Contents **Topic 07 - Generic Programming**

I Generic Programming - Introduction

Generic Programming

- **Generics**
  – method and class definitions which involve type parameters.
- **Generic Programming**
  – writing code that can be reused for objects of many different types.
- User-defined generic classes and methods
  [See examples 1 and 2]
- There are also generic classes and methods provided in the standard Java libraries:
  e.g. the **ArrayList** generic class,
  the **Collections.sort** generic method
  [See example 3]

Example 1 Simple Generic Method

Declare a Generic Method

Use the Generic Method

```
public class Main
{
   public static <T> void printTwice(T x)
   {
      System.out.println(x);
      System.out.println(x);
   }

   public static void main(String[] args)
   {
      printTwice("hello");    //This time T is a string
      printTwice(1234);       //This time T is an integer
      printTwice(4.0/3);      //This time T is a double
   }
}
```

x is called the *value parameter*.
T is called the *type parameter*.
<T> means that:
In the following, T is the type parameter which stands for the actual type which is known when printTwice is called.

Output:

```
hello
hello
1234
1234
1.3333333333333333
1.3333333333333333
```

Example 2 Simple Generic Class

<T> is called the type parameter.
<T> means that:
In the following, T is the type parameter which stands for the actual type which is known when an object of this Generic is created.

Declare a Generic Class

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

      public Pair() { } //first and second automatically initialized as null
      public Pair(T x1, T x2) { first = x1; second = x2; }

      @Override
      public String toString() {return "(1)"+first+" (2)"+second;}
}
```

Use the Generic Class

```
public class Main
{
      public static void main(String[] args)
      {
            Pair<String> p0 = new Pair<String>();
            Pair<String> p1 = new Pair<String>("hello","cheers");
            Pair<Boolean> p2 = new Pair<Boolean>(true,false);
            Pair<Integer> p3 = new Pair<Integer>(123,456);
            Pair<Number> p4 = new Pair<Number>(123,456);
            Pair<Object> p5 = new Pair<Object>(123,"cheers");
            System.out.println(p0);
            System.out.println(p1);
            System.out.println(p2);
            System.out.println(p3);
            System.out.println(p4);
            System.out.println(p5);
      }
}
```

Each time when we use the Generic Class `Pair`, we need to tell the type which T stands for.

Output:

```
(1)null (2)null
(1)hello (2)cheers
(1)true (2)false
(1)123 (2)456
(1)123 (2)456
(1)123 (2)cheers
```

Example 3 The **ArrayList** generic class, the **Collections.sort** generic method [provided in standard Java libraries]

```
import java.util.ArrayList;
import java.util.Collections;

public class Main
{
    public static void main(String[] args)
    {
        ArrayList<Integer> arrlist = new ArrayList<>();
        arrlist.add(1234);
        arrlist.add(8899);
        arrlist.add(36);
        Collections.sort(arrlist);
        System.out.println(arrlist); //Output: [36, 1234, 8899]
    }
}
```

[For interested students only]

The ArrayList class is defined as: **class ArrayList<E>**

The sort method is defined as: **public static <T extends Comparable<? super T>> void sort(List<T> list)**

Where **List** is a Java interface implemented by **ArrayList** and a number of other Java classes.

[http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html]
[http://stackoverflow.com/questions/4343202/difference-between-super-t-and-extends-t-in-java]

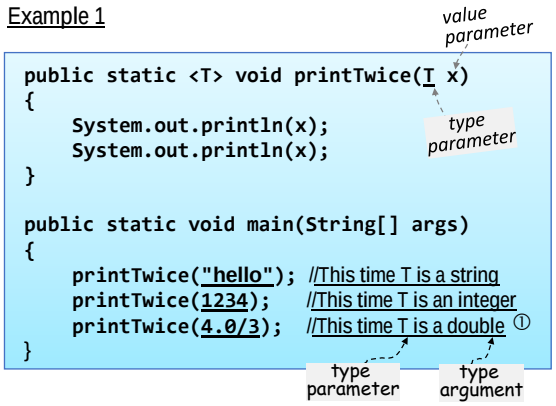
II Type Inference and Diamond Syntax

Type Inference

- *Type inference* is a Java compiler's ability to determine the type argument(s) that make the invocation applicable.
- Compiler looks at each method invocation① and corresponding declaration②, in order to decide the type.

Ref: http://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html

Example 1



Example 2

```
class Pair<T>
{
    private T first;
    private T second;
    ..
}
```

```
public static void main(String[] args)
{
    Pair<String> p0 = new Pair<String>();
    Pair<String> p1 = new Pair<String>("..","..");
    Pair<Boolean> p2 = new Pair<Boolean>(true,false);
                       ②
}
```

This time T is a boolean

The Diamond Syntax: <>

- We can omit the types in <> when **new** is used. (Since Java 7)
- i.e., simply write

```
Pair<String> p0 = new Pair<> ();
Pair<String> p1 = new Pair<> ("hello","cheers");
Pair<Boolean> p2 = new Pair<>(true,false);
```

Compiler checks the type of the object variable (here p0, p1, p2) to guess and fill in the type parameter.

III Pitfalls and Type Erasure

- Static fields belong to the generic class, not the instantiated classes.

```
class Smartphone {}
class Pager {}
class TabletPC {}
class MobileDevice<T> {
    private static int count=0;
    MobileDevice() {count++;System.out.println(count);}
    // ...
}
public class Main
{
    public static void main(String[] args)
    {
            MobileDevice<Smartphone> phone = new MobileDevice<>();
            MobileDevice<Pager> pager = new MobileDevice<>();
            MobileDevice<TabletPC> pc = new MobileDevice<>();
    }
}
```

Output:
1
2
3

- Cannot declare static fields of a type parameter

```
public class MobileDevice<T> {
    private static T os; // compile-time error

    // ...
}
```

```
MobileDevice<Smartphone> phone = new MobileDevice<>();
MobileDevice<Pager> pager = new MobileDevice<>();
MobileDevice<TabletPC> pc = new MobileDevice<>();
```

What should be the type of os?

[http://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createObjects]

- Cannot create an object instance of a type parameter

Compile error:

```
public static <E> void append(List<E> list) {
    E elem = new E();  // compile-time error
    list.add(elem);
}
```

- A class cannot have two overloaded methods that will have the same signature after *type erasure*.

```
public class X{
    public void print(Pair<String> strSet) { } //compile-time error
    public void print(Pair<Integer> intSet) { } //compile-time error
}
```

Compile-time error:
Method print(Pair<String>) has the same erasure print(Pair<T>) as another method in type X

For *Type Erasure* – We may think of it in this way:

> When the compiler generates the bytecode, type parameters in generic types are "replaced with the raw type* : *Object*".
>
> Advantage: *Type Erasure* ensures that no new classes are created for parameterized types; consequently generics incur no runtime overhead. (c.f. Different approach in C++ *templates*)

* A type parameter could be *bounded. If so, it is replaced by the bound instead of Object.*

Type Erasure [For interested students only]

http://docs.oracle.com/javase/tutorial/java/generics/erasure.html
http://docs.oracle.com/javase/tutorial/java/generics/genTypes.html

Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming. To implement generics, the Java compiler applies **type erasure** to:

- Replace all type parameters in generic types with ***raw types***, which are their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
- Insert type casts if necessary to preserve type safety.
- Generate bridge methods to preserve polymorphism in extended generic types.

Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur **no runtime overhead**.

The raw type for `Pair<T>` in Example 2 looks like:

```
class Pair
{
   private Object first;
   private Object second;
   public Pair() { }

   public Pair(Object first, Object second)
   {
      this.first = first;
      this.second = second;
   }

   @Override
   public String toString() {
       return "(1)"+first+" (2)"+second;
   }
}
```

```
public static void main(String[] args)
{
      Pair p0 = new Pair();
      Pair p1 = new Pair("hello","cheers");
      Pair p5 = new Pair(123,"cheers");
      System.out.println(p0); //(1)null (2)null
      System.out.println(p1); //(1)hello (2)cheers
      System.out.println(p5); //(1)123 (2)cheers
}
```

Pitfalls - More [For interested students only]

```
class Pair<T>
{
      private T first;
      private T second;
      public Pair() { }
      public Pair(T x1, T x2) { first = x1; second = x2; }
```

- Constructor headings do not include the type parameter

  We do not write

```
☒ public <T> Pair() { }
☒ public <T> Pair(T x1, T x2) { first = x1; second = x2; }
```

- Cannot use a generic class as the base type of an array

```
String [] x1=new String[3]; //OK
Pair<String>[] x2 = new Pair<String>[3]; //Error:
```

Cannot create a generic array of Pair<String>

- Cannot create, catch, or throw objects of parameterized types

```
class MathException<T> extends Exception { /* ... */ }    // compile-time error

class QueueFullException<T> extends Throwable { /* ... */} // compile-time error
```

- Cannot use casts or instanceof with parameterized types

```
if (list instanceof ArrayList<Integer>) // compile-time error
```

```
Pair<Integer> li = new Pair<>();
Pair<Number>  ln = (Pair<Number>) li; // compile-time error
```

But we can use *bounded wildcard* *:
Pair<? **extends Number>** **ln = li;**

IV Generic Class with more than 1 parameter [For interested students only]

Generic Class can have more than 1 type parameters

Example 2a

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

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

```
public static void main(String[] args)
{
   TwoTypePair<String, Integer> x1,x2,x3;
   x1 = new TwoTypePair<>("USD",123);
   x2 = new TwoTypePair<>("USD",456);
   x3 = new TwoTypePair<>("USD",123);

   System.out.println(x1.equals(x2));//false
   System.out.println(x1.equals(x3));//true
}
```

V Other Notes - Bounds for type parameter, Generic Interface, Inheritance with Generic Class

Bounds for Type Parameters

- To restrict the possible types that can be plugged in for a type parameter T
- Example:

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

Generic Interfaces (Similar to generic classes)

- Example:

```
interface PairInterface<T1, T2>
{
      void processThePair(T1 t1, T2 t2);
}
```

Inheritance with Generic Classes

- A generic class can be defined as a derived class of an ordinary class or of another generic class
- Example:

```
class C0 {}
class C1<T> extends C0  {}
class C2<T> extends C1<T>  {}
class C3<T> extends C1<String> {}
```