ENTERPRISE JAVA PROGRAMMING / PROGRAMMING II

MODULES CT545 / CT875, SEMESTER 2, ACADEMIC YEAR 2020-2021
NATIONAL UNIVERSITY OF IRELAND, GALWAY

Lecture 2a

Lecturer: Dr Patrick Mannion

TOPICS

- More about *generics* (*generic programming*) in Java

GENERIC PROGRAMMING

- *Generic programming (aka "generics")* allows for *types parameters, a* sort of placeholders for types
- The main benefits of generics are an increased level of *type safety* and the elimination of certain type casts
- Heavily used in Java and - albeit not in identical form - in many other object-oriented programming languages (such as C#), and missing from some others (e.g., C and Go), with very negative consequences for the programmer.

- ArrayList<Person> = new ArrayList<Person>();

GENERIC PROGRAMMING

- Similar from the users' point of view, but not technically similar approaches are, e.g.:
  - Templates (C++)
  - Parametric polymorphism (Haskell, Scala)
- Generic programming can be seen as (yet another) form of polymorphism.
  However, generics are very different from polymorphism by method overriding

GENERIC PROGRAMMING

- Motivating example:
  Assume you want to implement an algorithm for sorting a list of items.
- The algorithm should work <u>independently</u> from the types of the list elements - it should work with, e.g., numbers as well as with strings without any need to create two different versions of the same algorithm implementation (one for numbers, one for strings)
- We want to use the same piece of code for sorting all kinds of items (string, numbers, ...)
- Furthermore, we want the algorithm to be *type safe*: it should <u>not</u> be allowed to, e.g., compare a string with a number, and violations should be discovered <u>before</u> runtime. Java generics achieve this.

GENERIC PROGRAMMING

- But before we look at generics, we try to achieve one of the previously described goals (namely, the avoidance of code duplication) using known means (namely, class inheritance), just to provide some motivation for generics...

GENERIC PROGRAMMING

- Nothing new is involved in this "pseudo-generic" approach:
  - Challenge: a number of classes $g_{1..n}$ are identical except for one place in their code where a different type c_i is used within each g_i
  - In order to reduce the size of our program, we want to replace all these classes $g_{1..n}$ with a *single* class (type)
  - Solution: instead of the $g_{1..n}$, implement only <u>one</u> class g, and replace the c_i with a common <u>*superclass*</u> c of the c_i (class `Object` in the extreme case)
  - Sounds complicated, but it is just a precise formulation of a plain and simple use case of <u>inheritance</u>: a superclass can be used in place of its subclasses if none of the specific features of the subclasses are required

GENERIC PROGRAMMING

- Class g is then sometimes called *pre-Java 5 generic class*,
- But it's better to use the term "generic" only in connection with real generics (Java 5 or later)
- We will see that this works to some degree, but that it introduces problems, in particular wrt. type safety

GENERIC PROGRAMMING

Example: an implementation using a pre-Java 5 class `MemoryCell`:

```
// MemoryCell class
//  Object read( )          -->  Returns the stored value
//  void write( Object x ) -->  x is stored

public class MemoryCell
{
        // Public methods
    public Object read( )          { return storedValue; }
    public void write( Object x ) { storedValue = x; }

        // Private internal data representation
    private Object storedValue;
}
```

Example code in this lecture from Mark A. Weiss: Data Structures and Algorithm Analysis in Java

GENERIC PROGRAMMING

- The following slide just shows how we could apply this "pseudo-generic" class...

GENERIC PROGRAMMING

```java
public class TestMemoryCell
{
    public static void main( String [ ] args )
    {
        MemoryCell m = new MemoryCell( );

        m.write( "37" );
        String val = (String) m.read( );
        System.out.println( "Contents are: " + val );
    }
}
```

GENERIC PROGRAMMING

- The methods in class `MemoryCell` allow us to store/retrieve values of multiple different types (i.e., classes) in/from simulated memory cells
- Because we wanted to create only a single class `MemoryCell`, we use superclass `Object` within `MemoryCell` as a placeholder for *any type* of the values which should be stored/retrieved in the cell.
- E.g., method `void write(Object x)` works with *all* kinds of types of `x`, because `Object` is the superclass of *all* classes in Java and can be used (here) in place of more specific subclasses

GENERIC PROGRAMMING

- In other words:
  We simply utilized plain inheritance (here: from class `Object`)…

  If we didn't have inheritance (super-/subclasses), we would need to create multiple variants of class `MemoryCell` (one for storing strings in memory cells, another class for storing integers in memory cells, yet another one for doubles...)

GENERIC PROGRAMMING

- The previous approach is very simple, but requires a superclass (like `Object` in the example before).
- Alternatively, be could use an interface instead of a superclass...

GENERIC PROGRAMMING

- The example on the next slide shows an example for this, again using pre-Java 5 / "pseudo-generic" code
- All objects within the array which is passed as argument to method `findMax` need to have a class which implements interface `Comparable`, in order to make these objects *comparable*
  - `findMax` then simply returns the "largest" element in an array of objects
- For now, we can ignore the details of interface `Comparable`, we just look at its use
- In the example on the next slide, we use it in order to compare various geometric shapes... We assume that the shape-classes (e.g., `Circle`) all implement interface `Comparable`

```java
class FindMaxDemo
{
    /**
     * Return max item in a.
     * Precondition: a.length > 0
     */
    public static Comparable findMax( Comparable [ ] a )
    {
        int maxIndex = 0;

        for( int i = 1; i < a.length; i++ )
            if( a[ i ].compareTo( a[ maxIndex ] ) > 0 )
                maxIndex = i;

        return a[ maxIndex ];
    }

    /**
     * Test findMax on Shape and String objects.
     */
    public static void main( String [ ] args )
    {
        Shape [ ] sh1 = { new Circle(  2.0 ),
                          new Square(  3.0 ),
                          new Rectangle( 3.0, 4.0 ) };

        String [ ] st1 = { "Joe", "Bob", "Bill", "Zeke" };

        System.out.println( findMax( sh1 ) );
        System.out.println( findMax( st1 ) );
    }
}
```

GENERIC PROGRAMMING

- Again, if we couldn't use a common interface or a superclass for the items in the array in method `findMax`, we would need to create *multiple* versions of method `findMax` instead (one for finding the largest string in an array of strings, one for finding the largest number, ...!)
- A method like `findMax` is sometimes called a *pre-Java 5 generic method.* But again, it is better to avoid the terms "generic" for pre-Java 5 code.

GENERIC PROGRAMMING

- The approach seen on the previous slides using inheritance is extremely simple, but it comes at a high price:
- The use of a common superclass (or interface) impairs type safety, because it only "sees" the superclass (or interface), but not the specific types for which, e.g., `findMax` or `MemoryCell` are actually called
  => type checking becomes unspecific
  => less type safety
  => a lot more prone to programming errors
- If, e.g., you call `findMax` for an array of strings, the type checker couldn't warn you in case the array passed to the method accidentally contains a number!
- Furthermore, in case you want to specifically use the result of this method as, e.g., a string, you would need to type-cast it to type `String`

GENERIC PROGRAMMING

- The much better approach:

  Use real (Java >=5) generics...

GENERIC PROGRAMMING

- Java supports a special syntax for (real) generics
- "Generic" in Java >=5 means "parameterized by *type parameters*"

- Syntax:

```
class MyGenericClass<AnyType> {
    ...
}
```

a type parameter

GENERIC PROGRAMMING

- `AnyType` is called the *type parameter*.
- Each time the class `MyGenericClass` is used (e.g., in order to create an object of this class), we can pass a concrete type for the type parameter.
  Example:
  ```
  MyGenericClass<String> myObject =
      new MyGenericClass<String>()
  ```
- The concrete type which is passed for the type parameter is called *type argument* (in the example above, the type argument is `String`)
- You can use any (unused) name for the type parameter, but it should start with an uppercase letter

GENERIC PROGRAMMING

- `AnyType` can be used inside of the body of class `MyGenericClass` as a "placeholder" for concrete types (that is, reference types)
- Inside of the body of the class, `AnyType` is called a *type variable*
- In the class body, the type variable can be used almost anywhere where a concrete type (class or interface name) could be used. This way, the same class works with multiple different concrete types represented by the type variable.
- Class `MyGenericClass<AnyType>` is called a *generic class*
- The class name should always be noted together with the <type parameter>, e.g., write `MyGenericClass<AnyType>` instead of `MyGenericClass` (although the latter is also allowed by Java)
- Interfaces can also be generic (entirely analogously to classes)

GENERIC PROGRAMMING

Class `MemoryCell` again, but now as a *generic class*, where `AnyType` serves as a placeholder for concrete classes or interfaces:

type parameter

```
public class GenericMemoryCell<AnyType>
{
    public AnyType read( )
      { return storedValue; }
    public void write( AnyType x )
      { storedValue = x; }

    private AnyType storedValue;
}
```

type variables

GENERIC PROGRAMMING

- As a well-known other example, consider the Java Collection class `ArrayList` (a class for *resizable* arrays)
- This class can be used as an ordinary class, but also in form of a generic class, which should <u>always</u> be preferred over the non-generic form!
- Old (non-generic) way of use, e.g.:

```
ArrayList myListOfAnything = new ArrayList();
```

- New, generic way, where `ArrayList<...>` is a generic class:

```
ArrayList<String> myListOfStrings =
     new ArrayList<String>();
```

- You can add and retrieve elements to/from this array-list like this:

```
myListOfStrings.add("some string item");
              // ^ adds a new element at the end, size of array-list grows by one
int size = myListOfStrings.size();   // current number of elements
String itemF = stringList.get(0);   // retrieves the first element
String itemL = stringList.get(size-1);   // retrieves the last element
```

GENERIC PROGRAMMING

- As you can see, the generic form contains more information for the type checker:
  `ArrayList`**`<String>`** can contain <u>only</u> strings (and objects of subclasses of strings), not, e.g., numbers
- So the Java compiler (which does the type checking before runtime of the program) can show you an error in case you write by mistake, e.g., `myListOfStrings.add(123);`
- The type parameter of `ArrayList` can be instantiated with other types too, e.g., `ArrayList<Integer>` or `ArrayList<Rectangle>`

GENERIC PROGRAMMING

The generic version of interface `Comparable`:

```
package java.lang;

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

now with a type parameter ("AnyType") which gets instantiated with a concrete type when using the interface "Comparable"

You can use (e.g., implement) this interface like this:

```
class myClass implements Comparable<MyClass> {
      ...
}
```

The concrete type

GENERIC PROGRAMMING

- Using (real, >=Java 5) generics provides the following benefits:
  1) increase the *type safety* of the code (that's the main feature of Java generics), and
  2) provide higher comfort for the programmer by avoidance of type casts and code duplication
- With generics, many run-time type errors become compile-time errors, which are much easier to debug.
- Furthermore, error-prone type casting can be avoided because the types being used in the code are already more specific (so they don't need to be casted to subclasses)

GENERIC PROGRAMMING

- E.g., without generics, passing objects with an inappropriate (non-comparable) class to method `compareTo(...)` in the previous variant of interface `Comparable` would lead to a `ClassCastException` at runtime:

```
public interface Comparable {
    public int compareTo(Object o);
}
```

- Generics allow the compiler to detect such problems already *before* runtime ("statically")

GENERIC PROGRAMMING

- A generic class can basically be used like a normal class or interface, e.g.,
  - to create instances (objects) of this class, or
  - as type of a variable, or
  - as type of a method parameter or return value, or
  - to derive a subclass

- Remark: a generic class or interface can have multiple type parameters, e.g., MyGenericClass<T1, T2, T3> { … }

GENERIC PROGRAMMING

- As mentioned before, anywhere a generic class (or a generic interface) is *used*, the type parameter needs to be replaced with a concrete reference type (i.e., the name of a class or an interface) - the *type argument*.

  Three further examples:

- `MyGenericClass<String> obj = new MyGenericClass<String>();`

- `class AnotherClass extends MyGenericClass<Double> { ... }`

- `void myMethod(int a1, MyGenericClass<String> a1, String a3) {…`

GENERIC PROGRAMMING

- An example for wrong use:

```
MyGenericClass<String> obj = new MyGenericClass<Integer>();
```

GENERIC PROGRAMMING

- Type *bounds* allow you to restrict the range of type parameters (i.e., it puts constraints on the concrete types that type parameter represents):

  `class MyGenC <TypeParameter` **`extends Sup`**`> ...`

  creates a generic class where any concrete type for which `TypeParameter` stands must be either a subclass of class `Sup`,
  or the class for which `TypeParameter` stands must implement interface `Sup`

GENERIC PROGRAMMING

- More complex type bounds exist (in the lecture we cover only the basic cases)
- Later, you will see further examples for bounds. For the moment, you just need to know what (type) "bounds" means

GENERIC PROGRAMMING

- One (wrong!) way to intuitively conceive Java generics would be to imagine that each time a generic class is used (e.g., if an object of this class is created), every occurrence of the type variable in the body of the generic class would be physically *replaced* with the respective concrete type…
  E.g., `AnyType` being actually replaced by `String` in the body of `MyGenericClass` if there is
  ```
  MyGenericClass<String> obj = new MyGenericClass<String>();
  ```
  elsewhere
- However, this is <u>not</u> what Java actually does…
  (but more or less to how C++ *templates* are implemented)

GENERIC PROGRAMMING

- As we have seen, the foremost purpose of generics over pseudo-generics is to avoid type errors at compile time
- In fact, after type checking at compile time, they are not needed anymore and therefore generics are then translated by the compiler into "ordinary" (non-generic/or pre-Java5-"generic") Java code, using an approach called *type erasure*...

GENERIC PROGRAMMING

- Slightly simplified description of type erasure:
  1. The Java compiler (javac.exe) performs type checking

  Afterwards, it performs the actual type erasure:

  2. The compiler removes all type parameters
  3. It then replaces all type variables within the generic classes' body with their *bounds*, or with `Object` in case there is no bound (e.g., if the type parameter specification is `<AnyType extends SuperClass>`, `AnyType` is replaced with `SuperClass`)

GENERIC PROGRAMMING

- So what generic classes become after translation into simple Java looks not very different compared to the Pre-Java-5-"pseudo-generics" we have seen at the beginning of today's lecture (i.e., using superclasses or interfaces as types, instead of type variables)!
- However, the big difference is that the translation happens only *after type checking*. So type checking can still take advantage of the `<typeParameter>` information before the translation of the generics into simple Java code without generics

- In contrast to Java generics, C++ *templates* (which are similar to Java generics) are translated by creating a *copy* of the class for each instantiation of a type parameter)

GENERIC PROGRAMMING

- As said before, a generic class can basically be used like a non-generic class, for example as the type of a method parameter.

- Example:

```
void myMethod(MyGenericClass<String> argument) {

    System.out.println("Argument = " + argument);

}
```

GENERIC PROGRAMMING

- In the headers of method declarations which <u>use</u> generic classes (or generic interfaces), we can optionally use *wildcards* ("?") for concrete types. Optionally, they can even have bounds. E.g.,

  - `void myMethod(MyGenericClass<?> arg1, …) {…`

    `?` stands for an unknown type

  - `void myMethod(MyGenericClass<? extends Sup> arg1, …) {…`

    `?` stands for a subclass/implementation of interface `Sup`, or for `Sup` itself.

  - `void myMethod(MyGenericClass<? super Sub> arg1, …) {…`

    `?` stands for a superclass of `Sub`, or for `Sub` itself. Again, `Sub` could also be an interface.

GENERIC PROGRAMMING

- A concrete example for a type wildcard in a parameter declaration:

```
void myMethod(int a1, GenClass<? super Employee> a2, boolean a3) {
    ...
}
```

A valid call of this method would be `myMethod(5, p, true);`

provided that, e.g., `p` is an object of class `GenClass<Person>` and `Person` is a superclass of class `Employee`.

GENERIC PROGRAMMING

- Wildcards can also be used for return values, e.g., :

```
GenClass<? extends Employee> myMethod(int a1, boolean a2) {
   ...
   return o;
 }
```

- The meaning of the above is:

  method `myMethod` returns an object `o` which has generic class `GenClass<x>` as type, where `x` is any subclass of `Employee`

GENERIC PROGRAMMING

- Wildcards for types can furthermore be used with variable declarations, like in

```
Collection<?> someCollection = new ArrayList<String>();
```

- Meaning: variable `someCollection` has a generic class as type. The generic class can be any `Collection<x>` where `x` stands for any class, e.g., `String`

- Use wildcard `<?>` without bounds with care (why?)

GENERIC PROGRAMMING

- To see why such wildcards can be useful, we need to understand the important concept of *covariance*…
- The term "covariance" is typically introduced in the context of ordinary arrays (e.g., Integer a[50])...
- *Covariant arrays* means: arrays of subclasses are (in Java) *type-compatible* with arrays of superclasses: in a sense, it can be said that an array `Sub[]` is a kind of "subclass" of an array `Sup[]` if `Sub` is a sub-class of `Sup`.
- For example, (only!) if `Employee` is a subclass of `Person`,

  `Person[] array = new Employee[5];` compiles fine.
  We could also pass `Employee[]` as an argument to a method which accepts `Person[]` as a parameter.

GENERIC PROGRAMMING

- But unfortunately a problem with covariance is that it can easily lead to type confusion at runtime:
- E.g., let `Employee` and `Student` be subclasses of `Person`. Then

  `Person[] array = new Employee[5];`

  `array[0] = new Student();` compiles fine,

  but leads to an `ArrayStoreException` error at runtime, because `array` stores objects of type `Employee`, not `Student`…
- To avoid such problems, <u>Java generics are *not* covariant</u>…
- Remark: That Java's arrays are covariant is widely considered a design mistake.

GENERIC PROGRAMMING

- It would for example not be possible to pass an `ArrayList<Square>` to the method shown on the next slide, even if `Square` is a subclass of `Shape`.
- In other words: Although `Square` is a sub-class of `Shape`, `ArrayList<Square>` is not a subclass of `ArrayList<Shape>`!
- So, although ordinary Java arrays are covariant, Java generics are not covariant but so-called *invariant.*
- This is true even for "array-like" collections such as `ArrayList<...>`

GENERIC PROGRAMMING

problematic

```java
public static double totalArea( ArrayList<Shape> arr )
{
    double total = 0;

    for( Shape s : arr )
        if( s != null )
            total += s.area( );

    return total;
}
```

GENERIC PROGRAMMING

However, we can use the aforementioned wildcards in the method signature to carefully workaround this restriction:

```
public static double totalArea( ArrayList<? extends Shape> arr )
{
    double total = 0;

    for( Shape s : arr )
        if( s != null )
            total += s.area( );

    return total;
}
```

GENERIC PROGRAMMING

- Another, albeit less important, form of generics besides generic classes and interfaces are *generic static methods*
- Like generic classes, generic static methods also allow for type parameters...
- Syntax:

```
static <TypeParameter> ReturnType methodName(...) {
  ...
}
```

(Don't confuse generic methods with the methods shown before which have wildcards like <?...> in their signatures! - they previous methods weren't generic, they just made *use of* generic classes)

GENERIC PROGRAMMING

- Generic static methods can be used regardless of whether the class in which this method is declared is a generic class or an ordinary class.
- In contract to generic classes, if a generic static method is used (i.e., called), it is **not** required to provide a concrete type for the type parameter.
- Instead, the compiler infers the type argument automatically from the types of the actual arguments

GENERIC PROGRAMMING

- In the following example, the type parameter is `T`:

Type parameter

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) { c.add(o); }
 }
```

generic static method

Type variable

- We could call this method with, e.g.,

```
fromArrayToCollection(new String[100], new ArrayList<String>());
```

- Using this call, `T` is assumed to stand for concrete type (class) `String`. In particular, the green **`T`** in the method's body above now stands for `String`.

GENERIC PROGRAMMING

- The same effect would not be possible using wildcards
- E.g., the following would not work because the wildcard `?` is too unspecific. It does not allow the compiler to infer that `c`'s `add()` method can add object `o`:

```
static void fromArrayToCollection(Object[] a, Collection<?> c) {
     for (Object o : a) {
            c.add(o); // Compile time error, because element type
       // of Collection<?> elements is ?, unknown
```

- At runtime, the concrete replacement for `?` would be known (e.g., `String` if the method was called with a second argument of type `Collection<String>`), but since Java is a statically typed language, this would come too late for the (static) type checking Java performs...
- In contrast, generics (whether generic classes or generic methods) are resolved statically already.