Generics

Eleni Mangina

Room B2.05
School of Computer Science and Informatics
University College Dublin, Ireland

Interfaces…

- Have a look first at the following code, based on what you have learnt about Interfaces (Chapter 2):
    - Person.java
    - Student.java
    - PersonPairDirectory.java

Focus…

- **Generics framework:** Java 5.0 included generics framework for using abstract types in a way that avoids explicit casts
- **Generic type:** is a type that is not defined at compilation time, but becomes fully specified at run time.
- The generic framework allows us to define a class in terms of a set of **formal type parameters**, which could be used, for example, to abstract the types of some internal variables of the class.
- **Angle brackets < >** are used to enclose the list of formal type parameters
- Although any valid identifier can be used for a formal type parameter, **single-letter uppercase names** are conventionally used

- Given a class that has been defined with such parameterized types, we instantiate an object of this class by using **actual type parameters** to indicate the concrete types used.
- See example code.. A class Pair storing key-value pairs, where the types of the key and value are specified by parameters K and V, respectively. The main method creates two instances of this class, one for a String-Integer pair and the other for a Student-Double
  - Pair.java
  - Student.java

- In these examples, the actual type parameter can be an arbitrary type. To restrict the type of an actual parameter, we can use an extends clause, as shown in the PersonPairDirectory.java, which is defined in terms of a generic type parameter P, partially specified by stating that it extends class Person
- See code
  - PersonPairDirectoryGeneric.java

- Compare PersonPairDirectoryGeneric.java with PersonPairDirectory.java
- We can declare a variable referring to an instance of PersonPairDirectoryGeneric, that stores pairs of objects of type Student:

```
PersonPairDirectoryGeneric<Student> myStudentDirectory
```

- For such an instance, method findOther returns a value of type Student. Thus, the following statement, which does not use a cast, is correct:

```
Student cute_one = myStudentDirectory.findOther(smart_one)
```

Generic versions of methods

- We can include the generic definition among the method modifiers. For example, this is the definition of a method that can compare the keys from any two Pair objects, provided that their keys implement the Comparable interface:

```
Public static <K extends Comparable V,L,M> int
    comparablePairs(Pair<K,V> p, Pair<L,W> q){
        return p.getKey(),compareTo(q.getKey());
                                        //p's key implements compareTo
}
```

Important!

- In Generic Types, the elements stored in an array cannot be a type variable or a parameterized type. Java allows for an array to be defined with a parameterized type, but **it does not allow a parameterized type to be used to create a new array**. Fortunately it allows for an array defined with a parameterized type to be initialised with a newly created, nonparametric array. Even so, this latter mechanism causes java compiler to issue a warning, because it is not 100% type safe:

```
public static void main (String[] args) {
    Pair<String,Integer> [] a = new Pair[10] //right, but
    gives a warning
    Pair<String,Integer> [] b = new Pair<String,
    Integer>[10] //wrong!
    a[0] = new Pair<String,Integer>(); //this is right!
    a[0].set(“Dog”,10); //right!
    System.out.println(“First pair is “+a[0].getKey()+”, ”
    +a[0].getValue());
}
```

Back to Stacks… Stack Interface

```
/** * Interface for a stack: a collection of objects that
*are inserted and removed according to the last-in first-
*out principle. This interface includes the main methods
*of java.util.Stack.
* @author Roberto Tamassia
@author Michael Goodrich
@see EmptyStackException */
public interface Stack<E> {
/** * Return the number of elements in the stack.
@return number of elements in the stack. */
    public int size();
    /** * Return whether the stack is empty.
    * @return true if the stack is empty, false otherwise. */
    public boolean isEmpty();
    /** * Inspect the element at the top of the stack.
    * @return top element in the stack.
    * @exception EmptyStackException if the stack is empty. */
    public E top() throws EmptyStackException;
    /** * Insert an element at the top of the stack.
    * @param element to be inserted. */
    public void push (E element);
    /** * Remove the top element from the stack.
    * @return element removed.
    * @exception EmptyStackException if the stack is empty. */
    public E pop() throws EmptyStackException; }
```

Array based Implementation

```
/** * Implementation of the stack ADT using a fixed-length array. An
*exception is thrown if a push operation is attempted when the size
*of the stack is equal to the length of the array. This class
*includes the main methods of the built-in class java.util.Stack. */
public class ArrayStack<E> implements Stack<E> {
        protected int capacity; // The actual capacity of the stack array
        public static final int CAPACITY = 1000; // default array capacity
        protected E S[]; // Generic array used to implement the stack
        protected int top = -1; // index for the top of the stack
        public ArrayStack() { this(CAPACITY); // default capacity }
        public ArrayStack(int cap) {
                capacity = cap; S = (E[]) new Object[capacity];
                // compiler may give warning, but this is ok }
        public int size() { return (top + 1); }
        public boolean isEmpty() { return (top < 0); }
        public void push(E element) throws FullStackException {
                if (size() == capacity) throw new
                 FullStackException("Stack is full."); S[++top] = element; }
        public E top() throws EmptyStackException {
                if (isEmpty()) throw new EmptyStackException
                ("Stack is empty."); return S[top]; }
        public E pop() throws EmptyStackException {
                E element; if (isEmpty()) throw new
                EmptyStackException("Stack is empty.");
                element = S[top]; S[top--] = null;
                 // dereference S[top] for garbage collection. return element;
        }
```

```
public String toString() {
                String s; s = "["; if (size() > 0) s+= S[0];
                if (size() > 1) for (int i = 1; i <= size()-1; i++) { s += ", " + S[i]; }
                 return s + "]"; }
// Prints status information about a recent operation and the stack.
 public void status(String op, Object element) {
                System.out.print("------> " + op); // print this operation
                System.out.println(", returns " + element);
                // what was returned
                System.out.print("result: size = " + size() + ",
                                isEmpty = " + isEmpty());
                System.out.println(", stack: " + this);
                 // contents of the stack }
 /** * Test our program by performing a series of operations on stacks,
 * printing the operations performed, the returned elements and the
* contents of the stack involved, after each operation. */
public static void main(String[] args) {
Object o; ArrayStack<Integer> A = new ArrayStack<Integer>();
A.status("new ArrayStack<Integer> A", null);
A.push(7);  A.status("A.push(7)", null); o = A.pop();
A.status("A.pop()", o);  A.push(9);
A.status("A.push(9)", null); o = A.pop(); A.status("A.pop()", o);
ArrayStack<String> B = new ArrayStack<String>();
B.status("new ArrayStack<String> B", null); B.push("Bob");
B.status("B.push(\"Bob\")", null); B.push("Alice");
B.status("B.push(\"Alice\")", null); o = B.pop();
B.status("B.pop()", o); B.push("Eve");
B.status("B.push(\"Eve\")", null); } }
```

Example output? Let's run the code