Chapter 18 - Generic Classes

Chapter Goals

© Don Bayley/iStockphoto.

- To understand the objective of generic programming
- To implement generic classes and methods
- To explain the execution of generic methods in the virtual machine
- To describe 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.
  - In Java, achieved with type parameters or with inheritance
  - Type parameter example: Java's `ArrayList` (e.g. `ArrayList<String>`)
  - Inheritance example: `LinkedList` implemented in Section 16.1 can store objects of any class
- **Generic class:** has one or more type parameters.
- A type parameter for `ArrayList` denotes the element type:

```
public void add( E
element)  public  E
get(int index)
```

Type Parameter

- Can be instantiated with class or interface type:

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

- Cannot use a primitive type as a type parameter:

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

- Use corresponding wrapper class instead:

```
ArrayList<Double>
```

Type Parameters

- 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` from Chapter 16:

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

Type Parameters Increase Safety

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

Self Check 18.1

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>`

Self Check 18.2

The binary search tree class in Chapter 17 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 parameters?

Answer: It uses inheritance.

Self Check 18.3

Does the following code contain an error? If so, is it a compile-time or run-time error?

```
ArrayList<Integer> a = new ArrayList<>();
String s = a.get(0);
```

Answer: This is a compile-time error. You cannot assign the `Integer` expression `a.get(0)` to a string.

Self Check 18.4

Does the following code contain an error? If so, is it a compile-time or run-time error?

```
ArrayList<Double> a = new ArrayList<>();
a.add(3);
```

Answer: This is a compile-time error. The compiler won't convert 3 to a `Double`. Remedy: Call `a.add(3.0)`.

Self Check 18.5

Does the following code contain an error? If so, is it a compile-time or run-time error?

```
LinkedList a = new LinkedList();
a.addFirst("3.14");
double x = (Double) a.removeFirst();
```

Answer: This is a run-time error. `a.removeFirst()` yields a `String` that cannot be converted into a `Double`.

Remedy: Call `a.addFirst(3.14);`

Implementing Generic Classes

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

```
Pair<String, Integer> result
      = new Pair<>("Harry Hacker", 1729);
```

- Methods `getFirst` and `getSecond` retrieve first and second values of pair:

```
String name = result.getFirst();
Integer number =
   result.getSecond();
```

- Example of use: for a method that computes two values at the same time (method returns a `Pair<String, Integer>`).
- Generic `Pair` class requires two type parameters, one for each element type enclosed in angle brackets:

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

Implementing Generic Types

- Use short uppercase names for type variables.
- Examples

Type Variable	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

- Place the type variables for a generic class after the class name, enclosed in angle brackets (< and >):

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

- When you declare the instance variables and methods of the `Pair` class, use the variable `T` for the first element type and `S` for the second element type.
- Use type parameters for the types of generic instance variables, method parameter variables, and return values.

Class Pair

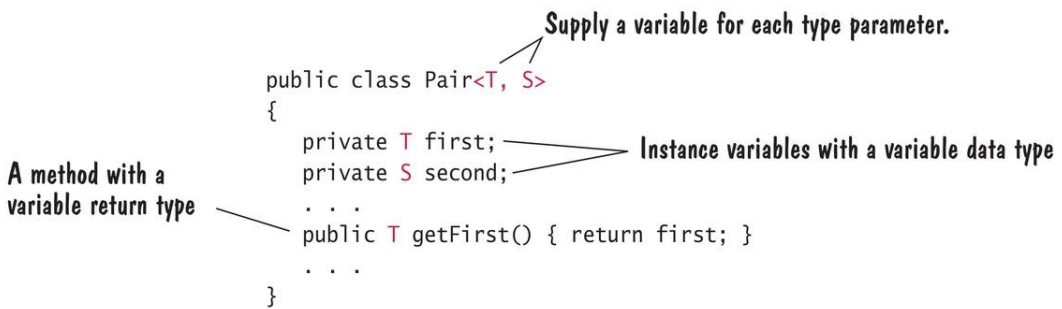
```
public class Pair<T, S>
{
   private T first;
   private S second;

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

Syntax 18.1 Declaring a Generic Class

Syntax

modifier `class` *GenericClassName*`<`*TypeVariable*$_1$`,` *TypeVariable*$_2$`, . . .>`
`{`
instance variables
constructors
methods
`}`



section_2/Pair.java

```
1       /**
2       This class collects a pair of elements of different types.
3       */
4       public class Pair<T, S>
5       {
6      private T first;
7      private S second;
8      /**
9
```

section_2/PairDemo.java

```
1  public class PairDemo
2  {
3  public static void main(String[] args)
4  {
5        String[] names = { "Tom", "Diana", "Harry" };
6        Pair<String, Integer> result = firstContaining(names, "a");
7        System.out.println(result.getFirst());
8        System.out.println("Expected: Diana");
9        System.out.println(result.getSecond());
```

Program Run:

```
Diana
Expected:
Diana  1
Expected: 1
```

Self Check 18.6

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 18.7

How would you use the generic `Pair` class to construct a pair containing `"Hello"` and `1729`?

Answer: `new Pair<String, Integer>("Hello", 1729)`

Self Check 18.8

What is the difference between an `ArrayList<Pair<String, Integer>>` and a `Pair<ArrayList<String>, Integer>`?

Answer: An `ArrayList<Pair<String, Integer>>` contains multiple pairs, for example `[(Tom, 1), (Harry, 3)]`. A `Pair<ArrayList<String>, Integer>` contains a list of strings and a single integer, such as `([Tom, Harry], 1)`.

Self Check 18.9

Write a method `roots` with a `Double` parameter variable `x` that returns both the positive and negative square root of `x` if `x` ≥ `0` or `null` otherwise.

Answer:

```
public static Pair<Double, Double> roots(Double x)
{
   if (x >= 0)
   {
      double r = Math.sqrt(x);
      return new Pair<Double, Double>(r, -r);
   }
   else { return null; }
}
```

Self Check 18.10

How would you implement a class `Triple` that collects three values of arbitrary types?

Answer: You have three type parameters: `Triple<T, S, U>`. Add an instance variable `U` third, a constructor argument for initializing it, and a method `U getThird()` for returning it.

Generic Methods

- **Generic method:** method with a type parameter.
- Can be declared inside non-generic class.
- Example: 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.
- Example: 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.

  Add 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

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

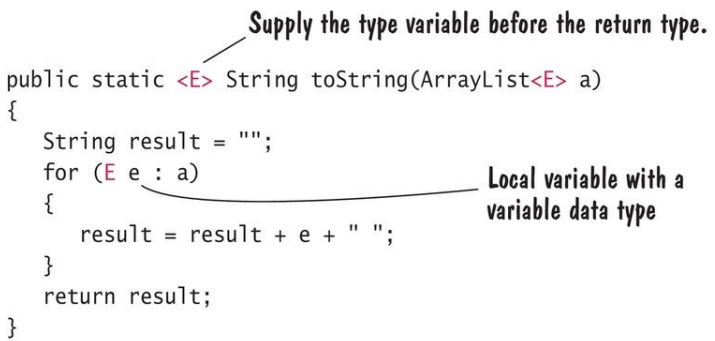
```
Rectangle[] rectangles = . .
.;
ArrayUtil.print(rectangles);
```

- The compiler deduces that `E` is `Rectangle`.
- You can also define generic methods that are not static.
- You can even have generic methods in generic classes.
- Cannot replace type variables with primitive types.

  Example: cannot use the generic `print` method to print an array of type `int[]`

Syntax 18.2 Declaring a Generic Method

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



Self Check 18.11

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

Answer: The output depends on the definition of the `toString` method in the `BankAccount` class.

Self Check 18.12

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.

Self Check 18.13

Consider this `fill` method:

```
public static <T> void fill(List<T> lst, T value)
{
    for (int i = 0; i < lst.size(); i++) { lst.set(i, value); }
}
```

If you have an array list `ArrayList <String> a = new ArrayList<String>(10);` how do you fill it with ten `"*"`?

Answer: `fill(a, "*");`

Self Check 18.14

What happens if you pass `42` instead of `"*"` to the fill method?

Answer: You get a compile-time error. An integer cannot be converted to a string.

Self Check 18.15

Consider this fill method:

```
public static <T> fill(T[] arr, T value)
{
   for (int i = 0; i < arr.length; i++) { arr[i] = value; }
}
```

What happens when you execute the following statements?

```
String[] a = new String[10];
fill(a, 42);
```

Answer: You get a run-time error. Unfortunately, the call compiles, with `T = Object`. This choice is justified because a `String[]` array is convertible to an `Object[]` array, and 42 becomes `new Integer(42)`, which is convertible to an `Object`. But when the program tries to store an `Integer` in the `String[]` array, an exception is thrown.

Constraining Type Variables

You can place restrictions on the type parameters of generic classes and methods.

Constraining Type Variables

- Type variables can be constrained with bounds.
- A generic method, `average`, needs to be able to measure the objects.
- Measurable interface from Section 10.1:

```
public interface Measurable
{
   double getMeasure();
}
```

- We can constrain the type of the elements to those that implement the `Measurable` type:

```
public static < E extends Measurable > double average(ArrayList<E> objects)
```

- This means, “E or one of its superclasses extends or implements Measurable”.

We say that `E` is a subtype of the `Measurable` type.

Constraining Type Variables

- Completed `average` method:

```
public static <E extends Measurable> double average(ArrayList<E> objects)
{
   if (objects.size() == 0) { return 0;
   }  double sum = 0;
   for (E obj : objects)
   {
      sum = sum + obj.getMeasure();
   }
   return sum / objects.size();
}
```

- In the call `obj.getMeasure()`

  It is legal to apply the `getMeasure` method to `obj`.

  `obj` has type `E`, and `E` is a subtype of `Measurable`.

Constraining Type Variables - Comparable Interface

- `Comparable` interface is a generic type.
- The type parameter specifies the type of the parameter variable of the `compareTo` method:

```
public interface Comparable<T>
{
   int compareTo(T other);
}
```

- `String` class implements `Comparable<String>`

  A String can be compared to other `String`.

  But not with objects of a different class.

Constraining Type Variables - Comparable Interface

- When writing a generic method `min` to find the smallest element in an array list,

  Require that type parameter `E` implements `Comparable<E>`

```
public static <E extends Comparable<E>> E min(ArrayList<E> objects)
{
   E smallest = objects.get(0);
   for (int i = 1; i < objects.size(); i++)
   {
      E obj = objects.get(i);
      if (obj.compareTo(smallest) < 0)
      {
         smallest = obj;
      }
   }
   return smallest;
}
```

- Because of the type constraint, `obj` must have a method of this form:

```
int compareTo(E other)
```

  So the the following call is valid:

```
obj.compareTo(smallest)
```

Constraining Type Variables

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

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

- `extends`, when applied to type parameters, actually means “extends or implements.”
- The bounds can be either classes or interfaces.
- Type parameters can be replaced with a class or interface type.

Self Check 18.16

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

Answer:

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

or, if you read Special Topic 18.1,

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

Self Check 18.17

Modify the `min` method to compute the minimum of an array list of elements that implements the `Measurable` interface.

Answer:

```
public static <E extends Measurable> E min(ArrayList<E> objects)
{
   E smallest = objects.get(0);
   for (int i = 1; i < objects.size(); i++)
   {
      E obj = objects.get(i);
      if (obj.getMeasure() < smallest.getMeasure())
      {
         smallest = obj;
      }
   }
   return smallest;
}
```

Self Check 18.18

Could we have declared the `min` method of Self Check 17 without type parameters, like this?

```
public static Measurable min(ArrayList<Measurable> a)
```

Answer: No. As described in Common Error 18.1, you cannot convert an `ArrayList<BankAccount>` to an `ArrayList<Measurable>`, even if `BankAccount` implements `Measurable`.

Self Check 18.19

Could we have declared the `min` method of Self Check 17 without type parameters for arrays, like this?

```
public static Measurable min(Measurable[] a)
```

Answer: Yes, but this method would not be as useful. Suppose accounts is an array of `BankAccount` objects. With this method, `min(accounts)` would return a result of type `Measurable`, whereas the generic method yields a `BankAccount`.

Self Check 18.20

How would you implement the generic `average` method for arrays?

Answer:

```
public static <E extends Measurable> double average(E[] objects)
{
   if (objects.length == 0) { return 0;
   }  double sum = 0;
   for (E obj : objects)
   {
      sum = sum + obj.getMeasure();
   }
   return sum / objects.length;
}
```

Self Check 18.21

Is it necessary to use a generic average method for arrays of measurable objects?

Answer: No. You can define

```
public static double average(Measurable[] objects)
{
   if (objects.length == 0) { return 0;
   }  double sum = 0;
   for (Measurable obj : objects)
   {
      sum = sum + obj.getMeasure();
   }
   return sum / objects.length;
```

For example, if `BankAccount` implements `Measurable`, a `BankAccount[]` array is convertible to a `Measurable[]` array. Contrast with Self Check 19, where the return type

was a generic type. Here, the return type is `double`, and there is no need for using generic types.

Genericity and Inheritance

- Common Error 18.1: One can not assign a subclass list to a superclass list.
- `ArrayList<SavingsAccount>` is not a subclass of `ArrayList<BankAccount>`.

  Even though `SavingsAccount` is a subclass of `BankAccount`

```
ArrayList<SavingsAccount> savingsAccounts = new
ArrayList<SavingsAccount>();  ArrayList<BankAccount> bankAccounts =
savingsAccounts;
    // Not legal - compile-time error
```

- Common Error 18.2: However, you can do the equivalent thing with arrays:

```
SavingsAccount[] savingsAccounts = new
SavingsAccount[10];  BankAccount bankAccounts =
savingsAccounts; // Legal
```

- But this assignment will give a run-time error:

```
BankAccount harrysChecking = new CheckingAccount();
bankAccounts[0] = harrysChecking; // Throws
ArrayStoreException
```

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

- Wildcard types are used to formulate subtle constraints on type parameters.
- A wildcard type is a type that can remain unknown.
- A method in a `LinkedList` class to add all elements of `LinkedList other`:

  `other` can be of any subclass of `E`.

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

- This declaration is too restrictive for the `min` method:

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

- Type parameter of the `Comparable` interface should be any supertype of the array list's element type:

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

Wildcard Types

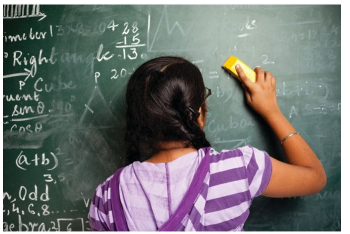
- A method in the `Collections` class which uses an unbounded wildcard:

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

- You can think of that declaration as a shorthand for:

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

Type Erasure



© VikramRaghuvanshi/iStockphoto.

In the Java virtual machine, generic types are erased.

Type Erasure

- The virtual machine erases type parameters, replacing them with their bounds or `Objects`.
- For example, generic class `Pair<T, S>` turns into the following raw class:

```
public class Pair
{
   private Object first;
   private Object
   second;


   public Pair( Object firstElement, Object secondElement)
   {
      first = firstElement;
      second =
      secondElement;

   }
   public Object getFirst() { return first; }
   public Object getSecond() { return second;
   }

}
```

Type Erasure

- Same process is applied to generic methods.
- In this generic method:

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

- The type parameter Is replaced with its bound `Measurable`:

```
public static Measurable min(Measurable[]
objects)
{
   Measurable smallest = objects[0];
   for (int i = 1; i < objects.length; i++)
   {
      Measurable obj = objects[i];
```

```
        {
            smallest =
            obj;
        }
    }
    return smallest;
}
```

Type Erasure

- Knowing about type erasure helps you understand limitations of Java generics.
- You cannot construct new objects of a generic type.
- For example, trying to fill an array with copies of default objects would be wrong:

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

- Type erasure yields:

```
public static void fillWithDefaults( Object[] a)
{
   for (int i = 0; i < a.length; i++)
      a[i] = new Object(); // Not
      useful
}
```

Type Erasure

- To solve this particular problem, you can supply a default object:

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

Type Erasure

- You cannot construct an array of a generic type:

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

- Because the array construction expression `new E[]` would be erased to `new Object[]`.
- One remedy is to use an array list instead:

```
public class Stack<E>
{
   private ArrayList<E> elements;
   . . .
   public Stack()
   {
      elements = new ArrayList<E>(); // Ok
   }
   . . .
}
```

Type Erasure

- Another solution is to use an array of objects and cast when reading elements from the array:

```
public class Stack<E>
{
   private Object[]
   elements;  private int
   currentSize;
   . . .
   public Stack()
   {
      elements = new Object[MAX_SIZE]; // Ok
   }
   . . .
   public E pop()
   {
      size--;
      return (E) elements[currentSize];
   }
}
```

- The cast `(E)` generates a warning because it cannot be checked at compile time.

Self Check 18.22

Suppose we want to eliminate the type bound in the `min` method of Section 18.5, by declaring the parameter variable as an array of `Comparable<E>` objects. Why doesn't this work?

Answer:

```
public static <E> Comparable<E> min(Comparable<E>[] objects)
```

is an error. You cannot have an array of a generic type.

Self Check 18.23

What is the erasure of the print method in Section 18.3?

Answer:

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

Self Check 18.24

Could the `Stack` example be implemented as follows?

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

Answer: This code compiles (with a warning), but it is a poor technique. In the future, if type erasure no longer happens, the code will be *wrong*. The cast from `Object[]` to `String[]` will cause a class cast exception.

Self Check 18.25

The `ArrayList<E>` class has a method:

```
Object[] toArray()
```

Why doesn't the method return an `E[]`?

Answer: Internally, `ArrayList` uses an `Object[]` array. Because of type erasure, it can't make an `E[]` array. The best it can do is make a copy of its internal `Object[]` array.

Self Check 18.26

The `ArrayList<E>` class has a second method:

```
E[] toArray(E[] a)
```

Why can this method return an array of type `E[]`? (*Hint:* Special Topic 18.2.)

Answer: It can use reflection to discover the element type of the parameter `a`, and then construct another array with that element type (or just call the `Arrays.copyOf` method).

Self Check 18.27

Why can't the method

```
static <T> T[] copyOf(T[] original, int newLength)
```

be implemented without reflection?

Answer: The method needs to construct a new array of type `T`. However, that is not possible in Java without reflection.