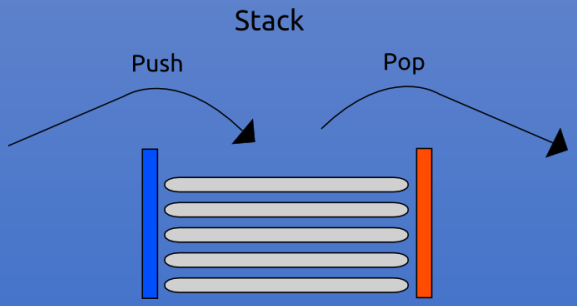

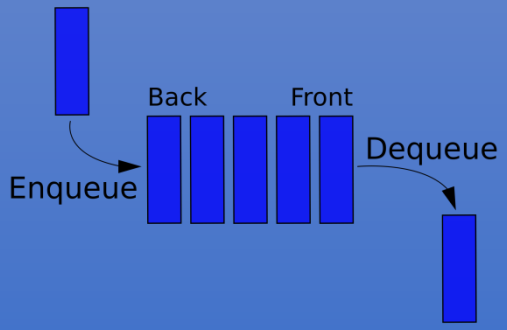

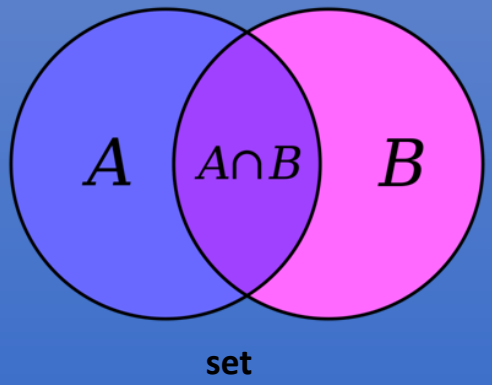

Generics (Classes&Methods)

Lecture 13

Dr. Tamer ABUHMED

Java Programming Course (SWE2023)

College of Computing

SUNG KYUN KWAN UNIVERSITY

Outline

- Generic Classes
- Multiple Type Generic Classes
- Bounds For Generic Type
- Generic Interfaces
- Generic Methods
- Generic Derived Classes

Generics

- A class definition with a type parameter is stored in a file and compiled just like any other class
- Once a parameterized class is compiled, it can be used like any other class
  - However, the class type plugged in for the type parameter must be specified before it can be used in a program
  - Doing this is said to *instantiate* the generic class

```
Sample<String> object =
    new Sample<String>();
```

A Class Definition with a Type Parameter

```
public class Sample<T>
{
    private T data;

    public void setData(T newData)
    {
        data = newData;
    }

    public T getData()
    {
        return data;
    }
}
```

T is a parameter for a type.

Class Definition with a Type Parameter

- A class that is defined with a parameter for a type is called a generic class or a parameterized class
  - The type parameter is included in angular brackets after the class name in the class definition heading
  - Any non-keyword identifier can be used for the type parameter, but by convention, the parameter starts with an uppercase letter
  - The type parameter can be used like other types used in the definition of a class

Tip: Compile with the `-Xlint` Option

- There are many pitfalls that can be encountered when using type parameters
- Compiling with the **`-Xlint`** option will provide more informative diagnostics of any problems or potential problems in the code

```
javac -Xlint Sample.java
```

A Generic Ordered Pair Class (Part 1 of 2)


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

    public Pair()
    {
        first = null;
        second = null;
    }

    public Pair(T firstItem, T secondItem)
    {
        first = firstItem;
        second = secondItem;
    }
```

Constructor headings do not include the type parameter in angular brackets.

(continued)

A Generic Ordered Pair Class (Part 2 of 2)


```
    public void setFirst(T newFirst)
    {
        first = newFirst;
    }

    public void setSecond(T newSecond)
    {
        second = newSecond;
    }

    public T getFirst()
    {
        return first;
    }

public T getSecond()
{
    return second;
}

public String toString()
{
    return ( "first: " + first.toString() + "\n"
             + "second: " + second.toString() );
}
```

```
    public boolean equals(Object otherObject)
    {
        if (otherObject == null)
            return false;
        else if (getClass() != otherObject.getClass())
            return false;
        else
        {
            Pair<T> otherPair = (Pair<T>)otherObject;
            return (first.equals(otherPair.first)
               && second.equals(otherPair.second));
        }
    }
}
```

Using Our Ordered Pair Class


```
import java.util.Scanner;

public class GenericPairDemo
{
   public static void main(String[] args)
   {
        Pair<String> secretPair =
             new Pair<String>("Happy", "Day");

        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter two words:");
        String word1 = keyboard.next();
        String word2 = keyboard.next();
        Pair<String> inputPair =
            new Pair<String>(word1, word2);
        if (inputPair.equals(secretPair))
        {
            System.out.println("You guessed the secret words");
            System.out.println("in the correct order!");
        }
        else
        {
            System.out.println("You guessed incorrectly.");
            System.out.println("You guessed");
            System.out.println(inputPair);
            System.out.println("The secret words are");
            System.out.println(secretPair);
        }
    }
}
```

Output

```
Enter two words:
two words
You guessed incorrectly.
You guessed
first: two
second: words
The secret words are
first: Happy
second: Day
```

A Generic Constructor Name Has No Type Parameter

- Although the class name in a parameterized class definition has a type parameter attached, the type parameter is not used in the heading of the constructor definition

```
public Pair<T>()
```

- A constructor can use the type parameter as the type for a parameter of the constructor, but in this case, the angular brackets are not used

```
public Pair(T first, T second)
```

- However, when a generic class is instantiated, the angular brackets are used

```
Pair<String> pair =
    new Pair<String>("Happy", "Day");
```

A Primitive Type Cannot be Plugged in for a Type Parameter

- The type plugged in for a type parameter must always be a reference type
  - It cannot be a primitive type such as `int`, `double`, or `char`
  - However, now that Java has automatic boxing, this is not a big restriction
  - Note: reference types can include arrays

Pitfall: A Type Parameter Cannot Be Used Everywhere a Type Name Can Be Used

- Within the definition of a parameterized class definition, there are places where an ordinary class name would be allowed, but a type parameter is not allowed
- In particular, the type parameter cannot be used in simple expressions using new to create a new object
  - For instance, the type parameter cannot be used as a constructor name or like a constructor:

```
T object = new T();
T[] a = new T[10];
```

Pitfall: An Instantiation of a Generic Class Cannot be an Array Base Type

- Arrays such as the following are illegal:

```
Pair<String>[] a =
   new Pair<String>[10];
```

  - Although this is a reasonable thing to want to do, it is not allowed given the way that Java implements generic classes

Using Our Ordered Pair Class and Automatic Boxing (Part 1 of 2)


```
import java.util.Scanner;

public class GenericPairDemo2
{
   public static void main(String[] args)
   {
        Pair<Integer> secretPair =
             new Pair<Integer>(42, 24);

        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter two numbers:");
        int n1 = keyboard.nextInt();
        int n2 = keyboard.nextInt();
        Pair<Integer> inputPair =
            new Pair<Integer>(n1, n2);
```

Automatic boxing allows you to use an `int` *argument for an* `Integer` *parameter.*

(continued)

Using Our Ordered Pair Class and Automatic Boxing (Part 2 of 2)


```
        if (inputPair.equals(secretPair))
        {
            System.out.println("You guessed the secret numbers");
            System.out.println("in the correct order!");
        }
        else
        {
            System.out.println("You guessed incorrectly.");
            System.out.println("You guessed");
            System.out.println(inputPair);
            System.out.println("The secret numbers are");
            System.out.println(secretPair);
        }
    }
}
```

Output

```
Enter two numbers:
42 24
You guessed the secret numbers
in the correct order!
```

A Class Definition Can Have More Than One Type Parameter

- A generic class definition can have any number of type parameters
  - Multiple type parameters are listed in angular brackets just as in the single type parameter case, but are separated by commas

Multiple Type Parameters (Part 1 of 2)


```
public class TwoTypePair<T1, T2>
{
    private T1 first;
    private T2 second;

    public TwoTypePair()
    {
        first = null;
        second = null;
    }

    public TwoTypePair(T1 firstItem, T2 secondItem)
    {
        first = firstItem;
        second = secondItem;
    }
```

```
public void setFirst(T1 newFirst)
{
    first = newFirst;
}

public void setSecond(T2 newSecond)
{
    second = newSecond;
}

public T1 getFirst()
{
    return first;
}
```

Multiple Type Parameters (Part 3 of 4)


```
public T2 getSecond()
{
    return second;
}

public String toString()
{
    return ( "first: " + first.toString() + "\n"
             + "second: " + second.toString() );
}
```

```
public boolean equals(Object otherObject)
{
    if (otherObject == null)
        return false;
    else if (getClass() != otherObject.getClass())
        return false;
    else
    {
        TwoTypePair<T1, T2> otherPair =
                    (TwoTypePair<T1, T2>)otherObject;
        return (first.equals(otherPair.first)
            && second.equals(otherPair.second));
    }
}
}
```

The first `equals` *is the* `equals` *of the type* `T1`*. The second* `equals` *is the* `equals` *of the type* `T2`*.*

A Generic Class Cannot Be an Exception Class

- It is not permitted to create a generic class with **`Exception`**, **`Error`**, **`Throwable`**, or any descendent class of **`Throwable`**
  - A generic class cannot be created whose objects are throwable

```
public class Gex<T> extends Exception
```

  - The above example will generate a compiler error message

Using a Generic Class with Two Type Parameters (Part 1 of 2)


```
import java.util.Scanner;

public class TwoTypePairDemo
{
   public static void main(String[] args)
   {
        TwoTypePair<String, Integer> rating =
             new TwoTypePair<String, Integer>("The Car Guys", 8);

        Scanner keyboard = new Scanner(System.in);
        System.out.println(
                    "Our current rating for " + rating.getFirst());
        System.out.println(" is " + rating.getSecond());

        System.out.println("How would you rate them?");
        int score = keyboard.nextInt();
        rating.setSecond(score);
```

(continued)

Using a Generic Class with Two Type Parameters (Part 2 of 2)


```
        System.out.println(
                    "Our new rating for " + rating.getFirst());
        System.out.println(" is " + rating.getSecond());
    }
}
```

Output

```
Our current rating for The Car Guys
is 8
How would you rate them?
10
Our new rating for The Car Guys
is 10
```

Bounds for Type Parameters

- Sometimes it makes sense to restrict the possible types that can be plugged in for a type parameter **`T`**
  - For instance, to ensure that only classes that implement the **`Comparable`** interface are plugged in for **`T`**, define a class as follows:

```
public class RClass<T extends Comparable>
```

  - "**`extends Comparable`**" serves as a *bound* on the type parameter **`T`**
  - Any attempt to plug in a type for **`T`** which does not implement the **`Comparable`** interface will result in a compiler error message

Bounds for Type Parameters

- A bound on a type may be a class name (rather than an interface name)
  - Then only descendent classes of the bounding class may be plugged in for the type parameters

```
public class ExClass<T extends Class1>
```

- A bounds expression may contain multiple interfaces and up to one class
- If there is more than one type parameter, the syntax is as follows:

```
public class Two<T1 extends Class1, T2 extends
  Class2 & Comparable>
```

A Bounded Type Parameter


```
public class Pair<T extends Comparable>
{
    private T first;
    private T second;

    public T max()
    {
        if (first.compareTo(second) <= 0)
            return first;
        else
            return second;
    }
```

<All the constructors and methods given in Display 14.5
are also included as part of this generic class definition>

```
}
```

Generic Interfaces

- An interface can have one or more type parameters
- The details and notation are the same as they are for classes with type parameters

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

Generic Methods

- When a generic class is defined, the type parameter can be used in the definitions of the methods for that generic class
- In addition, a generic method can be defined that has its own type parameter that is not the type parameter of any class
  - A generic method can be a member of an ordinary class or a member of a generic class that has some other type parameter
  - The type parameter of a generic method is local to that method, not to the class

Generic Methods

- The type parameter must be placed (in angular brackets) after all the modifiers, and before the returned type

```
public static <T> T genMethod(T[] a)
```

- When one of these generic methods is invoked, the method name is prefaced with the type to be plugged in, enclosed in angular brackets

```
String s = NonG.<String>genMethod(c);
```

Inheritance with Generic Classes

- A generic class can be defined as a derived class of an ordinary class or of another generic class
  - As in ordinary classes, an object of the subclass type would also be of the superclass type
- Given two classes: **`A`** and **`B`**, and given **`G`**: a generic class, there is no relationship between **`G<A>`** and **`G<B>`**
  - This is true regardless of the relationship between class **`A`** and **`B`**, e.g., if class **`B`** is a subclass of class **`A`**

A Derived Generic Class (Part 1 of 2)


```
public class UnorderedPair<T> extends Pair<T>
{
    public UnorderedPair()
    {
        setFirst(null);
        setSecond(null);
    }

    public UnorderedPair(T firstItem, T secondItem)
    {
        setFirst(firstItem);
        setSecond(secondItem);
    }
```

(continued)

A Derived Generic Class (Part 2 of 2)


```
    public boolean equals(Object otherObject)
    {
        if (otherObject == null)
            return false;
        else if (getClass() != otherObject.getClass())
            return false;
        else
        {
            UnorderedPair<T> otherPair =
                            (UnorderedPair<T>)otherObject;
            return (getFirst().equals(otherPair.getFirst())
               && getSecond().equals(otherPair.getSecond()))
               ||
                   (getFirst().equals(otherPair.getSecond())
               && getSecond().equals(otherPair.getFirst()));
        }
    }
}
```

Using UnorderedPair


```
public class UnorderedPairDemo
{
   public static void main(String[] args)
   {
        UnorderedPair<String> p1 =
             new UnorderedPair<String>("peanuts", "beer");
        UnorderedPair<String> p2 =
             new UnorderedPair<String>("beer", "peanuts");

        if (p1.equals(p2))
        {
            System.out.println(p1.getFirst() + " and " +
                      p1.getSecond() + " is the same as");
            System.out.println(p2.getFirst() + " and "
                               + p2.getSecond());
        }
    }
}
```

Output

```
peanuts and beer is the same as
beer and peanuts
```

Summary

- Generic Classes
- Multiple Type Generic Classes
- Bounds For Generic Type
- Generic Interfaces
- Generic Methods
- Generic Derived Classes