Modeling Object Behavior with Interfaces

José Paumard
PHD, Java Champion, JavaOne RockStar

@JosePaumard https://github.com/JosePaumard

Agenda

Extending a class allows you add fields and methods to this class

It is about extending state and behavior

Interfaces are about modeling behavior

Understanding the Java type system

Converting an object to another type

Creating Interfaces

An interface may have:

- abstract methods;
- constants;
- concrete methods;
- static concrete methods

Default and static methods must be public

```
public interface Consumer<T> {

    void accept(T t);



}
```

The Consumer interface models any object that accept an object

```java
public interface Consumer<T> {

    void accept(T t);


    default Consumer<T> andThen(Consumer<T> other) {
        // Implementation
    }
}
```

The Consumer interface models any object that accept an object

It also defines how you can chain consumers

How can you instantiate interfaces?

To instantiate an interface

You need to implement it

- with a concrete class
- with lambda expressions

Implementing an interface consists in

- creating a class

- with concrete implementations of all the abstract methods of that interface

Or

- creating a lambda expression

You can define methods that accept interfaces as arguments

In this case, your methods are independent of the implementations used

Defining Types

There are two types in Java:

The object type: class, abstract class, and interface

The primitive type: byte, short, int, long, float, double, char and boolean.

Object types are references

Primitive types are values

You cannot get a reference to a value

The primitive types cannot extend each other, they are just values

Object types can extend each other:

- an interface can extend another interface

- a class can implement an interface

- a class can extend another class

An abstract class can extend
a single other class, abstract or concrete

An concrete class can extend
a single other class, abstract or concrete

Java has multiple inheritance of type

An interface can model a specific behavior:

- Iterable
- Comparable

A class can implement several interfaces

```
List<String> strings = new ArrayList<>()
```

```
List<String> strings = new ArrayList<>()
```

This is the type

It can be:

- an interface
- an abstract class
- a concrete class

```
List<String> strings = new ArrayList<>()
```

This is the type

It can be:

- an interface
- an abstract class
- a concrete class

This is the implementation

It can be:

- a concrete class
- a lambda expression

The type and the implementation must be compatible

The implementation
must be a subtype of the type

```
Comparable<String> string = "Hello world!"
```

This code compiles because String implements Comparable

The compiler sees `string` as an object of type `Comparable`
Because the compiler only sees the type of a variable

So the following code does not compile

```
int length = string.length()
```

The methods available on a variable are the ones defined on its type

Converting Numeric Types

There are two cases to consider:

- the types are object type

- the types are primitive types

You can convert an object type A to an object type B

If the type B is an extension of the type A

You can convert any numerical primitive type A to any other numerical primitive type B

It can be done implicitly if there is no loss of precision

```
int i1 = 10;
long l1 = i1; // does compile
```

Remember: implicit conversion is allowed if there is no loss of precision

```
int i1 = 10;
long l1 = i1; // does compile

long l2 = 10L;
int i2 = l2; // does not compile (possible loss of precision)
int i3 = (int)l2; // does compile
```

Remember: implicit conversion is allowed if there is no loss of precision

All the operations on non-floating point primitive types are executed using int or long

```java
short i1 = 10;
short i2 = 10;

short i3 = i1 + i2; // does not compile
```

The last line does not compile:

- the addition is executed with int, so i1 and i2 are first converted to ints
- then the result is also an int, that cannot be implicitly converted to a short

```
short i1 = 10;
short i2 = 10;

short i3 = i1 + i2; // does not compile
short i4 = (short)(i1 + i2); // does compile
```

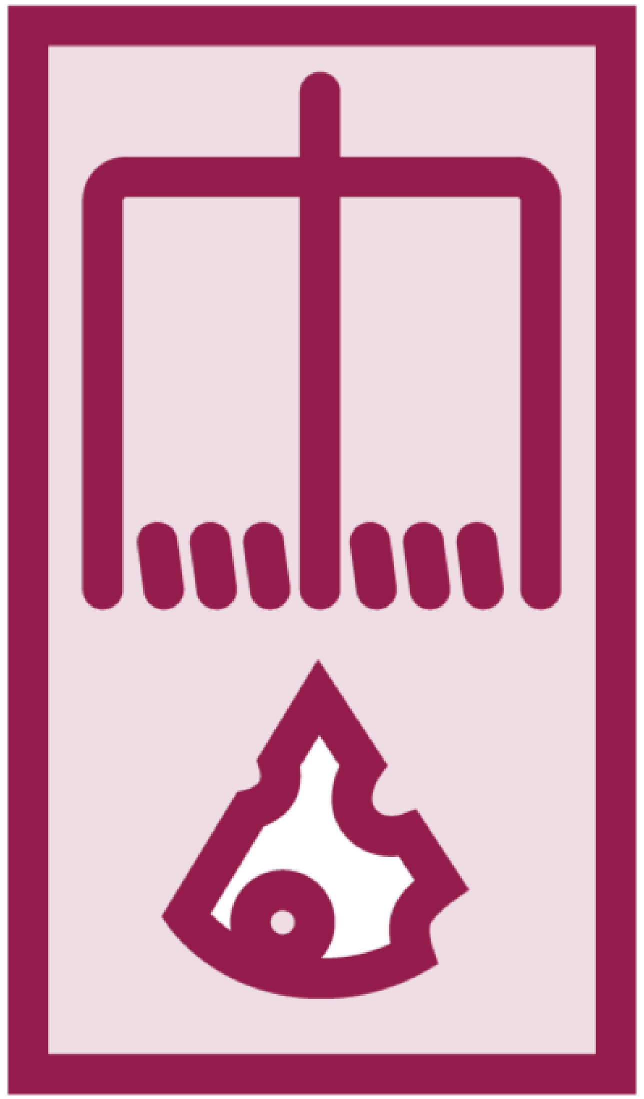
The last line does not compile:

- **the addition is executed with int, so i1 and i2 are first converted to ints**
- **then the result is also an int, that cannot be implicitly converted to a short**

```
float f1 = 3,14; // does not compile
float f2 = (float)3,14; // does compile
```

The last line does not compile:

- the addition is executed with int, so i1 and i2 are first converted to ints
- then the result is also an int, that cannot be implicitly converted to a short



Conversions between primitive types is very tricky

1) No implicit conversion that can lead to a loss of precision

2) Arithmetic operations are executed using ints or longs

Demo

Using interfaces to model behavior

Module Wrap Up

What did you learn?

How to create and use interfaces

What are types, type compatibility

Primitive types

Type conversion

Arithmetic for ints and longs

Up Next: Constructing an Object, Calling a Constructor from a Constructor