Week 12

Generic Classes

(Chapter 18)

Chapter Goals

- 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
```

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 (< and >):

```
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

Implementing Generic 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

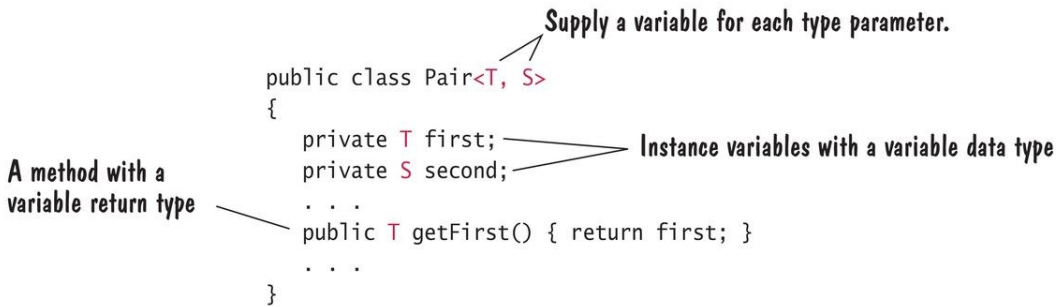
```
public class Pair<T, S>
{
  private T first;
  private S second;

  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
```

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:

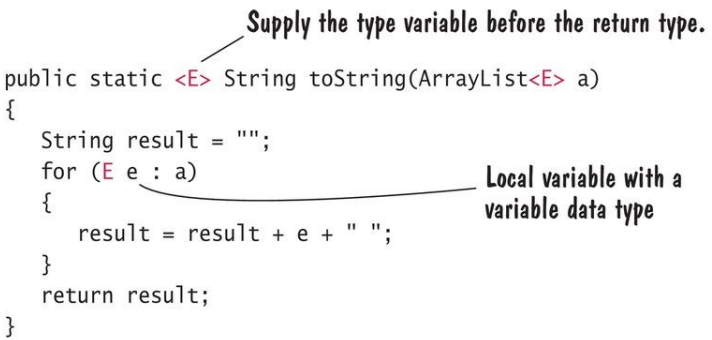
```
Rectangle[] rectangles = . . .;
ArrayUtil.print(rectangles);
```

- 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
}



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.