

# Chapter 22

## Generic Programming

# Chapter Goals

---

- **To understand the objective of generic programming**
- **To be able to implement generic classes and methods**
- **To understand the execution of generic methods in the virtual machine**
- **To know the limitations of generic programming in Java**

*Continued*

# Chapter Goals

---

- **To understand the relationship between generic types and inheritance**
- **To learn how to constrain type variables**

# Type Variables

- **Generic programming: creation of programming constructs that can be used with many different types**
  - In Java, achieved with inheritance or with type variables
- **For example:**
  - Type variables: Java's `ArrayList` (e.g. `ArrayList<String>`)
  - Inheritance: `LinkedList` implemented in Section 20.2 can store objects of any class

*Continued*

# Type Variables

- **Generic class:** declared with a type variable E
- **The type variable denotes the element type:**

```
public class ArrayList<E>
    // could use "ElementType" instead of E
{
    public ArrayList() { . . . }
    public void add(E element) { . . . }
    . . .
}
```

*Continued*

# Type Variables

- Can be instantiated with class or interface types

```
ArrayList<BankAccount>  
ArrayList<Measurable>
```

- Cannot use a primitive type as a type variable

```
ArrayList<double> // Wrong!
```

- Use corresponding wrapper class instead

```
ArrayList<Double>
```

# Type Variables

- **Supplied type replaces type variable in class interface**
- **Example: add in ArrayList<BankAccount>** has type variable E replaced with BankAccount:

```
public void add(BankAccount element)
```

- **Contrast with LinkedList.add:**

```
public void add(Object element)
```

# Type Variables Increase Safety

---

- Type variables make generic code safer and easier to read
- Impossible to add a String into an `ArrayList<BankAccount>`

*Continued*

# Type Variables Increase Safety

- **Can add a String into a LinkedList intended to hold bank accounts**

```
ArrayList<BankAccount> accounts1
    = new ArrayList<BankAccount>();
LinkedList accounts2 = new LinkedList();
    // Should hold BankAccount objects
accounts1.add("my savings");
    // Compile-time error
accounts2.add("my savings");
    // Not detected at compile time
. . .
BankAccount account = (BankAccount) accounts2.getFirst();
    // Run-time error
```

# Syntax 22.1: Instantiating a Generic Class

```
GenericClassName<Type1, Type2, . . .>
```

## Example:

```
ArrayList<BankAccount>  
HashMap<String, Integer>
```

## Purpose:

To supply specific types for the type variables of a generic class

# Self Check

---

1. The standard library provides a class `HashMap<K, V>` with key type `K` and value type `V`. Declare a hash map that maps strings to integers.
2. The binary search tree class in Chapter 21 is an example of generic programming because you can use it with any classes that implement the `Comparable` interface. Does it achieve genericity through inheritance or type variables?

# Answers

---

1. `HashMap<String, Integer>`
2. **It uses inheritance.**

# Implementing Generic Classes

- **Example: simple generic class that stores pairs of objects**

```
Pair<String, BankAccount> result
    = new Pair<String, BankAccount>
        >("Harry Hacker", harrysChecking);
```

- **The `getFirst` and `getSecond` retrieve first and second values of pair**

```
String name = result.getFirst();
BankAccount account = result.getSecond();
```

*Continued*

# Implementing Generic Classes

- **Example of use: return two values at the same time (method returns a `Pair`)**
- **Generic `Pair` class requires two type variables**

```
public class Pair<T, S>
```

# Good Type Variable Names

Type Variable	Name 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

# Class Pair

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

# Turning LinkedList into a Generic Class

```
public class LinkedList<E>
{
    . . .
    public E removeFirst()
    {
        if (first == null)
            throw new NoSuchElementException();
        E element = first.data;
        first = first.next;
        return element;
    }
    . . .
    private Node first;
}
```

*Continued*

# Turning LinkedList into a Generic Class

```
private class Node
{
    public E data;
    public Node next;
}
}
```

# Implementing Generic Classes

---

- Use type E when you receive, return, or store an element object
  - Complexities arise only when your data structure uses helper classes
  - If helpers are inner classes, no need to do anything special
  - Helper types defined outside generic class need to become generic classes too
- ```
public class ListNode<E>
```

# Syntax 22.2: Defining a Generic Class

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

## **Example:**

```
public class Pair<T, S>  
{  
    . . .  
}
```

## **Purpose:**

To define a generic class with methods and fields that depend on type variables

# File LinkedList.java

```
001: import java.util.NoSuchElementException;
002:
003: /**
004:     A linked list is a sequence of nodes with efficient
005:     element insertion and removal. This class
006:     contains a subset of the methods of the standard
007:     java.util.LinkedList class.
008: */
009: public class LinkedList<E>
010: {
011:     /**
012:         Constructs an empty linked list.
013:     */
014:     public LinkedList()
015:     {
016:         first = null;
017:     }
018:
```

**Continued**

# File LinkedList.java

```
019:    /**
020:        Returns the first element in the linked list.
021:        @return the first element in the linked list
022:    */
023:    public E getFirst()
024:    {
025:        if (first == null)
026:            throw new NoSuchElementException();
027:        return first.data;
028:    }
029:
030:    /**
031:        Removes the first element in the linked list.
032:        @return the removed element
033:    */
034:    public E removeFirst()
035:    {
```

**Continued**

# File LinkedList.java

```
036:         if (first == null)
037:             throw new NoSuchElementException();
038:         E element = first.data;
039:         first = first.next;
040:         return element;
041:     }
042:
043:     /**
044:      * Adds an element to the front of the linked list.
045:      * @param element the element to add
046:      */
047:     public void addFirst(E element)
048:     {
049:         Node newNode = new Node();
050:         newNode.data = element;
051:         newNode.next = first;
052:         first = newNode;
053:     }
```

**Continued**

# File LinkedList.java

```
054:
055:     /**
056:         Returns an iterator for iterating through this list.
057:         @return an iterator for iterating through this list
058:     */
059:     public ListIterator<E> listIterator()
060:     {
061:         return new LinkedListIterator();
062:     }
063:
064:     private Node first;
065:
066:     private class Node
067:     {
068:         public E data;
069:         public Node next;
070:     }
071:
```

**Continued**

# File LinkedList.java

```
072: private class LinkedListIterator implements ListIterator<E>
073: {
074:     /**
075:      * Constructs an iterator that points to the front
076:      * of the linked list.
077:      */
078:     public LinkedListIterator()
079:     {
080:         position = null;
081:         previous = null;
082:     }
083:
084:     /**
085:      * Moves the iterator past the next element.
086:      * @return the traversed element
087:      */
088:     public E next()
089:     {
```

**Continued**

# File LinkedList.java

```
090:         if (!hasNext())
091:             throw new NoSuchElementException();
092:         previous = position; // Remember for remove
093:
094:         if (position == null)
095:             position = first;
096:         else
097:             position = position.next;
098:
099:         return position.data;
100:     }
101:
102:     /**
103:      * Tests if there is an element after the iterator
104:      * position.
105:      * @return true if there is an element after the
106:      * iterator position
107:      */
```

**Continued**

# File LinkedList.java

```
108:     public boolean hasNext()
109:     {
110:         if (position == null)
111:             return first != null;
112:         else
113:             return position.next != null;
114:     }
115:
116:     /**
117:      * Adds an element before the iterator position
118:      * and moves the iterator past the inserted element.
119:      * @param element the element to add
120:      */
121:     public void add(E element)
122:     {
123:         if (position == null)
124:         {
```

**Continued**

# File LinkedList.java

```
125:         addFirst(element);
126:         position = first;
127:     }
128:     else
129:     {
130:         Node newNode = new Node();
131:         newNode.data = element;
132:         newNode.next = position.next;
133:         position.next = newNode;
134:         position = newNode;
135:     }
136:     previous = position;
137: }
138:
139: /**
140:     Removes the last traversed element. This method may
141:     only be called after a call to the next() method.
142: */
```

**Continued**

# File LinkedList.java

```
143:     public void remove ()
144:     {
145:         if (previous == position)
146:             throw new IllegalStateException ();
147:
148:         if (position == first)
149:         {
150:             removeFirst ();
151:         }
152:         else
153:         {
154:             previous.next = position.next;
155:         }
156:         position = previous;
157:     }
158:
```

**Continued**

# File LinkedList.java

```
159:         /**
160:          * Sets the last traversed element to a different
161:          * value.
162:          * @param element the element to set
163:          */
164:     public void set(E element)
165:     {
166:         if (position == null)
167:             throw new NoSuchElementException();
168:         position.data = element;
169:     }
170:
171:     private Node position;
172:     private Node previous;
173: }
174: }
```

# File ListIterator.java

```
01: /**
02:     A list iterator allows access of a position in a linked
03:     list. This interface contains a subset of the methods
04:     of the standard java.util.ListIterator interface. The
05:     methods for backward traversal are not included.
06: */
07: public interface ListIterator<E>
08: {
09:     /**
10:         Moves the iterator past the next element.
11:         @return the traversed element
12:     */
13:     E next();
14:
15:     /**
16:         Tests if there is an element after the iterator
17:         position.
```

**Continued**

# File ListIterator.java

```
18:         @return true if there is an element after the iterator
19:         position
20:     */
21:     boolean hasNext();
22:
23:     /**
24:      * Adds an element before the iterator position
25:      * and moves the iterator past the inserted element.
26:      * @param element the element to add
27:      */
28:     void add(E element);
29:
30:     /**
31:      * Removes the last traversed element. This method may
32:      * only be called after a call to the next() method.
33:      */
```

**Continued**

# File ListIterator.java

```
34:     void remove();
35:
36:     /**
37:         Sets the last traversed element to a different
38:         value.
39:         @param element the element to set
40:     */
41:     void set(E element);
42: }
```

# File ListTester.java

```
01: import java.util.LinkedList;
02: import java.util.ListIterator;
03:
04: /**
05:     A program that demonstrates the LinkedList class
06: */
07: public class ListTester
08: {
09:     public static void main(String[] args)
10:     {
11:         LinkedList<String> staff = new LinkedList<String>();
12:         staff.addLast("Dick");
13:         staff.addLast("Harry");
14:         staff.addLast("Romeo");
15:         staff.addLast("Tom");
16:
```

**Continued**

# File ListTester.java

```
17:         // | in the comments indicates the iterator position
18:
19:         ListIterator<String> iterator = staff.listIterator();
           // |DHRT
20:         iterator.next(); // D|HRT
21:         iterator.next(); // DH|RT
22:
23:         // Add more elements after second element
24:
25:         iterator.add("Juliet"); // DHJ|RT
26:         iterator.add("Nina"); // DHJN|RT
27:
28:         iterator.next(); // DHJNR|T
29:
30:         // Remove last traversed element
31:
32:         iterator.remove(); // DHJN|T
33:
```

**Continued**

# File ListTester.java

```
34:         // Print all elements
35:
36:         iterator = staff.listIterator();
37:         while (iterator.hasNext())
38:         {
39:             String element = iterator.next();
40:             System.out.println(element);
41:         }
42:     }
43: }
```

# File ListTester.java

---

- **Output:**

```
Dick  
Harry  
Juliet  
Nina  
Tom
```

# Self Check

---

1. How would you use the generic `Pair` class to construct a pair of strings `"Hello"` and `"World"`?
2. What change was made to the `ListIterator` interface, and why was that change necessary?

# Answers

---

1. 

```
new Pair<String, String>("Hello", "World")
```
2. **ListIterator<E> is now a generic type. Its interface depends on the element type of the linked list.**

# Generic Methods

- **Generic method:** method with a type variable
- **Can be defined inside ordinary and generic classes**
- **A regular (non-generic) method:**

```
/**
 * Prints all elements in an array of strings.
 * @param a the array to print
 */
public static void print(String[] a)
{
    for (String e : a)
        System.out.print(e + " ");
    System.out.println();
}
```

**Continued**

# Generic Methods

- **What if we want to print an array of Rectangle objects instead?**

```
public static <E> void print(E[] a)
{
    for (E e : a)
        System.out.print(e + " ");
    System.out.println();
}
```

# Generic Methods

- When calling a generic method, you need not instantiate the type variables:

```
Rectangle[] rectangles = . . . ;  
ArrayUtil.print(rectangles);
```

- The compiler deduces that **E** is `Rectangle`
- You can also define generic methods that are not static

*Continued*

# Generic Methods

---

- You can even have generic methods in generic classes
- Cannot replace type variables with primitive types
  - e.g.: cannot use the generic `print` method to print an array of type `int[]`

# Syntax 22.1: Instantiating a Generic Class

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

## Example:

```
public static <E> void print(E[] a)  
{  
    . . .  
}
```

## Purpose:

To define a generic method that depends on type variables

# Self Check

---

1. Exactly what does the generic `print` method print when you pass an array of `BankAccount` objects containing two bank accounts with zero balances?
2. Is the `getFirst` method of the `Pair` class a generic method?

# Answers

---

1. The output depends on the definition of the `toString` method in the `BankAccount` class.
2. No—the method has no type parameters. It is an ordinary method in a generic class.

# Constraining Type Variables

- **Type variables can be constrained with bounds**

```
public static <E extends Comparable> E min(E[] a)
{
    E smallest = a[0];
    for (int i = 1; i < a.length; i++)
        if (a[i].compareTo(smallest) < 0) smallest = a[i];
    return smallest;
}
```

*Continued*

# Constraining Type Variables

---

- Can call `min` with a `String[]` array but not with a `Rectangle[]` array
- `Comparable` bound necessary for calling `compareTo`  
Otherwise, `min` method would not have compiled

# Constraining Type Variables

- **Very occasionally, you need to supply two or more type bounds**

```
<E extends Comparable & Cloneable>
```

- **extends, when applied to type variables, actually means "extends or implements"**
- **The bounds can be either classes or interfaces**
- **Type variable can be replaced with a class or interface type**

# Self Check

---

1. Declare a generic `BinarySearchTree` class with an appropriate type variable.
2. Modify the `min` method to compute the minimum of an array of elements that implements the `Measurable` interface of Chapter 11.

# Answers

1.

```
public class BinarySearchTree <E extends Comparable>
```

2.

```
public static <E extends Measurable> E min(E[] a)
{
    E smallest = a[0];
    for (int i = 1; i < a.length; i++)
        if (a[i].getMeasure() < smallest.getMeasure()) < 0)
            smallest = a[i];
    return smallest;
}
```

# Wildcard Types

| Name                      | Syntax      | Meaning            |
|---------------------------|-------------|--------------------|
| Wildcard with lower bound | ? Extends B | Any subtype of B   |
| Wildcard with upper bound | ? Super B   | Any supertype of B |
| Unbounded wildcard        | ?           | Any type           |

# Wildcard Types

- ```
public void addAll(LinkedList<? extends E> other)
{
    ListIterator<E> iter = other.listIterator();
    while (iter.hasNext()) add(iter.next());
}
```

- ```
public static <E extends Comparable<E>> E min(E[] a)
```

- ```
public static <E extends Comparable<? super E>> E min(E[] a)
```

- ```
static void reverse(List<?> list)
```

*Continued*

# Wildcard Types

---

- You can think of that declaration as a shorthand for

```
static void <T> reverse(List<T> list)
```

# Raw Types

---

- **The virtual machine works with raw types, not with generic classes**
- **The raw type of a generic type is obtained by erasing the type variables**

*Continued*

# Raw Types

- For example, generic class `Pair<T, S>` turns into the following raw class:

```
public class Pair
{
    public Pair(Object firstElement, Object secondElement)
    {
        first = firstElement;
        second = secondElement;
    }
    public Object getFirst() { return first; }
    public Object getSecond() { return second; }
    private Object first;
    private Object second;
}
```

# Raw Types

- **Same process is applied to generic methods:**

```
public static Comparable min(Comparable[] a)
{
    Comparable smallest = a[0];
    for (int i = 1; i < a.length; i++)
        if (a[i].compareTo(smallest) < 0) smallest = a[i];
    return smallest;
}
```

*Continued*

# Raw Types

---

- **Knowing about raw types helps you understand limitations of Java generics**
- **For example, you cannot replace type variables with primitive types**
- **To interface with legacy code, you can convert between generic and raw types**

# Self Check

---

1. What is the erasure of the `print` method in Section 22.3?
2. What is the raw type of the `LinkedList<E>` class in Section 22.2?

# Answers

1.

```
public static void print(Object[] a)
{
    for (Object e : a)
        System.out.print(e + " ");
    System.out.println();
}
```

2. **The LinkedList class of Chapter 20.**