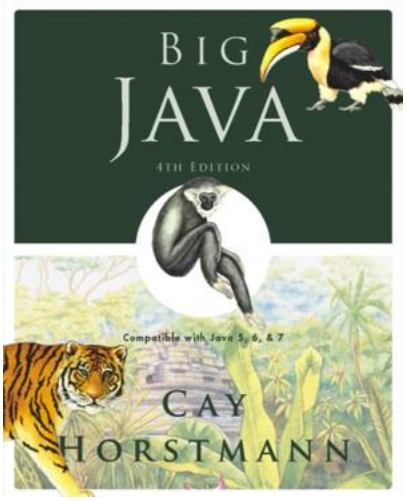


Chapter 17 – Generic Programming

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

Generic Classes and Type Parameters

- **Generic programming:** creation of programming constructs that can be used with many different types
- *Generic class*: declared with one or more type parameters
- A type parameter for `ArrayList` denotes the element type:

```
public class ArrayList<E>
{
   public ArrayList() { . . . }
   public void add(E element) { . . . }
   . . .
}
```


Type Parameters

- Can be instantiated with class or interface type:

```
ArrayList<Integer>
ArrayList<Iterable>
```

- Cannot use a primitive type as a type variable:

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

- Use corresponding wrapper class instead:

```
ArrayList<Double>
```

Type Parameters

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

```
public void add(Integer element)
```


Type Parameters Increase Safety

- Type parameters make generic code safer and easier to read.
- *Impossible to add a `String` into an `ArrayList<Integer>`*

Class `Pair`

```
public class Pair<T, S>
{
   private T first;
   private S second;

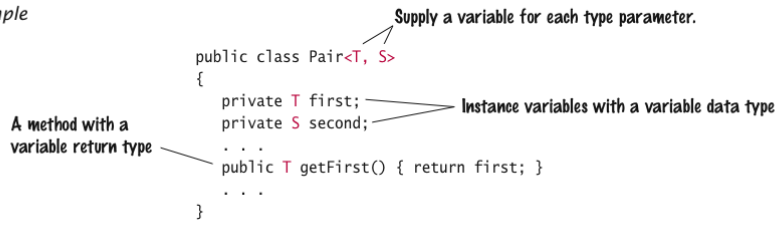
   public Pair(T firstElement, S secondElement)
   {
      first = firstElement;
      second = secondElement;
   }
   public T getFirst() { return first; }
   public S getSecond() { return second; }
}
```


Syntax 17.1 Declaring a Generic Class

Syntax

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

Example



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")
```


Self Check 17.4

What can you store in the following data structure?

```
ArrayList<Pair<String, Integer>>
```

Example: `[(Tom, 1), (Harry, 3)]`.

Self Check 17.4

What can you store in the following data structure?

`Pair<ArrayList<String>, Integer>`?

Example: `([Tom, Harry], 1)`.

Generic Methods

- **Generic method:** method with a type variable
- Can be defined inside non-generic classes
- Example: Want to declare a method that can print an array of any type:

```
public class ArrayUtil
{
   /** Prints all elements in an array.
   *   @param a the array to print
   */
   public <T> static void print(T[] a)
   {
      . . .
   }
}
```

Generic Methods

Often easier to see how to implement a generic method by starting with a concrete example; e.g. print the elements in an array of *strings*:

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


Generic Methods

- In order to make the method into a generic method:
  - *Replace* `String` *with a type parameter, say* `E`*, to denote the element type*
  - *Supply the type parameters between the method's modifiers and return type*

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

Generic Methods

- What happens when you call a generic method?

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

- The compiler deduces that `E` is `Rectangle`
- You can also define generic methods that are not static
- Cannot replace type variables with primitive types
  e.g.: cannot use the generic `print` method to print an array of type `int[]`

Syntax 17.2 Defining a Generic Method

Syntax

modifiers <*TypeVariable*$_1$, *TypeVariable*$_2$, . . .> *returnType methodName*(*parameters*)
{
  body
}

Example

Supply the type variable before the return type.

Local variable with a variable data type

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

Type instantiation

- You cannot instantiate generic types:

```
public static <E> void fillWithDefaults(E[] a)
{
   for (int i = 0; i < a.length; i++)
      a[i] = new E(); // ERROR
}
```

- You cannot construct an array of a generic type:

```
public class Stack<E>
{
   private E[] elements;
   . . .
   public Stack()
   {
      elements = new E[MAX_SIZE]; // Error
   }
}
```


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;
}
```

- 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

Wildcard Types

Name	Syntax	Meaning
Wildcard with lower bound	`? extends B`	Any subtype of `B`
Wildcard with higher bound	`? super B`	Any supertype of `B`
Unbounded wildcard	`?`	Any type

Examples of 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)
  ```
  ```