Secure OOP with Java

Lecture Unit - 08

Claudia Maderthaner <claudia.maderthaner@fh-hagenberg.at>

Generics

Generics

aka Parameterized Types

- Parameterize types, field, parameters or methods with types (type variables)

Container Types before Java 5

```
List list = new ArrayList();
list.add("hello");
String s1 = (String) list.get(0);

list.add(new Integer("42"));
String s2 = (String) list.get(1); // runtime error
```

- → not typesafe
- → `return` values need to casted

Container Types with Generics

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s1 = list.get(0); // look Ma - no cast!!

list.add(new Integer("42")); // compile error
```

Generic Type Declaration

```
class GenericClass<T1, ..., Tn> extends SuperClass implements I1, ..., In{
    // other class code
}
```

```
interface GenericInterface<T1, ..., Tn> extends I1, ..., In{
    // other interface code
}
```

```
package java.lang;

import java.util.*;

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

```
public class BigInteger extends Number implements Comparable<BigInteger> {
    // ...
    public int compareTo(BigInteger val) { ... }
    // ...
}
```

Typ Variable

- Represents a type unknown at compile time
- Will be replaced with a concrete type when used
- Type variables may not represent primitive data types or array types

```
public class Container<T> {
    private T value;

    public void set (T value) {
        this.value = value;
    }

    public T get() {
        return value;
    }
}
```

```
Container<String> c1 = new Container<String>();
c1.set("hi!");
String content = c1.get();

Container<LocalDate> c2 = new Container<LocalDate>();
c2.set(LocalData.now());
LocalDate date = c2.get();
```

Diamond Operator

```
Container<String> container = new Container<String>();

Container<String> anotherContainer = new Container<>(); // since Java 7
```

Type Variable Naming Conventions

E	Element
K	Key
N	Number
T	Type
V	Value
S, U, V	additional types

Bounds

- Used during declaration of generic types
- Declare a basic data type all object used must be assignable to

```
T extends Class

N extends Class & Interface
```

```
public class Container<N extends Number> {
    private N value;

    public int toInteger() {
        return value == null ? 0 : value.intValue();
    }
}
```

```
public class Container<T extends Comparable<T>> {
    private T value;

    public T getBigger(T other) {
        return this.compareTo(other) > 1 ? this : other;
    }
}
```

```
public class Container<N extends Number & Comparable<N>> {
    private N value;

    public int toInteger() {
        return value == null ? 0 : value.intValue();
    }

    public N getBigger(T other) {
        return this.compareTo(other) > 1 ? this : other;
    }
}
```

Inheritance and Generics

A generic class may inherit

- from a non-generic class

```
class A<T> extends B { ... }
```

- from a parameterized generic class

```
class A<T> extends B<String> { ... }
```

- from a generic class using the same type variable

```
class A<T> extends B<T> { ... }
```

Inheritance and Generics

A non-generic class may inherit

- from parameterized generic classes

```
class A extends B<String> { ... }
```

Type Erasure

- The generic type variable is removed during compilation
- There are **no** type variables during runtime
- The generic type is replaced with the bound type or `Object`
- The compiler adds
  - type casts in the byte code if necessary
  - bridge methods to keep up polymorphism

Raw Types

- During compile time different parameterizations represent different data types
- At runtime there is only **one** data type left
- Raw types circumvent generic type checks
  - → compiler warnings
  - → runtime exceptions

Limitations of Generic Types

- Static members may not use type variables of the class
- A class may not be a sub type of a type variable

```
class Prohibited<T> implements T { }
```

- It is no possible create objects or arrays with type variables

```
new T()
new T[]
```

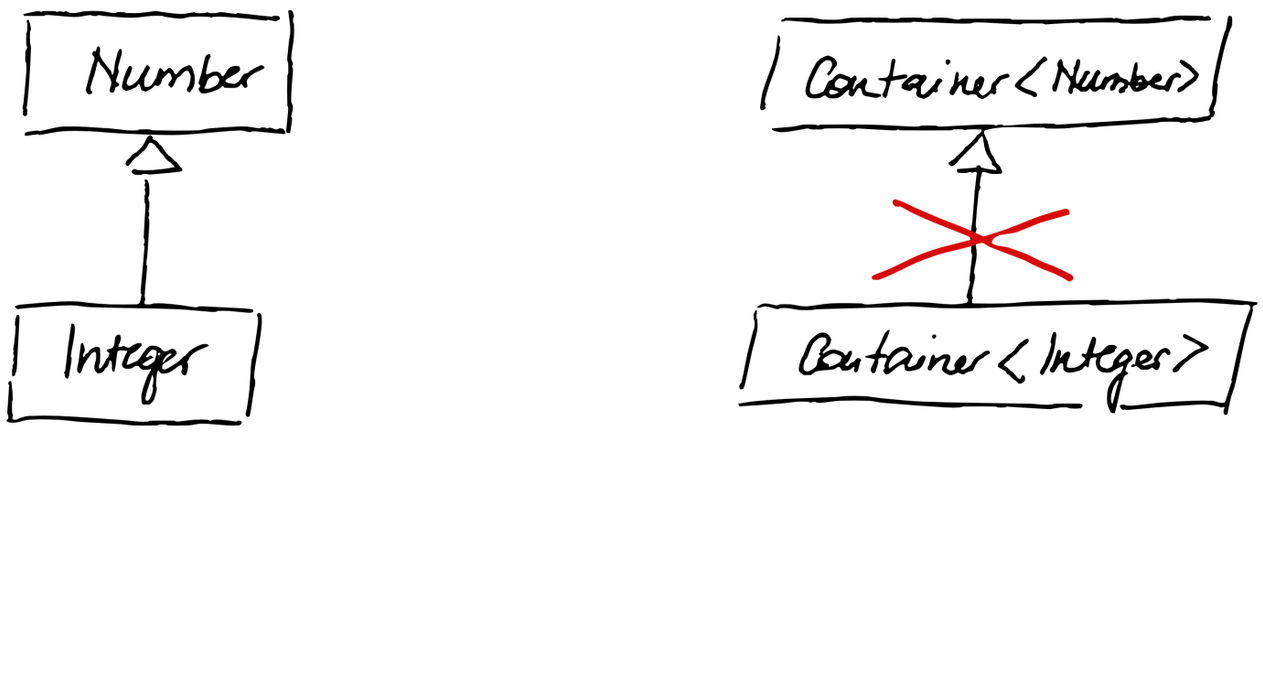
- `instanceof` does not work in most of the cases
- A type may not implement different parameterization of the same interface
- It is not allowed to use type variables in `catch` clauses

Generic Arrays

```
T[] genericArray = new T[size]; // compile error
```

```
T[] genericArray = (T[]) new Object[size];
```

Inheritance and Sub Types



Generic Methods

```
public static <T> boolean addAll(Collection <? super T> collection, T... elements) {
    boolean result = false;
    for (T element : elements) {
        result |= c.add(element);
    }
    return result;
}
```

```
List<String> names = new ArrayList<>();
Collections.addAll(names, "Huey", "Dewey", "Luoie");

Collections.<String> addAll(names, "Scrooge", "Donald", null);
```

Type Inference

- The compiler infers the correct type suitable for the method call

Wildcards

- Represent a unknown type

Upper Bound

```
? extends SuperType
```

LowerBound

```
? super SubType
```

Wildcard Usage

- Used in
  - method arguments
  - fields
  - local variables
  - `return` values
- not allowed with
  - type arguments in method calls
  - during creation of new objects from generic types
  - as super type

Wildcards in Generic Methods

```
public int getAge(Container<? extends Calendar> date) { ... }

public static void printList(List<?> list) { ... }

public static void addNumbers(List<? super Integer> list) { ... }
```

Enumerations

Constants

```
public class Seasons {
    public static final int SPRING = 1;
    public static final int SUMMER = 2;
    public static final int AUTUMN = 3;
    public static final int WINTER = 4;
}
```

Enumeration Types

- Variation of classes which only allow a limited number of instances (named constants).
- The instances are defined in the enum. → It is not possible to create new objects.
- All enum types are derived from `java.lang.Enum<E>`.

Enumeration Declaration

```
enum EnumName {
    ENUM_CONSTANT_1,
    ENUM_CONSTANT_2;
}
```

```
package enumeration;

public enum Seasons {
    SPRING, SUMMER, AUTUMN, WINTER;
}
```

```
javap calendar\Seasons.class
Compiled from "Seasons.java"
public final class calendar.Seasons extends java.lang.Enum<calendar.Seasons> {
  public static final calendar.Seasons SPRING;
  public static final calendar.Seasons SUMMER;
  public static final calendar.Seasons AUTUMN;
  public static final calendar.Seasons WINTER;
  static {};
  public static calendar.Seasons[] values();
  public static calendar.Seasons valueOf(java.lang.String);
}
```

Enum Features

- Typesafe
- May have fields and/or methods
- May implement interfaces
- May not be sub-classed
- Enum constants can be used in `switch` statements

Enumeration Methods

`static T[] values()`

return an array with all available enum constants

`static <T extends Enum<T>> T valueOf(Class<T> enumClass, String name)`

return enum instance with given name

`final String name()`

returns the name of the enum constant

`final int ordinal()`

returns the zero-based position of this enumerated constant in the enum declaration

Enumeration Methods

`final boolean equals(Object other)`

returns `true` if the specified object is equal to the enum constant

`final int hashCode()`

returns a hash code for the enum constant

`final int compareTo(E other)`

compares enum constants by their position in the enum declaration

`String toString()`

returns the name of this enum, as contained in the declaration

Enum with Members

```
public enum RomanLiteral {

    I(1), V(5), X(10), L(50), C(100), D(500), M(1000);

    private int value;

    private RomanLiteral(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

Annotations

Annotation Types

- Special kind of interfaces
- The represent metadata
  - Information for the compiler
  - Processed during compilation and deployment
  - Evaluated during runtime

```java
@Deprecated(since="1.2", forRemoval=true)
public final void suspend() { ... }
```

```java
@Override
public int hashCode() { ... }
```

```java
@SuppressWarnings("unchecked")
Pair<T, U> other = (Pair<T, U>) obj;
```

```java
public void process(@NonNull final String param) { ... }
```

Annotation Type Declaration

```
package java.lang;

import java.lang.annotation.*;
import static java.lang.annotation.ElementType.*;

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, MODULE, PARAMETER, TYPE})
public @interface Deprecated {
    String since() default "";
    boolean forRemoval() default false;
}
```

Annotation Features

- All annotation types extend `java.lang.annotation.Annotation`
- May not extend another interface
- Annotation are not generic
- All annotation methods have no arguments and are not generic
- Allowed `return` types
  - Primitive types
  - Enum types
  - Annotation types
  - `java.lang.String`
  - `java.lang.Class`
  - Arrays of all the types above

Records

Value Objects

- Primarily hold values
- Do not implement other logic

```java
import java.util.Objects;

public class Address {

    private String street;
    private String city;

    public Address(String street, String city) {
        this.street = street;
        this.city = city;
    }

    public String getStreet() {
        return street;
    }

    public void setStreet(String street) {
        this.street = street;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    @Override
    public int hashCode() {
        return Objects.hash(city, street);
    }
```

Record Types

```
record RecordName(Type name, ...) {
    // constraints for object creation
}
```

```
public record Address(String street, String city) { }
```

- `getters`, `equals`, `hashCode`, `toString` and a constructor are automatically provided
- All record objects are immutable

Contact

Moodle Discussion Board

claudia.maderthaner@fh-hagenberg.at