

Session 10

Generics

(<http://docs.oracle.com/javase/tutorial/java/generics/index.html>)

Generic class & Generic method

```

15
16 public class ClockList extends ArrayList implements I_List {
17
18     Scanner sc = new Scanner(System.in);
19
20     @Override
21     public int find(String code) {
22         int index = -1;
23         for (int i = 0; i < this.size(); i++) {
24             Clock clock = (Clock) this.get(i);
25             if (clock.getCode().equals(code)) {
26                 index = i;
27                 break;
28             }
29         }
30         return index;
31     }
32

```

```

15
16 public class ClockList extends ArrayList<Clock> implements I_List {
17
18     Scanner sc = new Scanner(System.in);
19
20     @Override
21     public int find(String code) {
22         int index = -1;
23         for (int i = 0; i < this.size(); i++) {
24             Clock clock = this.get(i);
25             if (clock.getCode().equals(code)) {
26                 index = i;
27                 break;
28             }
29         }
30         return index;
31     }
32

```

```
public static void main(String[] args) {
    // khai bao list chỉ được chứa phần tử kiểu Integer

    ArrayList<Integer> list = new ArrayList<>();

    list.add(123);
    list.add(true); // error
    list.add("String"); // error
    list.add(456);

    list.add(new String("123")); // các kiểu khác đều sẽ không hợp lệ
}
```

What is Generics?

- A technique allows programmers **creating general processes** on data whose data types are not determined (generic is not used) or they **can be determined (generic is used) when they are used**.
- A way allows programmer **implementing general algorithms** which can be used to process multi-type input → Polymorphism.

What is Generics?

ArrayList<T> list

GENERIC

ArrayList<String> list

ArrayList<Integer> list

ArrayList<Double> list

ArrayList<Character> list

Objectives

- How we can create a list of arbitrary elements?
- Generics in Java API (java.util package)
- Advantages of Generics
- How to create a generic class/ method/ interface
- How is a generic class treated by compiler?
- How to give bounded type parameters?
- Restrictions on Generics

A list of arbitrary elements

- Reference type conformity:
fatherRef=sonRef
 - The Object class is the ultimate class of all Java class
- We can create a list of elements which can belong to different classes
- A demonstration:

```
class Point {
    int x, y;
    Point(int x, int y) { this.x=x; this.y=y; }
    public String toString() { ... }
}

public class NonGenericDemo {
    Object[] ar = new Object[100];
    int n=0;
    void add(Object obj){ ar[n++]=obj; }
    void print(){
        for (int i=0; i<n;i++) System.out.println(ar[i]);
    }
    public static void main(String[] args){
        NonGenericDemo obj= new NonGenericDemo();
        obj.add(new String("Hello"));
        obj.add(5);
        obj.add(new Point(9,3));
        obj.print();
    }
}
```

```
run:
Hello
5
[9,3]
```

Generic Classes in java.util

- Almost of interfaces and classes related to lists in the Java API declared as generic.
 - Type Parameter Naming Conventions
 - By convention, type parameter names are single, uppercase letters.
 - The most commonly used type parameter names are:
 - E : Element/ K: Key
 - N – Number/ T - Type
 - V - Value
 - S,U,V etc. - 2nd, 3rd, 4th types
- java.lang.[Object](#)
 - java.util.[AbstractCollection](#)<E>
 - java.util.[AbstractList](#)<E>
 - java.util.[AbstractSequentialList](#)<E>
 - java.util.[LinkedList](#)<E>
 - java.util.[ArrayList](#)<E>
 - java.util.[Vector](#)<E>
 - java.util.[Stack](#)<E>
 - java.util.[AbstractQueue](#)<E>
 - java.util.[PriorityQueue](#)<E>
 - java.util.[AbstractSet](#)<E>
 - java.util.[EnumSet](#)<E>
 - java.util.[HashSet](#)<E>
 - java.util.[LinkedHashSet](#)<E>
 - java.util.[TreeSet](#)<E>
 - java.util.[AbstractMap](#)<K,V>
 - java.util.[EnumMap](#)<K,V>
 - java.util.[HashMap](#)<K,V>
 - java.util.[LinkedHashMap](#)<K,V>
 - java.util.[IdentityHashMap](#)<K,V>
 - java.util.[TreeMap](#)<K,V>
 - java.util.[WeakHashMap](#)<K,V>

Generics on a List

- Sometimes, we want to create a list with **restrictions as elements must belong to some types** → Generic
- Generic is a technique which allows a list of arbitrary objects and supports advantages if elements of a list belong to the same data type.

Advantages of Generics

- Generics add stability to your code by making more of your bugs detectable at compile time.
- Generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods and **limits on parametric types may be declared**.
- Code that uses generics has many benefits over non-generic code.
 - Stronger type checks at compile time
 - Elimination of casts.
 - Enabling programmers to implement generic algorithms.

Generics are not used

- The package `java.util` supports general-purpose implementations which allows lists containing arbitrary elements
- The **cost** of this flexibility is we may have to use a **casting operator** when accessing an element.

```
Generic1.java *
1 import java.util.Vector;
2 class Person {
3     String name; int age;
4     Person(String n, int a)
5     { name=n; age=a; }
6     void print ()
7     { System.out.println( name + ", " + age); }
8 }
9 public class Generic1 {
10     public static void main(String[] args) {
11         Vector v = new Vector();
12         v.add (new Person("Hoa", 23));
13         v.add (new Person("Tuấn", 27));
14         for (int i= v.size()-1; i>=0; i--)
15             > ({Person} (v.get(i))).print();
16     }
17 }
18
```

The class **Object** does not have the **print()** method

Output - Chapter08 (run)

```
run:
Tuấn, 27
Hoa, 23
BUILD SUCCESSFUL (total time: 0 seconds)
```

Generics are used

- If all elements of the collection are homogeneous (identical), the generic technique should be used.
- Generics add stability to your code by making more of your bugs detectable at compile time. Casting can not be used.

```

1 import java.util.Vector;
2 class Person2 {
3     String name; int age;
4     Person2(String n, int a)
5     { name=n; age=a; }
6     void print ()
7     { System.out.println( name + ", " + age); }
8 }
9 public class Generic2 {
10     public static void main(String[] args) {
11         Vector<Person2> v = new Vector<Person2> ();
12         v.add (new Person2 ("Hoa", 23));
13         v.add (new Person2 ("Tuấn", 27));
14         for (int i= v.size()-1; i>=0; i--)
15             v.get(i).print();
16     }
17 }
18

```

The casting operators are missed.

Output - Chapter08 (run)

```

run:
Tuấn, 27
Hoa, 23
BUILD SUCCESSFUL (total time: 1 second)

```

Using Generics- Syntax

- **Invoking and Instantiating a Generic Type**
 - *Box<Integer> integerBox = new Box<Integer>();*
- **The Diamond**
 - *Box<Integer> integerBox = new Box<>();*
- **Multiple Type Parameters**
 - *Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);*
- **Parameterized Types**
 - *OrderedPair<String, **Box<Integer>**> p = new OrderedPair<>("primes", new Box<Integer>(...));*

Implementing a Generic class

- Syntax:

```
class name<T1, T2, ..., Tn> {  
    code  
}
```

```
public class Box<T> {  
    // T stands for "Type"  
    private T t;  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}
```

Implementing Generic Methods

```

[-] /* Generic class for processing arrays */
[-] import java.util.Arrays;
    public class GenericArray <T> {
[-]     public static <T> T get( int i, T[] ar){
        return ar[i];
    }
[-]     public static <T> void output(T[] ar){
        for (T x: ar) System.out.print(x + ", ");
        System.out.println();
    }
[-]     public static <T> void sort(T[] ar){
        Arrays.sort(ar);
    }
    }

```

Implementing Generic Methods...

```
class GenericArrayUse {
    public static void main(String[] args){
        Integer a[]={1,2,3,4,5};
        GenericArray obj1= new GenericArray();
        obj1.output(a);
        System.out.println(GenericArray.get(3,a));
        Double b[]={1.1, 2.2, 3.3, 4.4};
        GenericArray<Double> obj2= new GenericArray<Double>();
        obj2.output(b);
        String list[]={ "you", "love", "I" };
        GenericArray<String> obj3= new GenericArray<String>();
        obj3.output(list);
        obj3.sort(list);
        obj3.output(list);
    }
}
```

Generic is not used

Generic is used

run:

1, 2, 3, 4, 5,
4

1.1, 2.2, 3.3, 4.4,
you, love, I,
I, love, you,

How generic class is treated?

- Compiler will save generic information in this class to class files (file.class)
- When this class is used (an object of this class is created)
 - If an argument types are declared: Compiler updates type information.
 - If no argument type is declared, type information in parameters are erased or changed to Object

Implementing a Generic Methods

- *Generic methods* are methods that introduce their own type parameters.
- The type parameter's scope is limited to the method where it is declared.
- The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type.

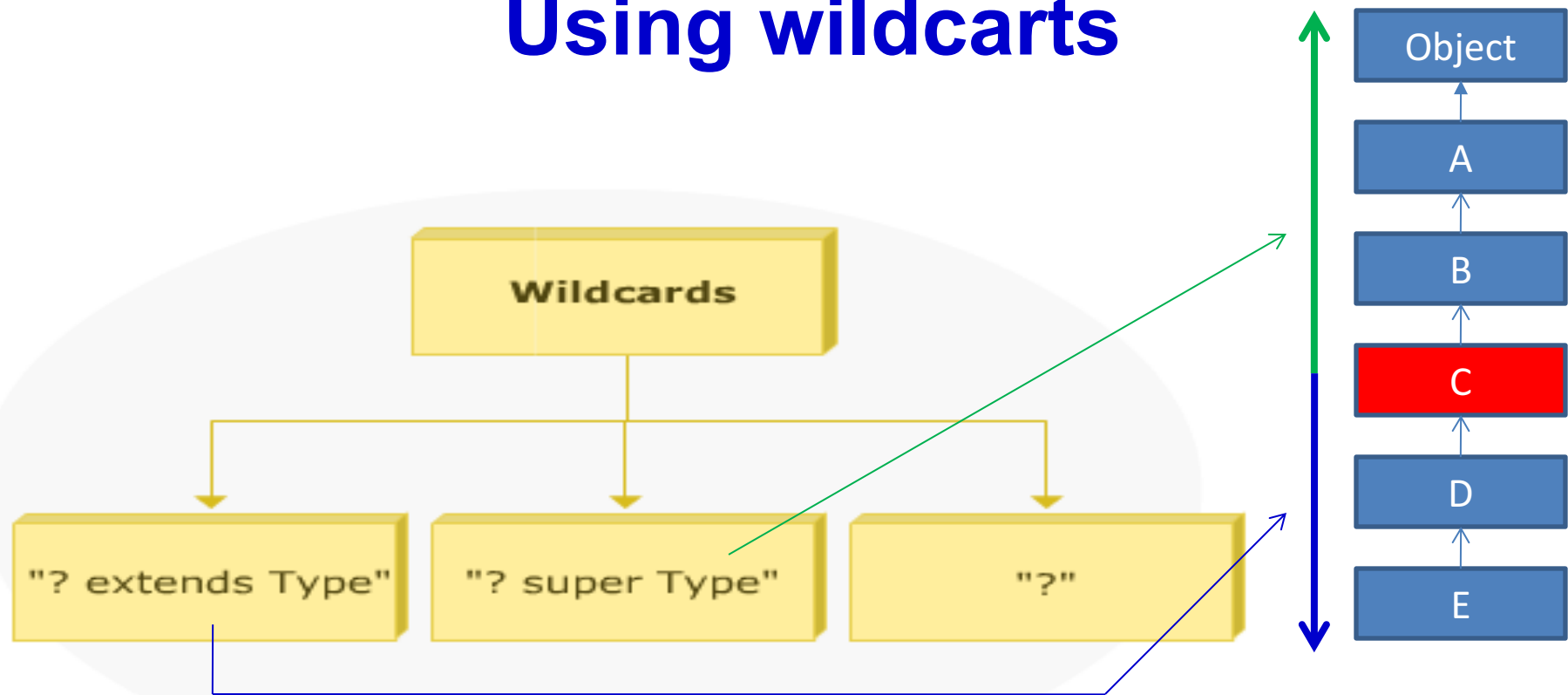
```
public static <K, V> boolean equals(Pair<K, V> p1, Pair<K, V>
p2) {
    return p1.getKey().equals(p2.getKey()) &&
        p1.getValue().equals(p2.getValue());
}
```

Bounded Type Parameters

- Restriction on types of arguments when a method is called.

```
public class GenericExtendDemo <T> {  
    private T t;  
    public void set(T t) { this.t = t; }  
    public T get() { return t;}  
    public <U extends Number> void inspect(U u){  
        System.out.println("T: " + t.getClass().getName());  
        System.out.println("U: " + u.getClass().getName());  
    }  
    public static void main(String[] args) {  
        GenericExtendDemo<Integer> integerBox = new GenericExtendDemo<Integer>();  
        integerBox.set(new Integer(10));  
        integerBox.inspect("some text"); //error:cannot be applied to given type  
    }  
}
```

Bounded Type Parameters: Using wildcards



The ? stands for an unknown type

? **extends** Type : a *bounded wildcard*. Type is upper bound

? **super** Type : a *bounded wildcard*. Type is lower bound

Wildcards

- The question mark (?), called the *wildcard*, represents an unknown type.
- The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable.
- The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a super type.

Wildcards Demo.


```
/* WildCard_Demo.java */
import java.util.*;
class A
{
    int a=3;
    public String toString() { return "" + a; }
}
class B extends A
{
    int b=5;
    public String toString() { return "" + (a+b); }
}
class C
{
    int c= 10;
    public String toString() { return "" + c; }
}
```

```
public class WildCard_Demo {
    /** Creates a new instance of WildCard_Demo */
    public WildCard_Demo() {
    }
    static public void print1 (Collection<?> col)
    {
        for (Object o:col) System.out.print(o + " ,");
    }
    static public void print2 (Collection<? extends A> col)
    {
        for (Object o:col) System.out.print(o+ " ,");
    }
    static public void print3 (Collection<? super B> col)
    {
        for (Object o:col) System.out.print(o+ " ,");
    }
    public static void main(String args[])
    {
        Vector VA= new Vector();
        VA.add(new A());VA.add(new A());VA.add(new A());
        Vector VB= new Vector();
        VB.add(new B());VB.add(new B());VB.add(new B());
        Vector VC= new Vector();
        VC.add(new C());VC.add(new C());VC.add(new C());
        WildCard_Demo.print1(VC); System.out.println();
        WildCard_Demo.print2(VB);System.out.println();
        WildCard_Demo.print3(VA);System.out.println();
        WildCard_Demo.print2(VC);
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
E:\GiangDay\Aptech-Materials\ACCP-2007-Sem2\Java-Advanced\classes>java WildCard_Demo
10 ,10 ,10 ,
8 ,8 ,8 ,
3 ,3 ,3 ,
Exception in thread "main" java.lang.ClassCastException: C cannot be cast to A
    at WildCard_Demo.print2(WildCard_Demo.java:23)
    at WildCard_Demo.main(WildCard_Demo.java:38)
E:\GiangDay\Aptech-Materials\ACCP-2007-Sem2\Java-Advanced\Assignments\Ass6\build\classes>pause
Press any key to continue . . .
```

Raw Types

- When a generic type like collection is used without a type parameter, it is called a raw type. Compiler will execute **erasure process** to remove all generic type information. All the type information between angle brackets are thrown out.



```
public class Box<T> {
    public void set(T t) { /* ... */ }
    // ...
}
//...
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;
Box rawBox = new Box();      // rawBox is a raw type of Box<T>
Box<Integer> intBox = rawBox; // warning: unchecked conversion
```

Type Erasure

- Type erasure will
 - Replace all **bounded type parameters** in generic types with their bounds


```
public <U extends A> void inspect (U obj)
```

→

```
public void inspect (A obj)
```
 - Change unbounded type parameter to Object


```
public void f (T obj)
```

→

```
public void f (Object obj)
```
 - Insert type casts if necessary to preserve type safety.
 - Generate bridge methods to preserve polymorphism in extended generic types.

→ The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.

Type Erasure – Demo.

- Erasure of Generic Method

Type information is erased.

```
public static <T> int count(T[] anArray, T elem) {
    int cnt = 0;
    for (T e : anArray)
        if (e.equals(elem))
            ++cnt;
    return cnt;
}

//T is unbounded, the Java compiler replaces it with Object
public static int count(Object[] anArray, Object elem) {
    int cnt = 0;
    for (Object e : anArray)
        if (e.equals(elem))
            ++cnt;
    return cnt;
}
```

Restrictions on Generics

- Cannot Instantiate Generic Types with Primitive Types.
- Cannot Create Instances of Type Parameters.
- Cannot Declare Static Fields Whose Types are Type Parameters.
- Cannot Use Casts or instanceof With Parameterized Types.
- Cannot Create Arrays of Parameterized Types.
- Cannot Create, Catch, or Throw Objects of Parameterized Types.
- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type.

Summary

- Generics on methods, classes and collections
- Bounded Type Parameters
- Working with Wildcards
- Working with type erasure
- Generic restrictions