Object Oriented Programming in Java

8: Generics

Licence

You are free to

- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material

under the following terms

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **NonCommercial** — You may not use the material for commercial purposes.
- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

- The samples and slides are inspired by the Object Oriented Programming course at the University of Zagreb, Faculty of Electrical Engineering and Computing, Zagreb, Croatia.
  - Original materials in Croatian were created by (in alphabetical order): Ivica Botički, Marko Čupić, Mario Kušek, Boris Milašinović, and Krešimir Pripužić under CC-BY-NC-SA licence.
  - Adapted for this course by Boris Milašinović and shared under the same licence
  - https://creativecommons.org/licenses/by-nc-sa/4.0/

Motivation for generics (1/4)

- Let's define a class for storing (unchangeable) pair of integers.

08_Generics/…/example1/IntPair.java

```
package swu.oopj.generics.example1;
public class IntPair {
	private int first;
	private int second;
	public IntPair(int x, int y) {
		this.first = x;
		this.second = y;
	}
	public int getFirst() { return first; }
	public int getSecond() { return second; }
	@Override
	public String toString() {
		return "(" + first + ", " + second + ")";
	}
}
```

Motivation for generics (2/4)

- Let's do the same thing for pair of strings.
  - Changes from IntPair are colored with red color

08_Generics/…/example1/StringPair.java

```
package swu.oopj.generics.example1;
public class StringPair {
        private String first;
        private String second;
        public StringPair(String x, String y) {
                this.first = x;
                this.second = y;
        }
        public String getFirst() { return first; }
        public String getSecond() { return second; }
        @Override
        public String toString() {
                return "(" + first + ", " + second + ")";
        }
}
```

Motivation for generics (3/3)

- We can notice that only changes in code are just type change
  - Code redundancy, repetition of the same code, …
- Usage is also almost the same

08_Generics/…/example1/Main.java

```
package swu.oopj.generics.example1;
public static void main(String[] args) {
        IntPair ip = new IntPair(3, 5);
        System.out.println("First = " + ip.getFirst());
        System.out.println("Second = " + ip.getSecond());
        System.out.println("Pair = " + ip.toString());

        StringPair sp = new StringPair("A", "B");
        System.out.println("First = " + sp.getFirst());
        System.out.println("Second = " + sp.getSecond());
        System.out.println("Pair = " + sp.toString());
}
```

Using Object instead of concrete type (1/2)

- Can we replace types to be Object?
  - partially – int is a primitive type and this is not possible, but let's ignore this problem for a while

```
public class Pair {
	private Object first;
	private Object second;
	public Pair(Object x, Object y) {
		this.first = x;
		this.second = y;
	}
	public Object getFirst() { return first; }
	public Object getSecond() { return second; }
	@Override
	public String toString() {
		return "(" + first + ", " + second + ")";
	}
}
```

Using Object instead of concrete type (2/2)

- Storing reference using *Object* type (upcast) leads to many downcasts later

```
Pair p = new Pair("A", "B");
String first = (String) p.getFirst();
String second = (String) p.getSecond();
```

- More important (and the much bigger problem) is that compiler cannot check do we use correct types
  - e.g. the following code would cause ClassCastException

```
Pair p = new Pair("A", new Food(…));
String second = (String) p.getSecond(); //ClassCastException
```

Parametrized classes

- Class is parametrized using type parameter T
  - A type variable can be any non-primitive type (class type, interface type, array type, another type)

08_Generics/…/example2/Pair.java

```
public class Pair<T> {
	private T first;
	private T second;
	public Pair(T x, T y) {
		this.first = x;
		this.second = y;
	}
	public T getFirst() { return first; }
	public T getSecond() { return second; }
	@Override
	public String toString() {
		return "(" + first + ", " + second + ")";
	}
}
```

Using parametrized types

- Class Pair is parametrized. The T in Pair<T> is ***type parameter***
- We can have pair of strings, pair of Food, pair of whatever we want
  - We declare desired pair type using angle brackets providing ***type argument*** inside.
  - Right side of the expression can have empty angle brackets as compiler knows from declaration which type argument must be used.

```
Pair<String> p = new Pair<>("ABC", "DEF");
```

```
Pair<Food> p = new Pair<>(new Food(...), new Food(...));
```

- If needed, classes can be parametrized with more than one type parameter, e.g. SomeClass<T, U, V>
  - The usage concept is the same:
    SomeClass<String, Food, String> c = new SomeClass<>(…)

Primitive types and their wrappers

- A primitive type (byte, short, int, long, char, boolean, float or double) cannot be used as type argument in parametrized classes!
- However there exists their wrappers (Byte, Short, Integer, Long, Character, Boolean, Float and Double)
  - Classes that wrap immutable primitive value
  - Allows us to have e.g. Pair<Integer>
  - Additionally have several useful static methods, e.g. Double.toHexString(2.15), Integer.parseInt("343") …
  - … and some instance methods to “unwrap” the value
- Note:
  - Byte, Short, Integer, Long, Float, and Double extends Number
    - This information would be useful later

Boxing

- Wrapping primitive type to an object of the wrapper class type is called ***boxing***

08_Generics/…/boxing_unboxing/BoxingUnboxing.java

```
int a = 10;
Integer x = Integer.valueOf(a); //boxing
```

- Value from the stack is copied (wrapped, boxed) in a new, immutable object on the heap containing value of *a,* and *x* stores address of the wrapper object
  - As wrappers are immutable, they can be cached and reused. Thus references to same boxed values can be the same.

Unboxing

- Extracting boxed value from a wrapper is called ***unboxing***

08_Generics/…/boxing_unboxing/BoxingUnboxing.java

```
int a = 10;
Integer x = Integer.valueOf(a); //boxing
int c = x.intValue(); //unboxing
```

- Value wrapped (boxed) on the heap is copied to the stack.

- Reminder: We can change the x as a reference (with x = somethings else), but boxed value is immutable.
  - Naturally, we can change the values of *a* and *c* as they are just memory locations of integers on the program stack

Autoboxing and autounboxing

- Java compiler automatically generates code for boxing and autoboxing. Thus we can write

```
int a = 10;
Integer x = a; //autoboxing: Integer.valueOf(a);
int c = x; //autounboxing: x.intValue();
```

- Also some more complex situations are supported, e.g.

```
int a = 10;
Integer x = Integer.valueOf(a); //boxing
int c = x.intValue(); //unboxing
int y = 20;
Integer z = x  + y;
// => (auto)unboxing x: x.intValue() + y  => produces int
// => (auto)boxing to z Integer.valueof(x.intValue() + y);
System.out.println("" + x + " + " + y + " = " + z);
// => uses StringBuilder to
// append strings and primitive values
```

08_Generics/…/boxing_unboxing/BoxingUnboxing.java

Parametrized type advantages

- No need to (explicitly) do upcast and downcast
- Compiler checks type correctness, e.g.

  Pair<String> p = new Pair<>("ABC", 123);

  leads to compiler error due to second argument mismatch (String is expected)
- Return values are known to compiler in advance, e.g.
  - *Pair<String> p* means that *p.getFirst()* is of type *String*
    - We can then write *p.getFirst().length()*
  - *Pair<Food> p* means that *p.getFirst()* is of type *Food*
    - We can write *p.getFirst().getPrice()* but not *p.getFirst().length()*
  - We cannot send wrong type to a method
    - See **08_Generics/…/example2/Main.java** for details

What happens internally in Java

- To implement generics, the Java compiler applies type erasure
  - Replaces all type parameters in generic types with their bounds (more about later) or *Object* if the type parameters are unbounded.
  - The produced bytecode 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.

Type erasure could be misused

- Due to Java type erasure concept, the following code is valid but not recommended

```
Pair iPair = new Pair(5,5);
Pair sPair = new Pair("X", "Y");
```

  - Compiler raises warning
  - After these statements we have lost type safety.
    - See **08_Generics/…/example2/RawPairMain.java** for details

Parametrized classes and inheritance

- Parametrized classes are new types that do not replicate type arguments inheritance relations
  - E.g. if class Food extends Item, then Pair<Food> does not extend Pair<Item>
- Parametrized classes can extend another classes and inheritance relation is preserved if appropriate types are used as type arguments, e.g. if *Pair<T>* extends *Ntuple<T>* (i.e. pair is special case of a n-tuple) then
  - Pair<String> is subclass of Ntuple<String>
  - Pair<Food> is subclass of Ntuple<Food>
  - Pair<Item> is subclass of Ntuple<Item>
  - Food extends Item but Pair<Food> is not subclass of Ntuple<Item> (or any other combination is valid if different types are used)

Method parametrization (1)

- An instance method can use parametrized types of its class, but it can be additionally parametrized
  - Type parameter is written before return value type

08_Generics/…/example3/Pair.java

```
public class Pair<T> {
    ...
    public <V> void printWith(V another) {
        System.out.format("first: %s second %s %n",
                          this.toString(), another.toString());
    }
```

  - Method type argument does need to be specified if it is obvious from provided arguments

08_Generics/…/example3/Main.java

```
Pair<Integer> iPair = new Pair<>(5, 5);
Pair<String> sPair = new Pair<>("A", "B");
iPair.printWith(sPair);
iPair.printWith("OOP");
```

Method parametrization (2)

- Provided parametrization is not much useful as V could be any type (e.g. String, Food, Pair<String>, …)
  - Only thing we now about V is that it is subclass of *Object*, and only methods inherited from Object could be used

08_Generics/…/example3/Pair.java

```
public class Pair<T> {
    ...
    public <V> void printWith(V another) {
        System.out.format("first: %s second %s %n",
                          this.toString(), another.toString());
    }
```

08_Generics/…/example3/Main.java

```
Pair<Integer> iPair = new Pair<>(5, 5);
Pair<String> sPair = new Pair<>("A", "B");
iPair.printWith(sPair);
iPair.printWith("OOP");
```

Method parametrization (3)

- Method parametrization could be used to specify type argument for the method's argument
  - Thus we now that another is Pair and we can use Pair's methods

08_Generics/…/example3/Pair.java

```
public class Pair<T> {
    ...
    public <V> void printWithPair(Pair<V> another) {
        System.out.format("first: %s second %s,%s %n",
                this.toString(), another.getFirst().toString(),
                                 another.getSecond().toString());
    }
```

08_Generics/…/example3/Main.java

```
Pair<Integer> iPair = new Pair<>(5, 5);
Pair<String> sPair = new Pair<>("A", "B");
iPair.printWithPair(sPair);
iPair.printWithPair(iPair);
//iPair.printWithPair("OOP"); //compile error
```

Method parametrization – wildcard "?" (1)

- In the previous example we didn't have any variable of type V nor it has been used anywhere in method. Purpose of V is to specify that argument is a Pair of something that can differ in type from the class type argument
  - In that case wildcard *?* can be used
    - *Notice that there is no <?> before return type*

```
public class Pair<T> {
    ...
    public void printWithPair(Pair<?> another) {
        ...
    }
```

08_Generics/…/example3/Pair.java

```
Pair<Integer> iPair = new Pair<>(5, 5);
Pair<String> sPair = new Pair<>("A", "B");
iPair.printWithPair(sPair);
//iPair.printWithPair("OOP"); //compile error
```

08_Generics/…/example3/Main.java

Method parametrization – wildcard "*?*" *(2)*

- Notice that there is no <?> before return type
- "?" can be used only for arguments parametrization, not for defining types
  - i.e. it is not possible to write

    void <?> printWith(? another) or ? some_variable = …

Static methods and parametrization

- Static methods cannot use class type parameters (neither as variables nor as arguments), but can use its own type parameters
- Classes cannot have static variables of parametrizes types

```
public class Pair<T> {
    static T t1; //compile error
    ...
    public static <V> void someMethod(Pair<V> arg) {
        V v; //OK
        T t; //compile error
         ...
    }
```

- Compile error:

  Cannot make a static reference to the non-static type T

Bounded type parameters – upper bound

- Type parameters could be bounded (lower and upper bound)
- Upper bound: T extends R
  - T can be casted to R
    - T is R, T is subclass or R (directly or indirectly), or T or T's superclasses implements R
  - E.g. Pair<T extends Number> means that Pair is pair of Integers,pair of Floats, …, and for value of getFirst() is guaranteed to be instance of Number (i.e. can be stored in reference of type Number)
- Multiple bounds can be specified using &
  - class SomeClass<T extends S & R & Q> { … }
  - T can be casted to S, R and Q. None or one of S, R and Q could be class type, others are interfaces

Upper bound - example 1 (1/2)

- Reminder: Integer extends Number
  - Number is abstract but *Pair<Number> num1 = new Pair<>(10, 5);* is valid due to autoboxing from int to Integer, and Integer is Number
  - Pair<Integer> is not subclass of Pair<Number>

```
public static void main(String[] args) {
        Pair<Number> num1 = new Pair<>(10, 5);
        Pair<Integer> num2 = new Pair<>(10, 3);
        m1(num1); //OK – num1 is Pair<Number>
        //m1(num2); //compile error
        ...
}

static void m1(Pair<Number> num) {
        System.out.println(num);
}
```

08_Generics/…/example4/Main.java

Upper bound - example 1 (2/2)

- m2 accepts Pair of everything that can be casted to Number
  - Thus it can accept Pair<Integer>, Pair<Number>, but not Pair<String>

```
public static void main(String[] args) {
        Pair<Number> num1 = new Pair<>(10, 5);
        Pair<Integer> num2 = new Pair<>(10, 3);
        ...
        m2(num1);
        m2(num2);

        Pair<String> ps = new Pair<>("A", "B");
        //m2(ps); //compile error
}
static <T extends Number> void m2(Pair<T> num) {
        System.out.println(num);
}
```

08_Generics/…/example4/Main.java

Upper bound - example 2 (1/2)

- Write a methods that counts how many elements are greater than some argument
- Same algorithm for any type. The only thing we need how to compare elements
- Initial idea for methods signature

```
public static <T> int cnt(T[] data, T x) {…}
```

  - What we know about T? Nothing more that it's superclass is Object and only methods we can use are defined in Object class
- We should bound method type parameter to comparable types, i.e. those that implements interface Comparable<T> with method *compareTo*

Upper bound - example 2 (2/2)

- We can use methods with arrays of Strings, Integers, but not e.g. with arrays of Items.

```
public static void main(String[] args) {
        Integer[] a = {1, 2, 3, 4, 5, 6};
        System.out.println(countGreaterThan(a, 2));
        String[] b = {"Boris", "Jack", "Marco",
                      "Alice", "Bob", "Peter"};
        System.out.println(countGreaterThan(b, "Boris"));
}
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;
        }
```

08_Generics/…/example5/Main.java

Bounded type parameters – wildcards and lower bound

- Lower bound is specified using keyword super
  - E.g. *void <T super Beverage> method(T arg)* means that argument *arg* is Beverage or some of its subclasses (Item, Object)
- Wildcard ? Could be used in combinaton with super and extends
  - <? extends E> (almost) same as <T extends E>
  - <? super E>

- These wildcards would be intensively used (and discussed) in forthcoming lectures.
  - If someone if already interested more can be found on these links:

https://docs.oracle.com/javase/tutorial/extra/generics/methods.html
https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html

Limitations of Java Generics (1)

- We cannot (directly) create new object of class type parameter

```
class Gen<T> {
  T obj;
  void test(){
    obj = new T(); //not possible
  }
}
```

- Reminder: type argument must be class, interface, array, another type, but not a primitive type:
  - Gen<int> is not possible, but Gen<Integer> is OK

Limitations of Java Generics (2)

- We cannot declare array of class type parameter, but we cannot create it directly

```
class Gen<T> {
  T[] arr; //O.K.
  Gen(int size){
    arr = new T[size]; //not possible
    Gen<Integer>[] gens = new Gen<Integer>[20];//not possible
```

- However, there are some workarounds than causes compiler warnings that are suppressed with annotation

```
@SuppressWarnings("unchecked")
```

```
Gen<?>[] gens = new Gen<?>[20];
arr = (T[]) new Object[size];
Gen<Integer>[] gens = (Gen<Integer>[]) new Gen<?>[20];
```

  - Note: Using bounds complicates previous code (instead of new Object[size], new Bound[size] is used, etc…)

Limitations of Java Generics (3)

- Class type parameter cannot be used for static variables and methods.
  - However, static methods can have its own type parameters (use of the same name is possible, but not recommended)

```
class Gen<T> {
  static T obj; //compiler error
  static T test(){  //compile error
    ...
  }
  static <V> V test(V arg, T arg) { //compile error
    T t; //compile error
  }
  static <V extends Number> V test(V arg) { V v; } //OK

  static <T> void test3(T x) { //some another T
    T t; //OK – it is not T from the class
  }
```