UNIVERSITY OF Southampton
School of Electronics and Computer Science

COMP2212 PROGRAMMING LANGUAGE CONCEPTS

Julian Rathke and Pawel Sobocinski

VARIANCE

COVARIANCE AND CONTRAVARIANCE

T subset U and V subset V (itself), soT x U subset of U x V because

T subset U and V subset V

- We have seen that in the type formers for pairs (records) and sums (variants) there is a relationship between the subtyping on the types of substructure and the subtyping of the structure itself.
  - For example, if `T <: U` and `V <: V` then `T x V <: U x V`
- Notice that the ordering between **`T`** and **`U`** is somehow 'preserved' in the latter subtyping relation.
- This preservation of order has a particular name. We say that the pair type former is a **covariant** type constructor.
- We saw above that records and variants are both covariant type formers.
- There are type formers that do not behave like this.
- Indeed, suppose we had a type former called **`Foo`**.
- So that given a type **`T`**, then **`Foo T`** is a new type.
- If subtyping for **`Foo`** is such that: if **`T <: U`** then **`Foo U <: Foo T`** then we call **`Foo`** a **contravariant** type former.
- Let's look at real example of a contravariant type former.

FUNCTION TYPES AND CONTRAVARIANCE

- Subtyping interacts with function types in a very interesting way.
- Let's just look at the (general) type rule:

Every U1 is subset of T1

$$\frac{U_1 <: T_1 \qquad T_2 <: U_2}{T_1 \to T_2 <: U_1 \to U_2}\textsc{SubFun}$$

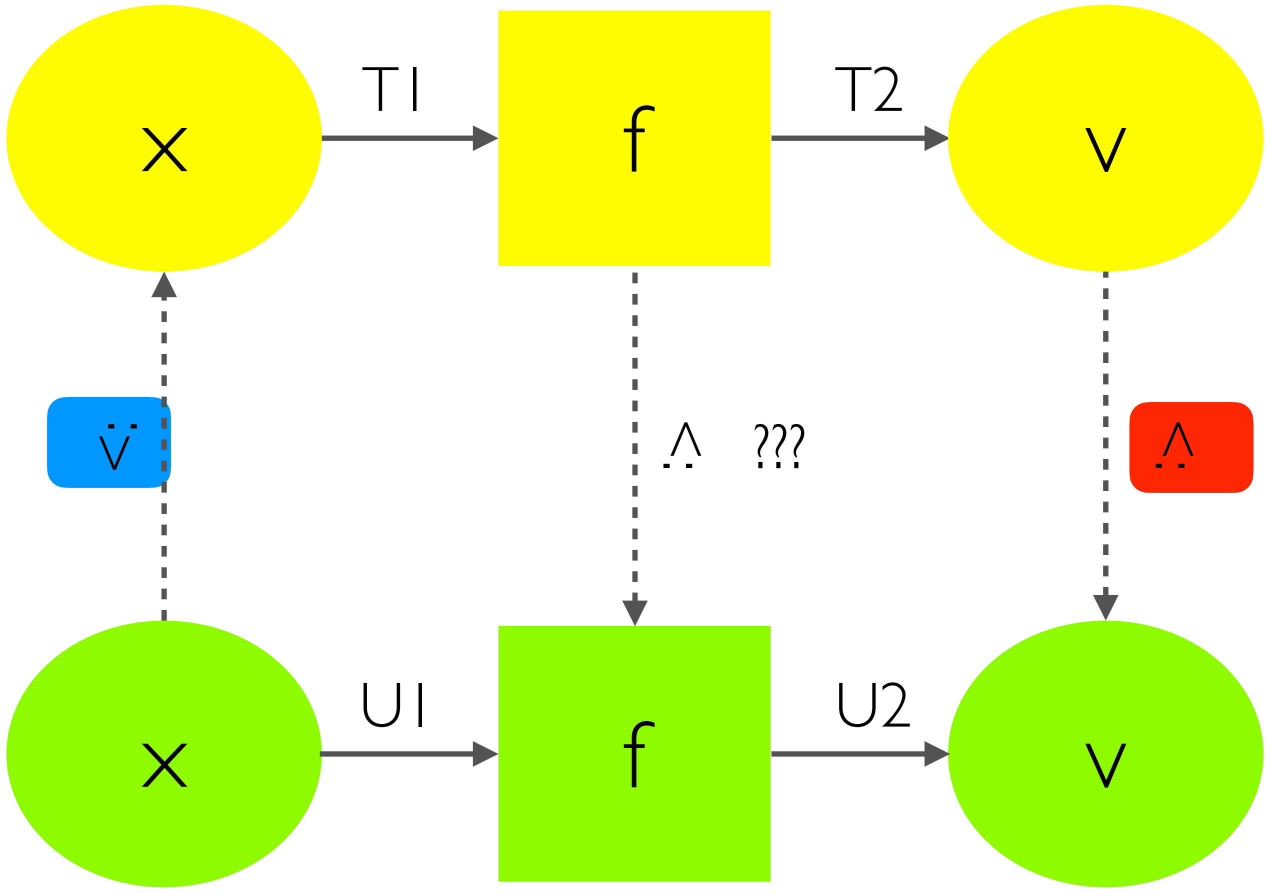
- What we see is that the function type former is covariant in its return type and contravariant in its argument type!

- Let's try and explain why this is:
  - Suppose we have a function `f:T₁ → T₂` then what can we do with it?
  - We can apply it to an argument `x`, say of type T_1,
  - But every U_1 is also a T_1 so `f` will accept any argument of type U_1 also.
  - Now, what does f return. It returns a value of type T_2.
  - But every value of type T_2 is also of type U_2, so f returns a value of type U_2.
  - That is, `f` can accept any argument of type U_1 and will return a value of type U_2.
  - That is, `f` is also a function of type $U_1 \to U_2$

FUNCTION TYPES AND CONTRAVARIANCE

$$\frac{\boxed{U_1 <: T_1} \qquad \boxed{T_2 <: U_2}}{T_1 \to T_2 <: U_1 \to U_2}\ \textsc{SubFun}$$

- Let's try see that pictorially:



OTHER STRUCTURES

- The List type former is covariant.
- This gives the simple subtyping rule

$$\frac{T <: U}{T \text{ List} <: U \text{ List}}\,\textsc{SubList}$$

- The situation for Arrays is a little more complicated.
- An array is non-structural in the sense that we don't build elements of the type using data constructors. Elements in the array can be modified.
- That is, the array must both consume data that it is given (array write) and produce data when requested (array read).
- These two operations have different variance requirements!

ARRAYS AND SUBTYPING

When an array is read, the Array type should be **covariant**

T <: U would imply T[ ] <: U[ ] so I can treat a T array as a U array. If I fetch data from the U array and actually get a value of T, that is okay because I can treat this T value as a U value.

When an array is written, the Array type should be **contravariant**

If T[ ] <: U[ ] and I write to what I think is a U[ ] (but is actually a T[ ] then this will be fine if I write a U value and U <: T because the T[ ] can accept this U value by pretending it is a T value.

We say that arrays are an **invariant** type former with subtyping rule:

$$\frac{T <: U \qquad U <: T}{T[\,] <: U[\,]}\ \textsc{SubArray}$$

SUBTYPING ARRAYS IN JAVA

- The type rule for Arrays in Java is **not** the rule I showed on the previous slide.
- Instead Java supports covariant array types

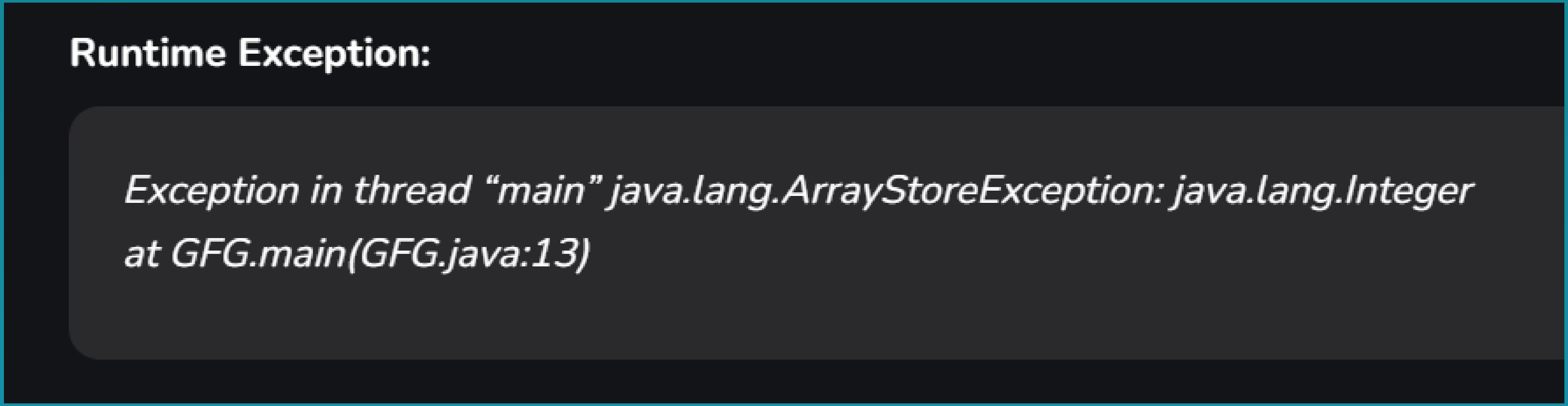
$$\frac{T <: U}{T[\,] <: U[\,]}\ \textsc{SubJavaArray}$$

- This may seem appealing as it certainly looks more flexible than invariant arrays.
- Yes, but it has the small drawback of being unsafe!
- That's right. Java is not type safe! Due to covariant array types.
- Here is proof:

```
class B extends A { }
class A {
    public static void main(String[] args){
     B[] b = new B[1];
     A.oops(b);
    }
    static void oops(A[] a){ a[0] = new A(); }
}
```

This code passes the Java type checker but throws a run time type error!

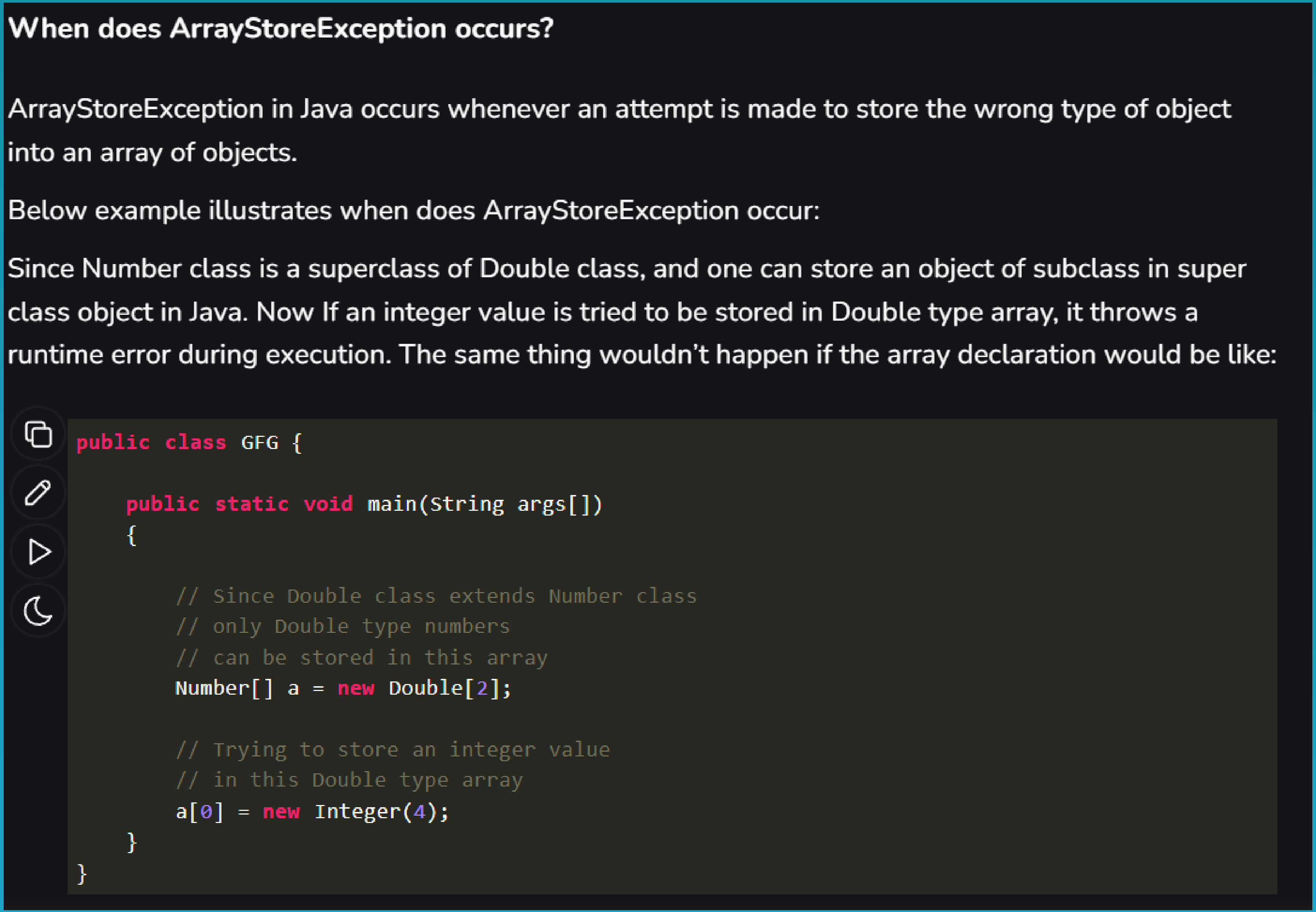
OOPS.



Runtime Exception:

```
Exception in thread "main" java.lang.ArrayStoreException: java.lang.Integer
at GFG.main(GFG.java:13)
```

NEXT LECTURE: TYPES FOR OBJECTS



When does ArrayStoreException occurs?

ArrayStoreException in Java occurs whenever an attempt is made to store the wrong type of object into an array of objects.

Below example illustrates when does ArrayStoreException occur:

Since Number class is a superclass of Double class, and one can store an object of subclass in super class object in Java. Now If an integer value is tried to be stored in Double type array, it throws a runtime error during execution. The same thing wouldn't happen if the array declaration would be like:

```java
public class GFG {

    public static void main(String args[])
    {

        // Since Double class extends Number class
        // only Double type numbers
        // can be stored in this array
        Number[] a = new Double[2];

        // Trying to store an integer value
        // in this Double type array
        a[0] = new Integer(4);
    }
}
```