ITI 1121. Introduction to Computing II

Parametric polymorphism

by

Marcel Turcotte

Version February 2, 2020

Preamble

Preamble

Overview

Overview

Parametric polymorphism

We will see that generic types allow the design of data structures that can save objects of various classes without compromising the static analysis of the type of expressions. These concepts will be used to design robust abstract data types.

General objective :

- This week you will be able to describe the mechanisms by which generic data structures can be designed in Java without compromising static type analysis.

Preamble

Learning objectives

Learning objectives

- **Use** a parameterized type.
- **Define** a new generic type.
- **Design** a generic class method.
- **Recognize** situations where generic types are advantageous.
- **Explain** in your own words how generic types lead to robust applications.

Readings:

- `https://docs.oracle.com/javase/tutorial/java/generics/index.html`

Preamble

Plane

Plan

1 Preamble

2 Motivation

3 Generics

4 Méthodes génériques

5 Prologue

Motivation

Generic data structures

Let's consider the example of the class **Pair** again. This time, we want to save the references of two objects of type **Time**.

- What are the **instance variables**?
- Give the signature of the **instance methods**?

Time Pair

```
public class Pair {
  private Time first;
  private Time second;
  public Pair(Time first, Time second) {
    this.first = first;
    this.second = second;
  }
  public Time getFirst() {
      return first;
  }
  public Time getSecond() {
      return second;
  }
}
```

⇒ `new Pair(new Time(14,30), new Time(16,0));`

Shape Pair

```
public class Pair {
  private Shape first;
  private Shape second;
  public Pair(Shape first, Shape second) {
    this.first = first;
    this.second = second;
  }
  public Shape getFirst() {
      return first;
  }
  public Shape getSecond() {
      return second;
  }
}
```

⇒ `new Pair(new Circle(0,0,0), new Rectangle(1,1,1,1);`

Discussion

Problem: We want to design a class **Pair** that we can use to save objects of **various types**.

- What will be the type of **variables** and the **parameters**?

Pair

```
public class Pair {
  private _______ first;
  private _______ second;
  public Pair(_______ first, _______ second) {
    this.first = first;
    this.second = second;
  }
  public _______ getFirst() {
      return first;
  }
  public  _______ getSecond() {
      return second;
  }
}
```

Caveat: a temporary solution

- We're first exploring an avenue using the **concepts** that we have been seen in class so far.
- That solution was actually the **only possible one** in 2004.
- This solution doesn't offer **any type safety**.

Find the type of the variable first!

- It must be possible to save the reference from **any object** in the variable **first**.
  - What's its type?
- What is the **most general type** in Java?
- What is the **most general class** in Java?

Pair

```
public class Pair {
  private Object first;
  private Object second;
  public Pair(_____ first, _____ second) {
    this.first = first;
    this.second = second;
  }
  public _____ getFirst() {
      return first;
  }
  public  _____ getSecond() {
      return second;
  }
}
```

Pair

```
public class Pair {
  private Object first;
  private Object second;
  public Pair(Object first, Object second) {
    this.first = first;
    this.second = second;
  }
  public _____ getFirst() { // type of the return value?
      return first;
  }
  public _____ getSecond() {
      return second;
  }
}
```

Pair

```
public class Pair {
  private Object first;
  private Object second;
  public Pair(Object first, Object second) {
    this.first = first;
    this.second = second;
  }
  public Object getFirst() {
      return first;
  }
  public Object getSecond() {
      return second;
  }
}
```

Pair

- How is this class used?

```
Pair p;

String a;
a = "King";
String b;
b = "Edward";

p = new Pair(a, b);

a = p.getFirst();
b = p.getSecond();
```

- Do you see a problem with these statements?

Pair

```
Pair p;

String a;
a = "King";
String b;
b = "Edward";

p = new Pair(a, b);

a = (String) p.getFirst();
b = (String) p.getFirst();
```

- The class textbfObject is more general than the class textbfString! Each time an element is removed from a data structure, the type of the return value must be forced (type cast).
- There will be an error at runtime if the object is not of type **String**.
- With this **type cast**, we're depriving ourselves of any help the compiler could give us.

Generics

Generics

Java 1.5 was a major update *. It introduced the concept of **generic types** (“**generics**”).

```
public class Pair<T> {
    ...
}
```

The class declaration has a **(formal) type parameter**. This is the type of the objects that will be saved by the objects of the class **Pair**.

*This was 2004.

Usage

The value of **T** must be specified when declaring a variable.

```
Pair<String> name;
Pair<Integer> range;
```

as well as when creating an object:

```
name = new Pair<String>("Hillary", "Clinton");
```

```
Integer min;
min = new Integer(0);

Integer max;
max = new Integer(100);

range = new Pair<Integer>(min, max);
```

Generic types: A winning solution

- Victories on two fronts:
  - A **single** implementation of the class **Pair** that is reused in multiple contexts (it saves references of objects of various types).
  - All this, without sacrificing the **validation of the types** at the compilation stage.

Generics

- A **generic type** is a type with **formal type parameters**.
- A **parameterized type** is an instantiation of a **generic type** with an **actual type argument**.

Generics

Defining a generic type.

- A **generic type** is a **reference type** having one or more parameters of type.
- A **generic type** is a class with one or more type parameters.

```java
public class Pair<T> {

    private T first;
    private T second;

    public Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }
    public T getFirst() {
        return first;
    }
    public T getSecond() {
        return second;
    }
    public void setFirst(T value) {
        first = value;
    }
    public void setSecond(T value) {
        second = value;
    }
}
```

Generics

- What did we get?
  - "Generic" and "robust" types.

```
Pair<Integer> range;
range = new Pair<Integer>(new Integer(0),new Integer(10));

Integer i;
i = range.getFirst();
```

Compilation errors!

The declared object of type **Pair<Integer>**, can only be used to save the reference of an object of type **Integer**.

```
range.setFirst("Voila");
```

will produce a **compile time error**, and that's what we're hoping for!

```
Test.java:20: setFirst(java.lang.Integer)
in Pair<java.lang.Integer> cannot be applied to
(java.lang.String)
        range.setFirst("Voila");
              ^
1 error
```

More compilation errors.

In the same way, a reference variable having the following compilation type **Pair**<**Integer**> cannot designate an object whose parameterized type is **Pair**<**String**>. The statement

```
range = new Pair<String>("Hillary", "Clinton");
```

will produce a compilation error,

```
Test.java:22: incompatible types
found   : Pair<java.lang.String>
required: Pair<java.lang.Integer>
        range = new Pair<String>("Hillary", "Clinton");
                ^
1 error
```

```java
public class Pair<X,Y> {

    private X first;
    private Y second;

    public Pair(X first, Y second) {
        this.first = first;
        this.second = second;
    }
    public X getFirst() {
        return first;
    }
    public Y getSecond() {
        return second;
    }
    public void setFirst(X value) {
        first = value;
    }
    public void setSecond(Y value) {
        second = value;
    }
}
```

Parameterized type

When using a generic **type**, in this case **Pair**, we must provide **type values** for each **formal type parameter**.

```
public class Test {
    public static void main(String[] args) {

        Pair<String,Integer> p;

        String attribute;
        attribute = new String("height");

        Integer value;
        value = new Integer(100);

        p = new Pair<String,Integer>(attribute, value);
    }
}
```

Quiz: are these statements valid?

```
Pair<String,Integer> p;
```

```
p = new Pair<String,Integer>();
```

```
p.setFirst("session");
```

```
p.setSecond(12345);
```

Quiz: are these statements valid?

```
public class T1 {
    public static void main(String[] args) {

        Pair<String,Integer> p;

        p = new Pair<Integer,String>();

    }
}
```

```
// > javac T1.java
// T1.java:6: incompatible types
// found   : Pair<java.lang.Integer,java.lang.String>
// required: Pair<java.lang.String,java.lang.Integer>
//       p = new Pair<Integer,String>();
//               ^
// 1 error
```

Quiz: are these statements valid?

```
public class T2 {
    public static void main(String[] args) {
        Pair<String,Integer> p;
        p = new Pair<String,Integer>();
        p.setFirst(12345);
        p.setSecond("session");
    }
}
```

```
T2.java:5: setFirst(java.lang.String) in
  Pair<java.lang.String,java.lang.Integer> cannot be applied to (int)
        p.setFirst(12345);
         ^
T2.java:6: setSecond(java.lang.Integer) in
  Pair<java.lang.String,java.lang.Integer> cannot be applied to (java.lang.String)
        p.setSecond("session");
         ^
2 errors
```

Quiz: are these statements valid?

```
public class T3 {
    public static void main(String[] args) {
        Pair<String,Integer> p;
        p = new Pair<String,Integer>("session", 12345);
        Integer s = p.getFirst();
    }
}
```

```
// > javac T3.java
// T3.java:8: incompatible types
// found   : java.lang.String
// required: java.lang.Integer
//       Integer s = p.getFirst();
//                                ^
// 1 error
```

Generics

Generics are used to define “general” data structures while detecting type errors at compile time!

Generic types and interfaces

- The **interface** also allows us to define a **reference type**.
- The **interface** can also have one or more type parameters.

Interface as generic type: Comparable.

```
public interface Comparable {
    int compareTo(Comparable other);
}
```

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

Time: without a parameterized type

```
public class Time implements Comparable {

    private int timeInSeconds;

    public int compareTo(Comparable obj) {

        Time other = (Time) obj;
        int result;
        if (timeInseconds < other.timeInSeconds) {
            result = -1;
        } else if (timeInSeconds == other.timeInSeconds) {
            result = 0;
        } else {
            result = 1;
        }
        return result;
    }
}
```

Time: with a parameterized type

```
public class Time implements Comparable<Time> {

    private int timeInSeconds;

    public int compareTo(Time other) {

        int result;
        if (timeInseconds < other.timeInSeconds) {
            result = -1;
        } else if (timeInSeconds == other.timeInSeconds) {
            result = 0;
        } else {
            result = 1;
        }
        return result;
    }
}
```

Discussion

```java
public interface Comparable {
    int compareTo(Comparable other);
}
```

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

- **Without** generic type, the contract we had asked us to implement a **compareTo** method whose parameter was of type **Comparable**. We then had to force the type, which could cause a runtime error.
- **With** a generic type, the contract specifies that one must implement a method **compareTo** whose type is the same as that of the class in question, here **Time** or **Person**.

Person: without parameterized type

```
public class Person implements Comparable {

    private int id;
    private String name;

    public int compareTo(Comparable obj) {
        Person other = (Person) obj;
        int result;
        if (id < other.id) {
            result = -1;
        } else if (id == other.id) {
            result = 0;
        } else {
            result = 1;
        }
        return result;
    }
}
```

Person: with parameterized type

```
public class Person implements Comparable<Person> {

    private int id;
    private String name;

    public int compareTo(Person other) {

        int result;
        if (id < other.id) {
            result = -1;
        } else if (id == other.id) {
            result = 0;
        } else {
            result = 1;
        }
        return result;
    }
}
```

```java
public class Pair<String> {

    private String first;
    private String second;

    public String getFirst() {
        return first;
    }
    public String getSecond() {
        return second;
    }
    public void setFirst(String value) {
        first = value;
    }
    public void setSecond(String value) {
        second = value;
    }
}
```

Generic methods

Generic class method

```
public class Test {
    public static <E> void display(E[] xs) {
        for (int i=0; i<xs.length; i++) {
            System.out.println(xs[i]);
        }
    }
}
```

```
Random generator;
generator = new Random();

Integer[] xs;
xs = new Integer[5];

for (int i=0; i<5; i++) {
    xs[i] = generator.nextInt(100);
}

display(xs);
```

```
> java TestDisplay
78,95,53,21,7
```

```
String[] words;
words = new String[7];

words[0] = "alpha";
words[1] = "bravo";
words[2] = "charlie";
words[3] = "delta";
words[4] = "echo";
words[5] = "foxtrot";
words[6] = "golf";

display(words);
```

```
> java TestDisplay
alpha,bravo,charlie,delta,echo,foxtrot,golf
```

Utils.max

```
public class Utils {

    public static <T extends Comparable<T>> T max(T a, T b) {
        if (a.compareTo(b) > 0) {
            return a;
        } else {
            return b;
        }
    }

}
```

Call to a generic class method

- The **call** to a generic class method **does not** (usually) require any additional syntax elements.
- The compiler makes the type inference automatically.

```
Integer i1, i2, iMax;

i1 = new Integer(1);
i2 = new Integer(10);

iMax = Utils.max(i1, i2);

System.out.println("iMax = " + iMax);
```

Here, the compiler infers that the parameter type must be **Integer**.

Calling a generic class method

```
String s1, s2, sMax;

s1 = new String("alpha");
s2 = new String("bravo");

sMax = Utils.max(s1,s2);

System.out.println("sMax = " + sMax);
```

Here, the compiler infers that the parameter type must be **String**.

Specify the value of the type parameter

We can still **specify** the type:

```
iMax = Utils.<Integer>max(i1, i2);
sMax = Utils.<String>max(s1, s2);
```

```java
public class SortAlgorithms {

    public static <T extends Comparable<T>>
                        void selectionSort(T[] xs) {

        for (int i = 0; i < xs.length; i++) {

            int min = i;

            for (int j = i+1; j < xs.length; j++) {
                if (xs[j].compareTo(xs[min]) < 0) {
                    min = j;
                }
            }

            T tmp = xs[min];
            xs[min] = xs[i];
            xs[i] = tmp;
        }
    }

}
```

A new syntax for loops

Java 1.5 had also introduced a new syntax for loops.

```
int[] xs;
xs = new int[] {1, 2, 3};

int sum = 0;

for (int x : xs) {
    sum += x;
}
```

Prologue

Summary

- A **generic type** is a **reference type** with a **formal type** parameter.
- A **parameterized type** is an instantiation of a generic type with an **actual type parameter**.
- Generic types allow the creation of "general" data structures; general in the sense that they save references to objects of various types **without compromising type validation**.

Next module

- **Abstract Data Type (ADT)**: stacks

References I


E. B. Koffman and Wolfgang P. A. T.
***Data Structures: Abstraction and Design Using Java*.**
John Wiley & Sons, 3e edition, 2016.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

École de **science informatique** et de génie électrique (SIGE)

Université d'Ottawa