int => Integer, double => Double
- Generic Types Differ Based on Their Type Arguments, i.e.,
  - List<Integer> is different from List<Double>
- Generic Class with more than one Type parameter:
  - Map<Integer, String> aMap = new HashMap<>();

Example

```
//A simple generic class.
//Here, T is a type parameter that will be replaced by a
//real type when an object of type Gen is created.
class Gen<T> {
    T ob; // declare a reference to an object of type T
    //Pass the constructor a reference to
    //an object of type T.
    Gen(T o) { ob = o; }
    //Return ob.
    T getob() {return ob; }
    //Show type of T.
    void showType() {
        System.out.println("Type of T is " +
                           ob.getClass().getName());
    }
}
```


Example (cont.)

```
class GenDemo {
    public static void main(String[] args) {
        //Create a Gen reference for Integers.
        Gen<Integer> iOb;
        //Create a Gen<Integer> object that store the integer number 88
        iOb = new Gen<Integer>(88);
        //Show the type of data used by iOb.
        iOb.showType();
        //Get the value in iOb. Notice that no cast is needed.
        System.out.println("value: " + iOb.getob());
        //Create a Gen object for Strings
        Gen<String> strOb = new Gen<String>("Generics Test");
        //Show the type of data used by strOb.
        strOb.showType();
        //Get the value of strOb. Again, notice that no cast is needed.
        System.out.println("value: " + strOb.getob());
    }
}
```


Generic Method

- Old version:

```
//pre: all elements of li are Strings
public void printFirstChar(ArrayList li) { }
```

- New version syntax:

```
//pre: none
public void printFirstChar(ArrayList<E> li) { }
```

- Method call examples:

```
ArrayList<String> list3 = new ArrayList<String>();
printFirstChar( list3 ); // compile ok
ArrayList<Integer> list4 = new ArrayList<Integer>();
printFirstChar( list4 ); // syntax error
```


Generics and Arrays

- You cannot create objects or arrays of a generics (parameterized) type

```
public class Foo<T> {
    private T myField;              // compile ok
    private T[] myArray;            // compile ok

    public Foo(T param) {
        myField = new T();     // compile error
        myArray = new T[10];   // compile error
    }
}
```


Generics and Arrays - Fixed

- To fixed, you can create variables of that type, accept them as parameters, return them, or create arrays by casting from Object[]

```
public class Foo<T> {
   private T myField;              // compile ok
   private T[] myArray;            // compile ok
   @SuppressWarnings("unchecked")
   public Foo(T param) {
      myField = param;             // compile ok now
      myArray = (T[])(new Object[10]);  // warning
   }
}
```

- Casting to generic types is not type-safe, so it generates a warning.

Comparing Generics Objects

- When comparing generic type objects for equality, use the equals method, i.e.,

```
public class ArrayList<E> {
    //search for E instance in the array list
    public int indexOf(E value) {
        for (int i = 0; i < size; i++) {
            //if(elementData[i] == value) { //wrong
            if(elementData[i].equals(value)) {
                return i;
            }
        }
        return -1; //not found
    }
}
```


Generic Types and Subclasses

- Let Shape is a superclass, and there are three subclasses Rectangle, Square, and Circle that extends from Shape. Then, we can define an array list of Shape and insert different instances of Rectangle, Square, and Circle to such an array list as the following example:

```
ArrayList<Shape> arlShape = new ArrayList<Shape>();
arlShape.add( new Rectangle() );
arlShape.add( new Square() );
arlShape.add( new Circle() );
// all okay
```


Bounded Type Generics

<Type extends SuperType>

- An upper bound; accepts the given supertype or any of its subtypes.
- Works for multiple superclass/interfaces with `&` :

<Type extends ClassA & InterfaceB & InterfaceC & ...>

<Type super SuperType>

- A lower bound; accepts the given supertype or any of its supertypes.

- Example:

```
// TreeSet works for any class T that extends the
   Comparable type
public class TreeSet<T extends Comparable<T>> {
    ...
}
```


Complex bounded types

- `public static <T extends Comparable<T>> T max(Collection<T> c)`
  - Find max value in any collection, if the elements can be compared.
- `public static <T> void copy( List<T2 super T> dst, List<T3 extends T> src)`
  - Copy all elements from src to dst. For this to be reasonable, dst must be able to safely store anything that could be in src. This means that all elements of src must be of dst's element type or a subtype.
- `public static <T extends Comparable<T2 super T>> void sort(List<T> list)`
  - Sort any list whose elements can be compared to the same type or a broader type.

Generics and subtyping

- Is `List<String>` a subtype of `List<Object>`?
- Is `Set<Giraffe>` a subtype of `Collection<Animal>`?

- **No**. That would violate the Liskov Substitutability Principle.
  - If we could pass a `Set<Giraffe>` to a method expecting a `Collection<Animal>`, that method could add other animals.

```
Set<Giraffe> set1 = new HashSet<Giraffe>();
Set<Animal>  set2 = set1;          // error
...
set2.add(new Zebra());
Giraffe geoffrey = set2.get(0); // error
```


Generic Interface

- You can use genetic type in the definition of an interface

```
//Represents a list of values.
public interface List<E> {
  public void add(E value);
  public void add(int index, E value);
  public E get(int index);
  public int indexOf(E value);
  public boolean isEmpty();
  public void remove(int index);
  public void set(int index, E value);
  public int size();
}
public class ArrayList<E> implements List<E> { ... }
public class LinkedList<E> implements List<E> { ... }
```


Use Wildcards in Generics

- `?` indicates a *wild-card* type parameter, one that can be any type.
  - `List<?> list = new List<?>(); // take anything`
- Difference between `List<?>` and `List<Object>`:
  - `?` can become any particular type; `Object` is just one such type.
  - `List<Object>` is restrictive; wouldn't take a `List<String>`
- Difference between `List<Foo>` and `List<? extends Foo>`:
  - The former allows anything that is a subtype of `Foo` in the same list.
    - e.g. List<Animal> could store both Giraffes and Zebras
  - The latter binds to a particular `Foo` subtype and allows ONLY that.
    - e.g. List<? extends Animal> might store only Giraffes but not Zebras

Type erasure

- All generics types become type `Object` once compiled.
  - One reason: Backward compatibility with old byte code.
  - At runtime, all generic instantiations have the same type (Object).

```
List<String>  lst1 = new ArrayList<String>();
List<Integer> lst2 = new ArrayList<Integer>();
lst1.getClass() == lst2.getClass()  // true
```

- You cannot use *instanceof* to discover a generic type parameter.

```
Collection<?> cs = new ArrayList<String>();
if (cs instanceof Collection<String>) {
    // illegal
}
```


Generics and casting

- Casting to generic type results in a warning.

```
List<?> l = new ArrayList<String>();   // ok
List<String> ls = (List<String>) l;    // warn
```

  - The compiler gives an unchecked warning, since this isn't something the runtime system is going to check for you.
  - Usually, if you think you need to do this, you're doing it wrong.

- The same is true of type variables:

```
public static <T> T badCast(T t, Object o) {
    return (T) o;    // unchecked warning
}
```


Technical Terms

- Abstract Data Type (ADT)
- binary search tree
- data structure
- dequeue
- dynamic structure
- enqueue
- first-in/first-out (FIFO)
- generic type
- Java collections framework
- key
- last-in/first-out (LIFO)
- link
- linked list
- list
- pop
- push
- queue
- reference
- self-referential object
- stack
- static structure
- traverse
- value

