Cape Peninsula
University of Technology

Software Design 2
SDN260S

Generic Classes and Methods

H. Mataifa

Department of Electronic, Electrical and Computer Engineering

Outline

- Background
- Generic Methods
- Generic Classes
- Raw Types
- Wildcards in Methods that Accept Type Parameters
- Generics and Inheritance

Background

- **Generics** (**Methods, Classes, Interfaces**):
  - Enable one to specify, with a single declaration, a set of related methods/classes/interfaces
  - Provides an alternative (and often more efficient) way of achieving polymorphism (i.e. an object's ability to respond differently in different environments)
  - Generics provide compile-time type safety, enabling catching of invalid types at compile time
  - Generics are among Java's most powerful capabilities for software reuse and compile-time type safety

- Why **Generic Methods**?
  - Useful when it's required to do identical operations on different argument types; a single generic method can be declared that can be called with arguments of different types
  - Same effect can be achieved by overloading methods (i.e. assigning different signatures to a given method), but is less flexible and less efficient
  - The argument type is then resolved by the compiler at compile time

Overloaded Methods vs. Generic Method

```java
// Fig. 21.1: OverloadedMethods.java
// Printing array elements using overloaded methods.
public class OverloadedMethods
{
   public static void main( String[] args )
   {
      // create arrays of Integer, Double and Character
      Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
      Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
      Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };

      System.out.println( "Array integerArray contains:" );
      printArray( integerArray ); // pass an Integer array
      System.out.println( "\nArray doubleArray contains:" );
      printArray( doubleArray ); // pass a Double array
      System.out.println( "\nArray characterArray contains:" );
      printArray( characterArray ); // pass a Character array
   } // end main

   // method printArray to print Integer array
   public static void printArray( Integer[] inputArray )
   {
      // display array elements
      for ( Integer element : inputArray )
         System.out.printf( "%s ", element );

      System.out.println();
   } // end method printArray

   // method printArray to print Double array
   public static void printArray( Double[] inputArray )
   {
      // display array elements
      for ( Double element : inputArray )
         System.out.printf( "%s ", element );

      System.out.println();
   } // end method printArray

   // method printArray to print Character array
   public static void printArray( Character[] inputArray )
   {
      // display array elements
      for ( Character element : inputArray )
         System.out.printf( "%s ", element );

      System.out.println();
   } // end method printArray
} // end class OverloadedMethods
```

Here we are using a generic type T rather than different data type

Generic method

```java
public static void printArray( T[] inputArray )
{
   // display array elements
   for ( T element : inputArray )
      System.out.printf( "%s ", element );

   System.out.println();
} // end method printArray
```

In this code we dont need 3 different so that we can operate in different types of data

All the methods have been repalced bt one method

```java
// Fig. 21.3: GenericMethodTest.java
// Printing array elements using generic method printArray.

public class GenericMethodTest
{
   public static void main( String[] args )
   {
      // create arrays of Integer, Double and Character
      Integer[] intArray = { 1, 2, 3, 4, 5 };
      Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
      Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

      System.out.println( "Array integerArray contains:" );
      printArray( integerArray ); // pass an Integer array
      System.out.println( "\nArray doubleArray contains:" );
      printArray( doubleArray ); // pass a Double array
      System.out.println( "\nArray characterArray contains:" );
      printArray( characterArray ); // pass a Character array
   } // end main

   // generic method printArray
   public static < T > void printArray( T[] inputArray )
   {
      // display array elements
      for ( T element : inputArray )
         System.out.printf( "%s ", element );

      System.out.println();
   } // end method printArray
} // end class GenericMethodTest
```

Overloaded Methods vs. Generic Method

```java
1  // Fig. 21.3: GenericMethodTest.java
2  // Printing array elements using generic method printArray.
3
4  public class GenericMethodTest
5  {
6     public static void main( String[] args )
7     {
8        // create arrays of Integer, Double and Character
9        Integer[] intArray = { 1, 2, 3, 4, 5 };
10       Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
11       Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
12
13       System.out.println( "Array integerArray contains:" );
14       printArray( integerArray ); // pass an Integer array
15       System.out.println( "\nArray doubleArray contains:" );
16       printArray( doubleArray ); // pass a Double array
17       System.out.println( "\nArray characterArray contains:" );
18       printArray( characterArray ); // pass a Character array
19    } // end main
20
21    // generic method printArray
22    public static < T > void printArray( T[] inputArray )
23    {
24       // display array elements
25       for ( T element : inputArray )
26          System.out.printf( "%s ", element );
27
28       System.out.println();
29    } // end method printArray
30 } // end class GenericMethodTest
```

,<T> tells the compiler taht when ever you come across T it should treat is as a generic type

T here is a paramter type here

Here T is a local variable type

Notes:

- Every generic method declaration has a type-parameter section (<**T**>, line 22) preceding the return type; can be one or more type parameters, comma-separated
- A type parameter is an identifier that specifies a generic type name; can be used to declare return type, parameter types, and local variable types
- Parameter types can represent only reference (and no primitive) types

meaning it wont work int or double all you have to do is use a wrapper like Integer or Double

- A syntax error occurs when a generic method declaration doesn't have a type-parameter section
- A type parameter is typically specified as a capital letter (**T** in the example)

Generic Methods: Implementation and Compile-Time Issues

- **Compilation errors** occur when:
  - The compiler cannot match a method call to a non-generic or a generic method declaration
  - The compiler doesn't find a method declaration that matches a method call exactly, but does find two or more methods that can satisfy the method call
- **Compile-time translation**:
  - When the compiler translates a generic method into Java bytecode, it removes the type-parameter section and replaces type parameters with actual types, a process known as **erasure**
  - By default, all generic types are replaced with **Object**

PrintArray before compilation

```java
    // generic method printArray
    public static < T > void printArray( T[] inputArray )
    {
       // display array elements
       for ( T element : inputArray )
          System.out.printf( "%s ", element );

       System.out.println();
    } // end method printArray
 } // end class GenericMethodTest
```

PrintArray after compilation

```java
public static void printArray( Object[] inputArray )
{
   // display array elements
   for ( Object element : inputArray )
      System.out.printf( "%s ", element );

   System.out.println();
} // end method printArray
```

Generic Method maximum

```java
// Fig. 21.5: MaximumTest.java
// Generic method maximum returns the largest of three objects.

public class MaximumTest
{
   public static void main( String[] args )
   {
      System.out.printf( "Maximum of %d, %d and %d is %d\n\n", 3, 4, 5,
         maximum( 3, 4, 5 ) );
      System.out.printf( "Maximum of %.1f, %.1f and %.1f is %.1f\n\n",
         6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );
      System.out.printf( "Maximum of %s, %s and %s is %s\n", "pear",
         "apple", "orange", maximum( "pear", "apple", "orange" ) );
   } // end main

   // determines the largest of three Comparable objects
   public static < T extends Comparable< T > > T maximum( T x, T y, T z )
   {
      T max = x; // assume x is initially the largest

      if ( y.compareTo( max ) > 0 )
         max = y; // y is the largest so far

      if ( z.compareTo( max ) > 0 )
         max = z; // z is the largest

      return max; // returns the largest object
   } // end method maximum
} // end class MaximumTest
```

Type parameter section
Extends keyword is for inheretence

Notes:

- Generic method **maximum** returns largest of three **Objects**, uses generic interface **Comparable<T>**’s **CompareTo** method
- Type-parameter **T** extends **Comparable<T>**, to enable the generic method to use **CompareTo** method
- **Comparable** is known as the upper bound of type-parameter **T** in generic method **maximum**; when the compiler performs **erasure**, it replaces every type-parameter instance with **Comparable**

Generic Method maximum

```java
public static Comparable maximum(Comparable x, Comparable y, Comparable z)
{
   Comparable max = x; // assume x is initially the largest

   if ( y.compareTo( max ) > 0 )
      max = y; // y is the largest so far

   if ( z.compareTo( max ) > 0 )
      max = z; // z is the largest

   return max; // returns the largest object
} // end method maximum
```

- **Overloading Generic Methods**:
  - A generic method may be overloaded, where a class provides two or more generic methods that specify the same method name but different method parameters
  - A generic method can also be overloaded by a non-generic method
  - When a compiler encounters a method-call, it searches for the method declaration that most precisely matches the method name and the argument types specified in the call

Generic Classes

- **Generic classes** provide a means for defining classes in a type-independent manner:
  - A data structure such as a **Stack** can be defined independently of the element type it manipulates
- Generic classes are also known as parameterized classes or parameterized types; they can accept one or more different type parameters

Generic Class Stack

```
// Fig. 21.7: Stack.java
// Stack generic class declaration.
import java.util.ArrayList;

public class Stack< T >
{
   private ArrayList< T > elements; // ArrayList stores stack elements

   // no-argument constructor creates a stack of the default size
   public Stack()
   {
      this( 10 ); // default stack size
   } // end no-argument Stack constructor

   // constructor creates a stack of the specified number of elements
   public Stack( int capacity )
   {
      int initCapacity = capacity > 0 ? capacity : 10; // validate
      elements = new ArrayList< T >( initCapacity ); // create ArrayList
   } // end one-argument Stack constructor

   // push element onto stack
   public void push( T pushValue )
   {
      elements.add( pushValue ); // place pushValue on Stack
   } // end method push

   // return the top element if not empty; else throw EmptyStackException
   public T pop()
   {
      if ( elements.isEmpty() ) // if stack is empty
         throw new EmptyStackException( "Stack is empty, cannot pop" );

      // remove and return top element of Stack
      return elements.remove( elements.size() - 1 );
   } // end method pop
} // end class Stack< T >
```

Class name must be followed by a type parameter

Generic Class Stack (EmptyStackException)

```java
// Fig. 21.8: EmptyStackException.java
// EmptyStackException class declaration.
public class EmptyStackException extends RuntimeException
{
   // no-argument constructor
   public EmptyStackException()
   {
      this( "Stack is empty" );
   } // end no-argument EmptyStackException constructor

   // one-argument constructor
   public EmptyStackException( String message )
   {
      super( message );
   } // end one-argument EmptyStackException constructor
} // end class EmptyStackException
```

Generic Class StackTest

```java
// Fig. 21.9: StackTest.java
// Stack generic class test program.

public class StackTest
{
   public static void main( String[] args )
   {
      double[] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5 };
      int[] integerElements = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

      // Create a Stack< Double > and a Stack< Integer >
      Stack< Double > doubleStack = new Stack< Double >( 5 );
      Stack< Integer > integerStack = new Stack< Integer >();

      // push elements of doubleElements onto doubleStack
      testPushDouble( doubleStack, doubleElements );
      testPopDouble( doubleStack ); // pop from doubleStack

      // push elements of integerElements onto integerStack
      testPushInteger( integerStack, integerElements );
      testPopInteger( integerStack ); // pop from integerStack
   } // end main

   // test push method with double stack
   private static void testPushDouble(
      Stack< Double > stack, double[] values )
   {
      System.out.println( "\nPushing elements onto doubleStack" );

      // push elements to Stack
      for ( double value : values )
      {
         System.out.printf( "%.1f ", value );
         stack.push( value ); // push onto doubleStack
      } // end for
   } // end method testPushDouble

   // test pop method with double stack
   private static void testPopDouble( Stack< Double > stack )
   {
      // pop elements from stack
      try
      {
         System.out.println( "\nPopping elements from doubleStack" );
         double popValue; // store element removed from stack

         // remove all elements from Stack
         while ( true )
         {
            popValue = stack.pop(); // pop from doubleStack
            System.out.printf( "%.1f ", popValue );
         } // end while
      } // end try
      catch( EmptyStackException emptyStackException )
      {
         System.err.println();
         emptyStackException.printStackTrace();
      } // end catch EmptyStackException
   } // end method testPopDouble

```

Generic Class StackTest

```java
   // test push method with integer stack
   private static void testPushInteger(
      Stack< Integer > stack, int[] values )
   {
      System.out.println( "\nPushing elements onto integerStack" );

      // push elements to Stack
      for ( int value : values )
      {
         System.out.printf( "%d ", value );
         stack.push( value ); // push onto integerStack
      } // end for
   } // end method testPushInteger

   // test pop method with integer stack
   private static void testPopInteger( Stack< Integer > stack )
   {
      // pop elements from stack
      try
      {
         System.out.println( "\nPopping elements from integerStack" );
         int popValue; // store element removed from stack

         // remove all elements from Stack
         while ( true )
         {
            popValue = stack.pop(); // pop from intStack
            System.out.printf( "%d ", popValue );
         } // end while
      } // end try
      catch( EmptyStackException emptyStackException )
      {
         System.err.println();
         emptyStackException.printStackTrace();
      } // end catch EmptyStackException
   } // end method testPopInteger
} // end class StackTest
```

Generic Class (with Generic Methods)

```java
// Fig. 21.10: StackTest2.java
// Passing generic Stack objects to generic methods.
public class StackTest2
{
   public static void main( String[] args )
   {
      Double[] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5 };
      Integer[] integerElements = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

      // Create a Stack< Double > and a Stack< Integer >
      Stack< Double > doubleStack = new Stack< Double >( 5 );
      Stack< Integer > integerStack = new Stack< Integer >();

      // push elements of doubleElements onto doubleStack
      testPush( "doubleStack", doubleStack, doubleElements );
      testPop( "doubleStack", doubleStack ); // pop from doubleStack

      // push elements of integerElements onto integerStack
      testPush( "integerStack", integerStack, integerElements );
      testPop( "integerStack", integerStack ); // pop from integerStack
   } // end main

   // generic method testPush pushes elements onto a Stack
   public static < T > void testPush( String name , Stack< T > stack,
      T[] elements )
   {
      System.out.printf( "\nPushing elements onto %s\n", name );

      // push elements onto Stack
      for ( T element : elements )
      {
         System.out.printf( "%s ", element );
         stack.push( element ); // push element onto stack
      } // end for
   } // end method testPush

```

Notes:

- Methods **testPushDouble** and **testPushInteger** are nearly identical, except for the element type; likewise the methods **testPopDouble** and **testPopInteger**
- They can thus be implemented using generic methods

Generic Class (with Generic Methods)

```java
    // generic method testPop pops elements from a Stack
    public static < T > void testPop( String name, Stack< T > stack )
    {
       // pop elements from stack
       try
       {
          System.out.printf( "\nPopping elements from %s\n", name );
          T popValue; // store element removed from stack

          // remove all elements from Stack
          while ( true )
          {
             popValue = stack.pop();
             System.out.printf( "%s ", popValue );
          } // end while
       } // end try
       catch( EmptyStackException emptyStackException )
       {
          System.out.println();
          emptyStackException.printStackTrace();
       } // end catch EmptyStackException
    } // end method testPop
 } // end class StackTest2
```

Notes:

- Methods **testPushDouble** and **testPushInteger** are nearly identical, except for the element type; likewise the methods **testPopDouble** and **testPopInteger**
- They can thus be implemented using generic methods

Raw Types

- **Raw type**:
  - An instance (**Object**) of a generic class where the type parameter has not been specified

```
Stack objectStack = new Stack( 5 ); // no type-argument specified
```

  - The compiler implicitly uses type **Object** throughout the generic class for each type argument
  - A raw type collection (e.g. **Stack**) can manipulate objects of any type
  - Important for backward compatibility with prior versions of Java (e.g. data structures of **Collections** Framework that previously stored references to **Objects**, now implemented as generic types)
  - It is possible to combine raw types and parameterized types in declarations and assignments:

```
Stack rawTypeStack2 = new Stack< Double >( 5 );
```

  - Raw-type operations are unsafe (possible erroneous assignment) and could lead to exceptions

```
Stack< Integer > integerStack = new Stack( 10 );
```

Raw-Type Stack

```java
// Fig. 21.11: RawTypeTest.java
// Raw type test program.
public class RawTypeTest
{
   public static void main( String[] args )
   {
      Double[] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5 };
      Integer[] integerElements = {  1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

      // Stack of raw types assigned to Stack of raw types variable
      Stack rawTypeStack1 = new Stack( 5 );

      // Stack< Double > assigned to Stack of raw types variable
      Stack rawTypeStack2 = new Stack< Double >( 5 );

      // Stack of raw types assigned to Stack< Integer > variable
      Stack< Integer > integerStack = new Stack( 10 );

      testPush( "rawTypeStack1", rawTypeStack1, doubleElements );
      testPop( "rawTypeStack1", rawTypeStack1 );
      testPush( "rawTypeStack2", rawTypeStack2, doubleElements );
      testPop( "rawTypeStack2", rawTypeStack2 );
      testPush( "integerStack", integerStack, integerElements );
      testPop( "integerStack", integerStack );
   } // end main

   // generic method pushes elements onto stack
   public static < T > void testPush( String name, Stack< T > stack,
      T[] elements )
   {
      System.out.printf( "\nPushing elements onto %s\n", name );

      // push elements onto Stack
      for ( T element : elements )
      {
         System.out.printf( "%s ", element );
         stack.push( element ); // push element onto stack
      } // end for
   } // end method testPush

```

Raw-Type Stack

```
    // generic method testPop pops elements from stack
    public static < T > void testPop( String name, Stack< T > stack )
    {
       // pop elements from stack
       try
       {
          System.out.printf( "\nPopping elements from %s\n", name );
          T popValue; // store element removed from stack

          // remove elements from Stack
          while ( true )
          {
             popValue = stack.pop(); // pop from stack
             System.out.printf( "%s ", popValue );
          } // end while
       } // end try
       catch( EmptyStackException emptyStackException )
       {
          System.out.println();
          emptyStackException.printStackTrace();
       } // end catch EmptyStackException
    } // end method testPop
 } // end class RawTypeTest
```

Wild Cards in Methods that Accept Type Parameters

- **Wildcard**:
  - Enables defining parameterized types that can act as supertypes or subtypes (e.g. **Number** is superclass of **Integer**, but **Array<Number>** is not a supertype of **Array<Integer>**)
  - A wilcard-type argument is denoted by a question mark (**?**), representing a “unknown type”
  - **Program 21.13** defines method **Sum** (lines 23-32) to total numbers (double, integer, etc) in an **ArrayList<Number>**; however, it cannot operate on e.g. **ArrayList<Integer>**
  - **Program 21.14** uses a wildcard-type argument to solve the problem; **ArrayList<? extends Number>** represents an **ArrayList** of any type that subclasses **Number**

Method Sum without Wildcard

```java
// Fig. 21.13: TotalNumbers.java
// Totaling the numbers in an ArrayList<Number>.
import java.util.ArrayList;

public class TotalNumbers
{
   public static void main( String[] args )
   {
      // create, initialize and output ArrayList of Numbers containing
      // both Integers and Doubles, then display total of the elements
      Number[] numbers = { 1, 2.4, 3, 4.1 }; // Integers and Doubles
      ArrayList< Number > numberList = new ArrayList< Number >();

      for ( Number element : numbers )
         numberList.add( element ); // place each number in numberList

      System.out.printf( "numberList contains: %s\n", numberList );
      System.out.printf( "Total of the elements in numberList: %.1f\n",
         sum( numberList ) );
   } // end main

   // calculate total of ArrayList elements
   public static double sum( ArrayList< Number > list )
   {
      double total = 0; // initialize total

      // calculate sum
      for ( Number element : list )
         total += element.doubleValue();

      return total;
   } // end method sum
} // end class TotalNumbers
```

Method Sum with Wildcard

```
// Fig. 21.14: WildcardTest.java
// Wildcard test program.
import java.util.ArrayList;

public class WildcardTest
{
   public static void main( String[] args )
   {
      // create, initialize and output ArrayList of Integers, then
      // display total of the elements
      Integer[] integers = { 1, 2, 3, 4, 5 };
      ArrayList< Integer > integerList = new ArrayList< Integer >();

      // insert elements in integerList
      for ( Integer element : integers )
         integerList.add( element );

      System.out.printf( "integerList contains: %s\n", integerList );
      System.out.printf( "Total of the elements in integerList: %.0f\n\n",
         sum( integerList ) );

      // create, initialize and output ArrayList of Doubles, then
      // display total of the elements
      Double[] doubles = { 1.1, 3.3, 5.5 };
      ArrayList< Double > doubleList = new ArrayList< Double >();

      // insert elements in doubleList
      for ( Double element : doubles )
         doubleList.add( element );

      System.out.printf( "doubleList contains: %s\n", doubleList );
      System.out.printf( "Total of the elements in doubleList: %.1f\n\n",
         sum( doubleList ) );
```

Method Sum with Wildcard

```
      // create, initialize and output ArrayList of Numbers containing
      // both Integers and Doubles, then display total of the elements
      Number[] numbers = { 1, 2.4, 3, 4.1 }; // Integers and Doubles
      ArrayList< Number > numberList = new ArrayList< Number >();

      // insert elements in numberList
      for ( Number element : numbers )
         numberList.add( element );

      System.out.printf( "numberList contains: %s\n", numberList );
      System.out.printf( "Total of the elements in numberList: %.1f\n",
         sum( numberList ) );
   } // end main

   // total the elements; using a wildcard in the ArrayList parameter
   public static double sum( ArrayList< ? extends Number > list )
   {
      double total = 0; // initialize total

      // calculate sum
      for ( Number element : list )
         total += element.doubleValue();

      return total;
   } // end method sum
} // end class WildcardTest
```

Generics and Inheritance

- A generic class can be derived from a nongeneric class (e.g. nongeneric **Object** class is a direct or indirect superclass of every generic class)

- A generic class can be derived from another generic class (e.g. generic class **Stack** is a subclass of generic class **Vector**)

- A nongeneric class can be derived from a generic class (e.g. nongeneric class **Properties** is a subclass of generic class **Hashtable**)

- A generic method in a subclass can override a generic method in a superclass if both methods have the same signature

Number of arguments is the same

Exercise

Write a generic method ***selectionSort*** based on the sort program of **Figs. 19.6 – 19.7** (chapter 19, textbook). Write a test program that generates, sorts and outputs an Integer array and a float array. [Hint: use <T extends Comparable<T>> in the type-parameter section for method ***selectionSort***, so that you can use method compareTo to compare objects of the type that **T** represents].

Generic method to Non-generic method