Generics

Code listing 1: multiple type parameters

// Code from https://docs.oracle.com/javase/tutorial/java/generics/types.html

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class MyPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public MyPair(K key, V value) {
      this.key = key;
      this.value = value;
    }

    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

Two instantiations of the MyPair class can be created as follows:

```
Pair<String, Integer> p1 = new MyPair<String, Integer>("Even", 8);

Pair<String, String>  p2 = new MyPair<String, String>("hello", "world");
```

or by using the diamond notation:

```
MyPair<String, Integer> p1 = new MyPair<>("Even", 8);

MyPair<String, String>  p2 = new MyPair<>("hello", "world");
```

or also:

```
MyPair<String, Box<Integer>> p = new MyPair<>("primes", new
Box<Integer>(...));
```

Code listing 2: generic methods

```
// Code from
// https://docs.oracle.com/javase/tutorial/java/generics/methods.html

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

public class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

Method invocation:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
// boolean same = Util.compare(p1, p2);  // more compact
```

Code listing 3: bounded type parameters

```
public class Box<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){
        // De T sólo podemos asumir que es un Object
        System.out.println("T: " + t.getClass().getName());
        // Sabemos que U es al menos un Number
        System.out.println("U: " + u.getClass().getName()+ u.intValue());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(10));
        integerBox.inspect(new Float(1.2)); // ok, Float extends Number
        integerBox.inspect("some text"); // compiler error!

    }
}
```

The compiler error is:

```
<U>inspect(U) in Box<java.lang.Integer> cannot be applied to
(java.lang.String)
```

Code listing 4: bounded type parameters

```
public class NaturalNumber<T extends Integer> {

    private T n;

    public NaturalNumber(T n)  { this.n = n; }

    public boolean isEven() {
        return n.intValue() % 2 == 0;   // calls Integer.intValue
    }

    // ...
}
```

Code listing 5: generic methods and bounded type parameters

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

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

Code listing 5b: multiple bounded type parameters

```
public class MBounded<T extends Box<Integer> & Comparable<Integer>> {
    public MBounded(T elem) {
      // the interface of both Box and Comparable can be used on elem
       elem.set(4);
       Box<Integer> box = new Box<>();
       box.set(5);
       System.out.println(elem.compareTo(box.get()));
    }
}
```

Code listing 6: generics in C++

```
// Code from https://www.tutorialspoint.com/cplusplus/cpp_templates.htm

#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <stdexcept>

using namespace std;

template <class T>
class Stack {
   private:
      vector<T> elems;     // elements

   public:
      void push(T const&);      // push element
      void pop();               // pop element
      T top() const;            // return top element
      bool empty() const{       // return true if empty.
         return elems.empty();
      }
```

```
};

template <class T>
void Stack<T>::push (T const& elem) {
   // append copy of passed element
   elems.push_back(elem);
}

template <class T>
void Stack<T>::pop () {
   if (elems.empty()) {
      throw out_of_range("Stack<>::pop(): empty stack");
   }

   // remove last element
   elems.pop_back();
}

template <class T>
T Stack<T>::top () const {
   if (elems.empty()) {
      throw out_of_range("Stack<>::top(): empty stack");
   }

   // return copy of last element
   return elems.back();
}

int main() {
   try {
      Stack<int>         intStack;  // stack of ints
      Stack<string> stringStack;    // stack of strings

      // manipulate int stack
      intStack.push(7);
      cout << intStack.top() <<endl;

      // manipulate string stack
      stringStack.push("hello");
      cout << stringStack.top() << std::endl;
      stringStack.pop();
      stringStack.pop();
   } catch (exception const& ex) {
      cerr << "Exception: " << ex.what() <<endl;
      return -1;
   }
}
```

Reflection

Code listing 7: creating an object and invoking a method through reflection

```
// Code from http://stackoverflow.com/a/2215872

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class Foo {

    public void printAMessage() {
        System.out.println(toString()+": a message");
    }
    public void printAnotherMessage(String theString) {
        System.out.println(toString()+": another message: " + theString);
    }

    public static void main(String[] args) {
        Class<?> c = null;
        try {
            String name= "Foo";  // usually read from file
            c = Class.forName(name);
            Method method1 = c.getDeclaredMethod("printAMessage",
                             new Class<?>[0]);
            Method method2 = c.getDeclaredMethod("printAnotherMessage",
                             new Class<?>[]{String.class});
            Object o = c.newInstance();
            System.out.println("this is my instance: " + o.toString());
            method1.invoke(o);  // null parameter if method was static
            method2.invoke(o, "this is my message");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (NoSuchMethodException nsme){
            nsme.printStackTrace();
        } catch (IllegalAccessException iae) {
            iae.printStackTrace();
        } catch (InstantiationException ie) {
            ie.printStackTrace();
        } catch (InvocationTargetException ite) {
            // use ite.getTargetException() to handle the exceptions
            // thrown by the invoked method...
            ite.printStackTrace();
        }
    }
}
```

Output:

```
this is my instance: Foo@6d06d69c
Foo@6d06d69c: a message
Foo@6d06d69c: another message: this is my message
```

Code listing 8: factory pattern with reflection

Non-scalable solution:

```
public static Shape getFactoryShape (String s) {
    Shape temp = null;
    if (s.equals ("Circle"))
        temp = new Circle ();
    else if (s.equals ("Square"))
        temp = new Square ();
    else if (s.equals ("Triangle")
        temp = new Triangle ();
    else
        // ...
        // continues for each kind of shape
    return temp;
}
```

Reflection-based solution:

```
public static Shape getFactoryShape (String s) {
    Shape temp = null;
    try {
        temp = (Shape) Class.forName ("es.ua"+s).newInstance ();
    }
    catch (Exception e) {
        ...
    }
    return temp;
}
```

Code listing 9: finding an inherited method

```
Method findMethod(Class cls, String methodName, Class[] paramTypes) {
    Method method = null;
    while (cls != null) {
        try {
```

```
            method = cls.getDeclaredMethod(methodName, paramTypes);
            break;
        } catch (NoSuchMethodException ex) {
            cls = cls.getSuperclass();
        }
    }
    return method;
}

...

findMethod(Shape.class, “equals”, new Class[]{Object.class});
```