


Generics

Objectives

- To know the benefits of generics.
- To use generic classes and interfaces.
- To declare generic classes and interfaces.
- To understand why generic types can improve reliability and readability.
- To design and implement generic classes.

Why Do You Get a Warning?

```
public class ShowUncheckedWarning {  
    public static void main(String[] args) {  
        java.util.ArrayList list =  
            new java.util.ArrayList();  
        list.add("Java Programming");  
    }  
}
```



To understand the **compile warning** on this line, you need to learn JDK 1.5 generics.

Fix the Warning

```
public class ShowUncheckedWarning {  
    public static void main(String[] args) {  
        java.util.ArrayList<String> list =  
            new java.util.ArrayList<String>();  
        list.add("Java Programming");  
    }  
}
```

No compile warning on this line.

What is Generics?

- *Generics* is the capability to parameterize types.
- With this capability, you can define a class or a method with generic types that can be substituted using concrete types by the compiler.
- For example,
 - you may define a generic stack class as a generic type.
 - you may create
 - a stack object for holding strings
 - and a stack object for holding numbers.
 - Strings and numbers are concrete types that replace the generic type.

Why Generics?

- The key benefit of generics is:
 - to enable errors to be detected at compile time rather than at runtime.
 - A generic class or method permits you to specify allowable types of objects that the class or method may work with. If you attempt to use the class or method with an incompatible object, the compile error occurs.

Generic ArrayList in JDK 1.5

java.util.ArrayList

+ArrayList()
+add(o: Object) : void
+add(index: int, o: Object) : void
+clear(): void
+contains(o: Object): boolean
+get(index: int) : Object
+indexOf(o: Object) : int
+isEmpty(): boolean
+lastIndexOf(o: Object) : int
+remove(o: Object): boolean
+size(): int
+remove(index: int) : boolean
+set(index: int, o: Object) : Object

(a) ArrayList before JDK 1.5

java.util.ArrayList<E>

+ArrayList()
+add(o: **E**) : void
+add(index: int, o: **E**) : void
+clear(): void
+contains(o: Object): boolean
+get(index: int) : **E**
+indexOf(o: Object) : int
+isEmpty(): boolean
+lastIndexOf(o: Object) : int
+remove(o: Object): boolean
+size(): int
+remove(index: int) : boolean
+set(index: int, o: **E**) : **E**

(b) ArrayList in JDK 1.5

No Casting Needed

```
ArrayList<Double> list = new ArrayList<Double>();  
list.add(5.5); // 5.5 is automatically converted to new Double(5.5)  
list.add(3.0); // 3.0 is automatically converted to new Double(3.0)  
Double doubleObject = list.get(0); // No casting is needed  
double d = list.get(1); // Automatically converted to double
```


Raw Type

```
// raw type  
ArrayList list = new ArrayList();
```

This is roughly equivalent to

```
ArrayList<Object> list = new ArrayList<Object>();
```

Raw Type is Unsafe

- The compiler allow this?

```
ArrayList names = new ArrayList();  
names.add("Med");  
names.add("Abdullah");  
String teacher = (String) names.get(0);
```

```
// this will compile but crash at runtime; bad  
Student oops = (Student) names.get(1);
```

Avoiding Unsafe Raw Types

Use

```
new ArrayList<ConcreteType>()
```

Instead of

```
new ArrayList();
```

Compile Time Checking

For example, the compiler checks whether generics is used correctly. The code in (b) uses the raw type.

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

(a)

```
ArrayList list = new ArrayList();  
list.add("Oklahoma");  
String state = (String)(list.get(0));
```

(b)

Implementing generics

// a parameterized (generic) class

```
public class name<Type> {
```

or

```
public class name<Type1, ..., Typen> {
```

- By putting the **Type** in < >, you are demanding that any client that constructs your object must supply a type parameter.
 - You can require multiple type parameters separated by commas.

Restrictions on Generics

- Restriction 1: Cannot Create an Instance of a Generic Type. (i.e., `new E()`).
- Restriction 2: Generic Array Creation is Not Allowed. (i.e., `new E[100]`).
- Restriction 3: A Generic Type Parameter of a Class Is Not Allowed in a Static Context.
- Restriction 4: Exception Classes Cannot be Generic.

Generics and arrays (15.4)

```
public class Foo<T> {  
    private T myField;           // ok  
    private T[] myArray;        // ok  
  
    public Foo(T param) {  
        myField = new T();      // error  
        myArray = new T[10];    // error  
    }  
}
```

- You cannot create objects or arrays of a parameterized type.

Generics/arrays, fixed

```
public class Foo<T> {  
    private T myField;           // ok  
    private T[] myArray;        // ok  
  
    public Foo(T param) {  
        myField = param;           // ok  
        T[] a2 = (T[]) (new Object[10]); // ok  
    }  
}
```

- But you can create variables of that type, accept them as parameters, return them, or create arrays by casting `Object[]`.
 - Casting to generic types is not type-safe, so it generates a warning.

Declaring Generic Classes and Interfaces

GenericStack<E>

-list: java.util.ArrayList<E>

+GenericStack()

+getSize(): int

+peek(): E

+pop(): E

+push(o: E): E

+isEmpty(): boolean

An array list to store elements.

Creates an empty stack.

Returns the number of elements in this stack.

Returns the top element in this stack.

Returns and removes the top element in this stack.

Adds a new element to the top of this stack.

Returns true if the stack is empty.

Important Facts

It is important to note that a generic class is shared by all its instances regardless of its actual generic type.

```
GenericStack<String> stack1 = new GenericStack<String>();  
GenericStack<Integer> stack2 = new  
    GenericStack<Integer>();
```

Comparing generic objects

```
public class ArrayList<E> {  
    ...  
    public int indexOf(E value) {  
        for (int i = 0; i < size; i++) {  
            // if (elementData[i] == value) {  
                if (elementData[i].equals(value)) {  
                    return i;  
                }  
            }  
        }  
        return -1;  
    }  
}
```

- When testing objects of type `E` for equality, must use `equals`

A generic 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> { ...
```

Generic methods

```
public static <Type> returnType name (params)
{
```

- When you want to make just a single (often static) method generic in a class, precede its return type by type parameter(s).

```
public class Collections {
    ...
    public static <T> void copy(List<T> dst, List<T> src) {
        for (T t : src) {
            dst.add(t);
        }
    }
}
```

Bounded type parameters

<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:

```
// tree set works for any comparable type
public class TreeSet<T extends Comparable<T>> {
    ...
}
```