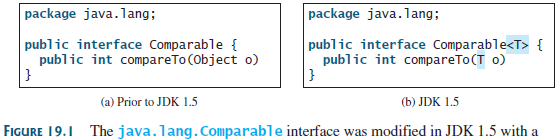
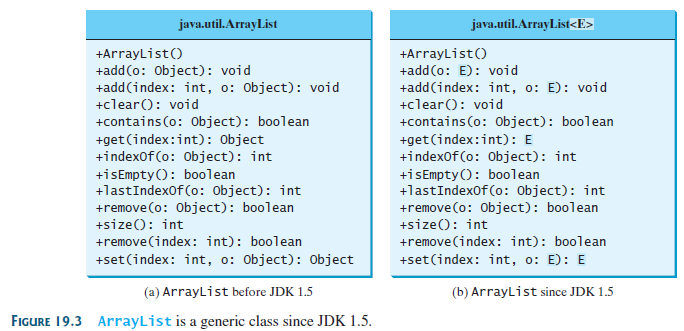
COP 2251 – Java Programming II – Generics – Chapter 19

* Generics are about defining classes and methods with a generic (unspecified) type.
* At compile time, the generic types are replaced with concrete (specific) types.
* The advantage: errors can be detected at compile time rather than runtime, and this makes Java programs more reliable. Attempts to use an incompatible type will not compile.
* Generics were introduced with Java JDK 1.5, and some classes and interfaces in the Java API use generics.
* Conventionally, **<T>** or **<E>** is used to indicate a generic type in a class or method.
* Note how the Comparable interface was modified in Figures 19.1 and 19.2 on page 752.



The ArrayList Class

* Recall that the ArrayList class exists because arrays are immutable.
* See Figure 19.3 on page 753 to see how ArrayList was updated to use generics.



* IMPORTANT: generics must be replaced with reference (aka object) types, not primitives.
* This means an ArrayList cannot contain ints or doubles, for example. Integer or Double instances are required. However, Java does allow **autoboxing** of primitives into their respective wrapper types. See bottom of page 753.
* **Auto-unboxing** eliminates casting when accessing an ArrayList of wrapped primitives like Integers or Doubles. See page 754.
* Using generics eliminates the need for casting when retrieving an element from an ArrayList because the type is known by the compiler. See page 754.

Try **IntArrayList.java** and **StringArrayList.java**. These programs are “extras” not in our book.

Generic Classes and Interfaces

Examine Listing 19.1 **GenericStack.java** on page 755 or in the zip.

Note:

* The use of **<E>** in the class signature line after the class name.
* This class can create a stack of any reference (object) type.
* The sole instance field is a generic ArrayList.
* The peek() method returns the generic type (line 8).
* The push() method adds a new object of the generic type to the ArrayList.

Try **TestGenericStack.java**. It uses **GenericStack.java** and class **String.**

Programming Exercise

Write a program that creates a GenericStack of **four Rectangle** objects. The Rectangle class is in the project. Use a loop to display the areas of all four rectangles.

Generic Methods

* A generic method is one that can work with a generic type.
* When creating a generic method, **<E>** or **<T>** is placed after the keyword static.

Try **GenericMethodDemo.java**.

Note how the static print method is called by prefixing the method name with the type in < >, but the compiler can determine the actual type at runtime. Test this by adding to main:

print(strings);

Note that the <E> in the print() method is **not the return type.**

* A generic type can be a subtype of another type. See page 757. This is called a **bounded** generic type. The syntax is **<E extends nameOfSuperclass>.**

**Try BoundedTypeDemo.java**.  
Make another 4 x 1 Rectangle in main. Run the equalArea method with the rectangles as args.

**Try MyGenericMethodDemo.java**.

Case Study: Sorting an Array of Objects

* The array types in this example implement the **Comparable** interface and can be sorted with its **compareTo()** method.

Try **GenericSort.java** to sort several arrays of objects. The method header line specifies that the method can sort an array of any object type that extends the Comparable interface.

**public** **static** <E **extends** Comparable<E>> **void** sort(E[] list) {

Raw Types

* A generic class can be used without specifying a concrete type. This is called a **raw type**.
* This is equivalent to specifying the **Object** class as the concrete type.
* Raw types enable backward compatibility for classes like ArrayList, although this is unsafe.
* The unsafe issue exists because **any** reference type could be added at compile time but the program will crash at runtime if the actual types are not as needed.
* Contrast **Max.java** (old syntax) with updated syntax in **MaxUsingGenericType.java** (page 760)
* Note: class Max has old, non-generic code but MaxUsingGenericType uses safe generic code.

**Try TestMaxClasses.java (not in book).**

* The program will compile but crashes at runtime with a ClassCastException.
* Comment out line 8 and uncomment line 9. Note that the program will not compile now because max was defined with a single generic type.

**Try ShowRawTypesIssue.java** (not in book)

* The Max.max() call on line 10 is not a compiler issue because this method uses raw types, but the program crashes when you run it because you can’t compare an Integer with a String.
* Uncomment the last line to see that when MaxUsingGenericType.max() is used, the compiler does detect the error. This is why the Java gods developed generics.
* Eclipse will display a lightbulb warning notice in the margin if you forget to use generic syntax.

Wildcard Generic Types

* Wildcard refers to a group of related types as specified by inheritance.
* Wildcard generic types take three forms:
  1. Unbounded

**<?>**

This is the same as **<? extends Object>.**

* 1. Bounded

**<? extends T>**

This means **class T or a subclass of T**.

* 1. Lower-bound

**<? super T>**

This means **class T or a superclass of class T**.

* Read page 761-764.

Try **WildCardNeedDemo.java**. Uncomment the print() to see the problem: intStack is type GenericStack<Integer> but the method parameter is GenericStack<Number>.

**Mini-Exercise**: Modify the max() method’s argument so WildCardNeedDemo.java runs okay**.**

Try **AnyWildCardDemo.java** and **SuperWildCardDemo.java**.

Erasure and Restrictions on Generics

* Java uses generic syntax to compile code but then erases it so the generic information is not available at runtime. This is called **type erasure**, and it’s needed for backward compatibility with raw types.
* After the compiler finds that generics are safe, the generic types are converted to raw types at runtime. Compare the code in the boxes (a) and (b) on page 764.
* Because of this **type erasure** method, there are restrictions on how generics can be used.
* There are four restrictions:
  1. You cannot use a generic type as a constructor.
     + **new E();** is not permitted since E is not available at runtime.
  2. You cannot create an array using a generic class.
     + **new E[ ]** is not permitted
  3. A generic class type cannot be used in a static context.
     + See page 751-752.
  4. Exception classes cannot be generic.
     + A generic class cannot extend **Throwable** or any of its subclasses.

Case Study: Generic Matrix Class

* The chapter ends with several examples in a case study of E[ ][ ] classes.
* We will downplay this, but the examples are in the zipped project.

Feel free to try the classes containing **Matrix** in the class name.

**Example**Study classes Printed, Book, TextBook and Blog (also shown on the next page).

Try **TestPrinted.java**.

**package** chap19\_550;

**public** **abstract** **class** Printed {

**private** String title;

**public** Printed(String title) {

**this**.title = title;

}

@Override

**public** String toString() {

**return** "Printed title = " + title;

}

}

**package** chap19\_550;

**public** **class** Book **extends** Printed {

**private** String genre;

**public** Book(String title, String genre) {

**super**(title);

**this**.genre = genre;

}

@Override

**public** String toString() {

**return** "Book genre = " + genre + ", " + **super**.toString();

}

}

**package** chap19\_550;

**public** **class** Blog **extends** Printed {

**private** String theme;

**public** Blog(String title, String theme) {

**super**(title);

**this**.theme = theme;

}

@Override

**public** String toString() {

**return** "Blog theme = " + theme + ", " + **super**.toString();

}

}

**package** chap19\_550;

**public** **class** TextBook **extends** Book {

**private** String subject;

**public** TextBook(String title, String genre, String subject) {

**super**(title, genre);

**this**.subject = subject;

}

@Override

**public** String toString() {

**return** "TextBook subject = " + subject + ", " + **super**.toString();

}

}