LECTURE 10

JAVA GENERICS

Ian Stavness, CMPT 270, 2017-18 Term 1

Copyright © 2013 by John Wiley & Sons. All rights reserved.

18.1 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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

18.1 Generic Classes and Type Parameters

- *Generic class*: declared with one or more type parameters
- E.g., standard library declares class `ArrayList<E>`
  - `E` is the type variable that denotes the element type
  - Same variable name used in method declarations, e.g.

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


Copyright © 2013 by John Wiley & Sons. All rights reserved.

Type Parameters (1)

- Can be instantiated with class or interface type:

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

- Cannot use a primitive type as a type variable:

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

- Use corresponding wrapper class instead:

```
ArrayList<Double>
```


Copyright © 2013 by John Wiley & Sons. All rights reserved.

Type Parameters (2)

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

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

- Contrast with `add` method of class `LinkedList` from Chapter 16:

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

```
public class LinkedList
 {
...
    private class Node
    {
       public Object data;
       public Node next;
    }
}
```


Copyright © 2013 by John Wiley & Sons. All rights reserved.

Type Parameters Increase Safety

- Type parameters make generic code safer and easier to read
- e.g.
  - Impossible to add a `String` into an `ArrayList<BankAccount>`
  - Can add a `String` into a `LinkedList` intended to hold bank accounts

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Type Parameters Increase Safety (2)

```
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
```


Copyright © 2013 by John Wiley & Sons. All rights reserved.

18.2 Implementing Generic Types

- Example: simple generic class that stores *pairs* of objects; e.g.

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

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

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


Copyright © 2013 by John Wiley & Sons. All rights reserved.

Implementing Generic Types

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

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


Copyright © 2013by John Wiley & Sons. All rights reserved.

Good Variable Names

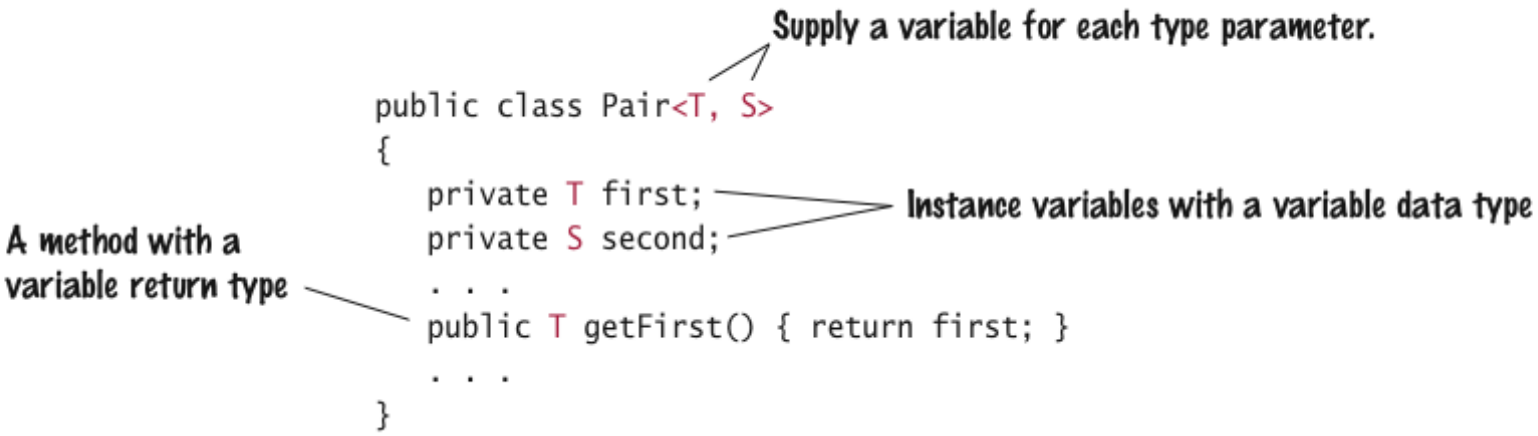
Type Variable	Meaning
`E`	Element type in a collection
`K`	Key type in a map
`V`	Value type in a map
`T`	General type
`S, U`	Additional general types

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Syntax 18.1 Declaring a Generic Class

Syntax

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



Copyright © 2013 by John Wiley & Sons. All rights reserved.

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     /**
10       Constructs a pair containing two given elements.
11       @param firstElement the first element
12       @param secondElement the second element
13    */
14    public Pair(T firstElement, S secondElement)
15    {
16       first = firstElement;
17       second = secondElement;
18    }
19
```

Continued

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Pair.java (cont.)

```
20      /**
21         Gets the first element of this pair.
22         @return the first element
23      */
24      public T getFirst() { return first; }
25
26      /**
27         Gets the second element of this pair.
28         @return the second element
29      */
30      public S getSecond() { return second; }
31
32      public String toString() {
            return "(" + first + ", " + second + ")"; }
33  }
```


Copyright © 2013 by John Wiley & Sons. All rights reserved.

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());
10       System.out.println("Expected: 1");
11    }
12
```

Continued

Copyright © 2013 by John Wiley & Sons. All rights reserved.

PairDemo.java (cont.)

```
13      /**
14         Gets the first String containing a given string, together
15         with its index.
16         @param strings an array of strings
17         @param sub a string
18         @return a pair (strings[i], i) where strings[i] is the first
19         strings[i] containing str, or a pair (null, -1) if there is no
20         match.
21      */
22      public static Pair<String, Integer> firstContaining(
23         String[] strings, String sub)
24      {
25         for (int i = 0; i < strings.length; i++)
26         {
27            if (strings[i].contains(sub))
28            {
29               return new Pair<String, Integer>(strings[i], i);
30            }
31         }
32         return new Pair<String, Integer>(null, -1);
33      }
34   }
```

Continued

Copyright © 2013 by John Wiley & Sons. All rights reserved.

PairDemo.java (cont.)

Program Run:

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


Copyright © 2013 by John Wiley & Sons. All rights reserved.

18.3 Generic Methods

- **Generic method:** method with a type parameter
- 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)
   {
      . . .
   }
   . . .
}
```


Copyright © 2013 by John Wiley & Sons. All rights reserved.

Generic Methods (2)

- 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();
   }
   . . .
}
```


Copyright © 2013 by John Wiley & Sons. All rights reserved.

Generic Methods (3)

- 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();
}
```


Copyright © 2013 by John Wiley & Sons. All rights reserved.

Generic Methods (4)

- When calling a generic method, you need not instantiate the type parameters; e.g.

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

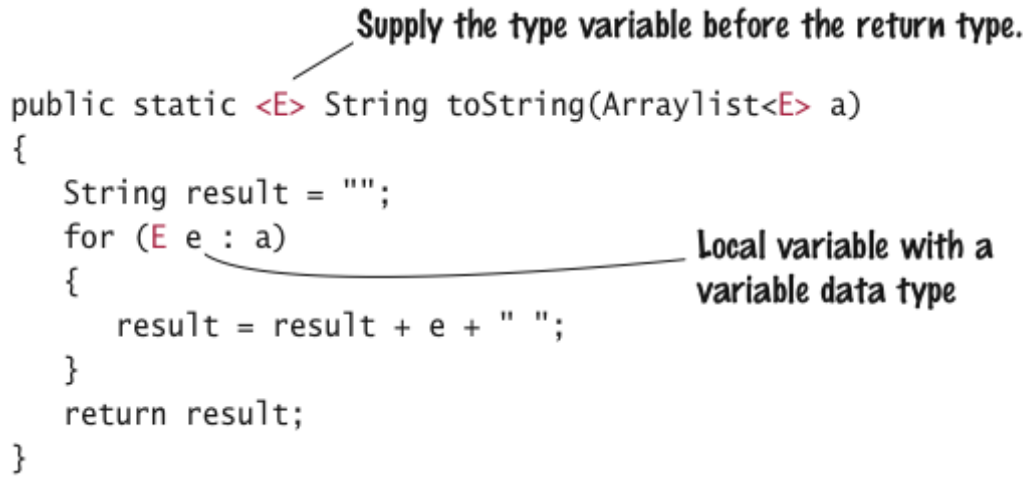
  - The compiler deduces that `E` is `Rectangle`
- You can also define generic methods that are not static
- You can even have generic methods in generic classes
- Cannot replace type variables with primitive types
  - e.g. cannot use the generic `print` method to print an array of type `int[]`
  - Implement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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Syntax 18.2 Declaring a Generic Method

Syntax

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



Copyright © 2013 by John Wiley & Sons. All rights reserved.

18.4 Constraining Type Parameters

- Type parameters can be constrained with bounds; e.g.

```
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;
}
```


Copyright © 2013 by John Wiley & Sons. All rights reserved.

Constraining Type Parameters

- Very occasionally, you need to supply two or more type bounds; e.g.

```
<E extends Comparable<E> & Measurable>
```

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

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

Copyright © 2013 by John Wiley & Sons. All rights reserved.

18.5 Type Erasure

- The virtual machine erases type parameters, replacing them with their bounds or `Object`s
- E.g., 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; }
}
```


Copyright © 2013 by John Wiley & Sons. All rights reserved.

Type Erasure (2)

- Same process is applied to generic methods:

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


Copyright © 2013 by John Wiley & Sons. All rights reserved.

Type Erasure (3)

- Knowing about raw types helps you understand limitations of Java generics
- E.g., 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
}
```


Copyright © 2013 by John Wiley & Sons. All rights reserved.

Type Erasure (4)

- 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;
   }
}
```


Copyright © 2013 by John Wiley & Sons. All rights reserved.

Type Erasure (5)

- 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[]`

Copyright © 2013 by John Wiley & Sons. All rights reserved.

Type Erasure (6)

- A remedy is to use an array list instead:

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


Copyright © 2013 by John Wiley & Sons. All rights reserved.