Chapter Seventeen: 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
• 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

• *Generic class*: declared with a type variable `E`

• The type variable denotes the element type:
  ```java
  public class ArrayList<E> // could use "ElementType"
  instead of E
  {
    public ArrayList() { . . . }
    public void add(E element) { . . . }
    . . .
  }
  ```
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`:

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

- **Contrast with** `LinkedList.add`:

  ```java
  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>*

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

```java
ArrayList<BankAccount> accounts1 = new ArrayList<BankAccount>();

LinkedList accounts2 = new LinkedList();

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 17.1

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.

**Answer:** `HashMap<String, Integer>`
Implementing Generic Classes

• Example: simple generic class that stores pairs of objects

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

• The `getFirst` and `getSecond` retrieve first and second values of pair

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

• 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

<table>
<thead>
<tr>
<th>Type Variable</th>
<th>Name Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Element type in a collection</td>
</tr>
<tr>
<td>K</td>
<td>Key type in a map</td>
</tr>
<tr>
<td>V</td>
<td>Value type in a map</td>
</tr>
<tr>
<td>T</td>
<td>General type</td>
</tr>
<tr>
<td>S, U</td>
<td>Additional general types</td>
</tr>
</tbody>
</table>
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;
}
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;

    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.
A linked list is a sequence of nodes with efficient element insertion and removal. This class contains a subset of the methods of the standard java.util.LinkedList class.

```java
public class LinkedList<E>
{
    public LinkedList()
    {
        first = null;
    }
    public LinkedList()
    {
        first = null;
    }
    @return the first element in the linked list
    */
Continued
public E getFirst()
{
    if (first == null)
    {
        throw new NoSuchElementException();
    }
    return first.data;
}

/**
 * Removes the first element in the linked list.
 * @return the removed element
 */
public E removeFirst()
{
    if (first == null)
    {
        throw new NoSuchElementException();
    }
    E element = first.data;
    first = first.next;
    return element;
}

/**
 * Adds an element to the front of the linked list.
 * @param element the element to add
 */
public void addFirst(E element) {
    Node newNode = new Node();
    newNode.data = element;
    newNode.next = first;
    first = newNode;
}

public ListIterator<E> listIterator() {
    return new LinkedListIterator();
}

private Node first;

private class Node {
    // Continuation
```java
public E data;
public Node next;
}
private class LinkedListIterator implements ListIterator<E>
{
/**
   * Constructs an iterator that points to the front of the linked list.
   */
public LinkedListIterator()
{
    position = null;
    previous = null;
}
/**
   * Moves the iterator past the next element.
   * @return the traversed element
   */
public E next()
{
    
```

Continued
if (!hasNext())
        throw new NoSuchElementException();
    previous = position; // Remember for remove
if (position == null)
    position = first;
else
    position = position.next;
    return position.data;
}
/**
 Tests if there is an element after the iterator position.
 @return true if there is an element after the iterator position
 */
public boolean hasNext()
{
    if (position == null)
        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:             {
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;
position = newNode;
}
previous = position;
}

/**
 * Removes the last traversed element. This method may
 * only be called after a call to the next() method.
 */
public void remove()
{
    if (previous == position)
        throw new IllegalStateException();
    if (position == first)
    {
        removeFirst();
    }
    else
    {
        previous.next = position.next;
    }
```java
public void set(E element) {
    if (position == null)
        throw new NoSuchElementException();
    position.data = element;
}
```

```java
private Node position;
private Node previous;
```
A list iterator allows access of a position in a linked list. This interface contains a subset of the methods of the standard java.util.ListIterator interface. The methods for backward traversal are not included.

```java
public interface ListIterator<E>
{
    /**
     * Moves the iterator past the next element.
     * @return the traversed element
     */
    E next();

    /**
     * Tests if there is an element after the iterator position.
     * @return true if there is an element after the iterator position
     */
    boolean hasNext();
}
```

Continued
/**
 * Adds an element before the iterator position
 * and moves the iterator past the inserted element.
 * @param element the element to add
 */
void add(E element);

/**
 * Removes the last traversed element. This method may
 * only be called after a call to the next() method.
 */
void remove();

/**
 * Sets the last traversed element to a different
 * value.
 * @param element the element to set
 */
void set(E element);
A program that tests the LinkedList class

public class ListTester
{
    public static void main(String[] args)
    {
        LinkedList<String> staff = new LinkedList<String>();
        staff.addFirst("Tom");
        staff.addFirst("Romeo");
        staff.addFirst("Harry");
        staff.addFirst("Dick");

        // | in the comments indicates the iterator position
        ListIterator<String> iterator = staff.listIterator(); // |DHRT
        iterator.next(); // D|HRT
        iterator.next(); // DH|RT

        // Add more elements after second element

        Continued
iterator.add("Juliet"); // DHJ|RT
iterator.add("Nina"); // DHJN|RT

iterator.next(); // DHJNR|T

// Remove last traversed element
iterator.remove(); // DHJN|T

// Print all elements
iterator = staff.listIterator();

while (iterator.hasNext())
{
    String element = iterator.next();
    System.out.print(element + " ");
}
System.out.println();
System.out.println("Expected: Dick Harry Juliet Nina Tom");
}
Output:

Dick Harry Juliet Nina Tom
Expected: Dick Harry Juliet Nina Tom
Self Check 17.3

How would you use the generic `Pair` class to construct a pair of strings "Hello" and "World"?

**Answer:** `new Pair<String, String>("Hello", "World")`
What change was made to the `ListIterator` interface, and why was that change necessary?

**Answer:** `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:

```java
/**
 * 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 (cont.)

• 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:

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

• The compiler deduces that `E` is `Rectangle`

• You can also define generic methods that are not static

• 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 22.3 Defining a Generic Method**

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

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

**Purpose:**
To define a generic method that depends on type variables.
Self Check 17.5

Exactly what does the generic print method \texttt{print} when you pass an array of \texttt{BankAccount} objects containing two bank accounts with zero balances?

\textbf{Answer:} The output depends on the definition of the \texttt{toString} method in the \texttt{BankAccount} class.
Self Check 17.6

Is the `getFirst` method of the `Pair` class a generic method?

**Answer:** 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

```java
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;
}
```

• 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

\(<E \text{ extends Comparable & Cloneable}>\)

• \textit{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 17.7

How would you constrain the type variable for a generic `BinarySearchTree` class?

**Answer:**

```java
public class BinarySearchTree<E extends Comparable>
```
## Wildcard Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wildcard with lower bound</td>
<td>? extends B</td>
<td>Any subtype of B</td>
</tr>
<tr>
<td>Wildcard with higher bound</td>
<td>? super B</td>
<td>Any supertype of B</td>
</tr>
<tr>
<td>Unbounded wildcard</td>
<td>?</td>
<td>Any type</td>
</tr>
</tbody>
</table>
**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.
- 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:

```java
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;
}
```

• 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 17.9

What is the erasure of the `print` method in Section 17.3?

**Answer:**

```java
public static void print(Object[] a) {
    for (Object e : a)
        System.out.print(e + " ");
    System.out.println();
}
```
Self Check 17.10

What is the raw type of the LinkedList\(<E>\) class in Section 17.2?

**Answer:** The LinkedList class of Chapter 15.