Expressions
Collections, Generics

Tutorial 5 (24th February 2021)

Radboud University

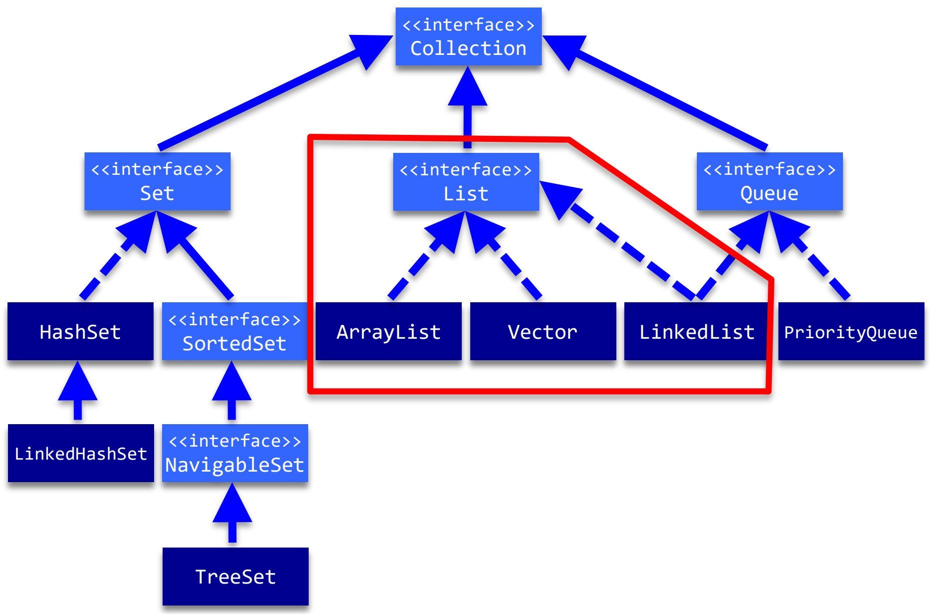
Lists

interface **`List`** is an extension of **`Collection`**

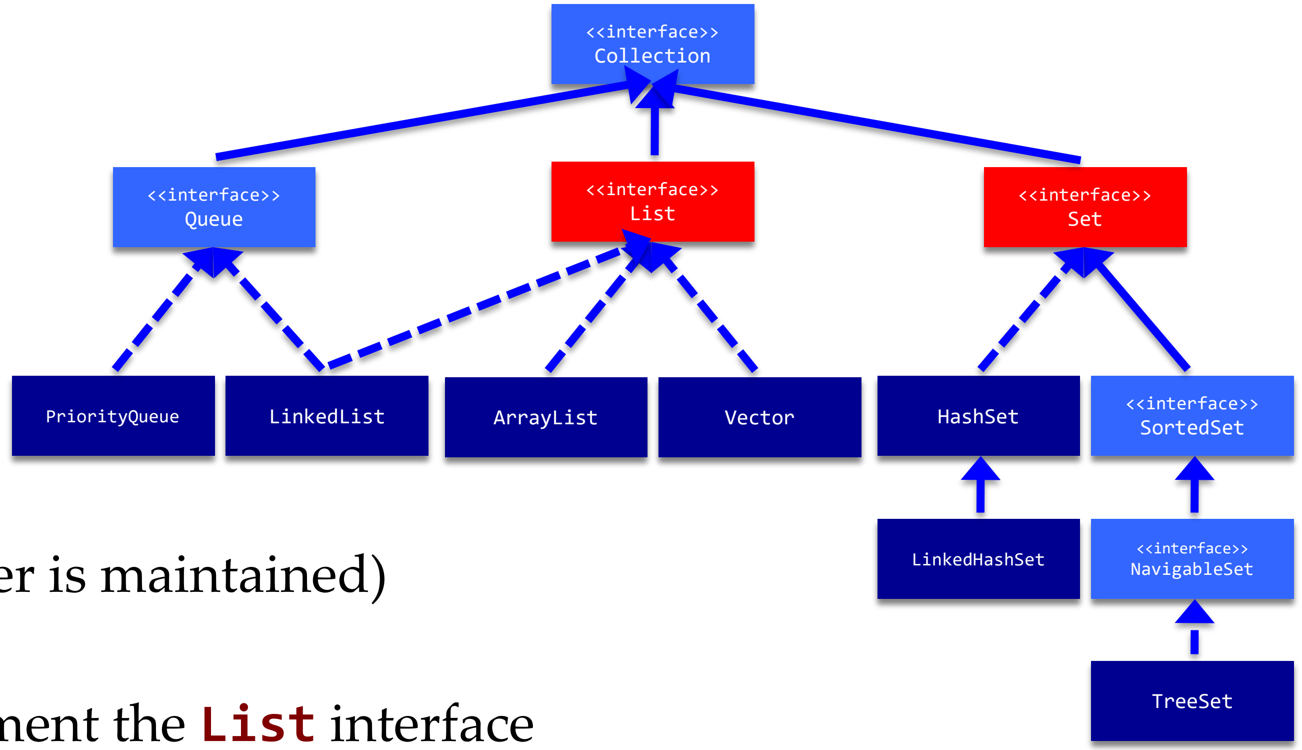
- `List` adds methods to manipulate elements via indices
  - `void add(int index, E element)`
  - `E get(int index)`
  - `E remove (int index)`
  - `E set (int index, E element)` …

`ArrayList` implements the interface **`List`**

other implementations are
`LinkedList` and **`Vector`**



collection relationships



`Set`

- does not contain duplicates
- can (sometimes) be sorted !

`List`

- elements are ordered (insertion order is maintained)
- elements can occur more than once
- **`ArrayList`** and **`LinkedList`** implement the **`List`** interface
  - next lecture we will discuss the differences between **`ArrayList`** and **`LinkedList`**
- **`Vector`** is very similar to **`ArrayList`** in API,
  vectors are thread-safe and hence somewhat slower

the class `Collections`

do not confuse it with the interface **`Collection`**

- contains algorithms for collections
- like **`Arrays`** for arrays

implemented algorithms:

```
    sort, binarySearch, reverse, shuffle,
    fill, copy, min, max, addAll,
  frequency, disjoint
```

three ways to access all list elements

```java
for (int i = 0; i < list.size(); i += 1) {
    Card card = list.get(i);
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen1: " + card);
    }
}
```

+ any order possible
− get(i) can be inefficient

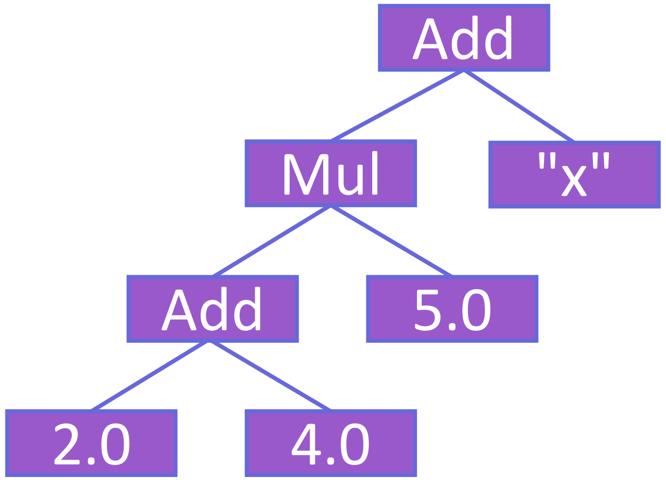
```java
for (Card card : list) {
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen2: " + card);
    }
}
```

+ compact
+ efficient
− list cannot be changed

```java
Iterator<Card> iter = list.iterator();
while (iter.hasNext()) {
    Card card = iter.next();
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen3: " + card);
    }
}
```

+ efficient
+ flexible
+ not restricted to loop
! remove only last item

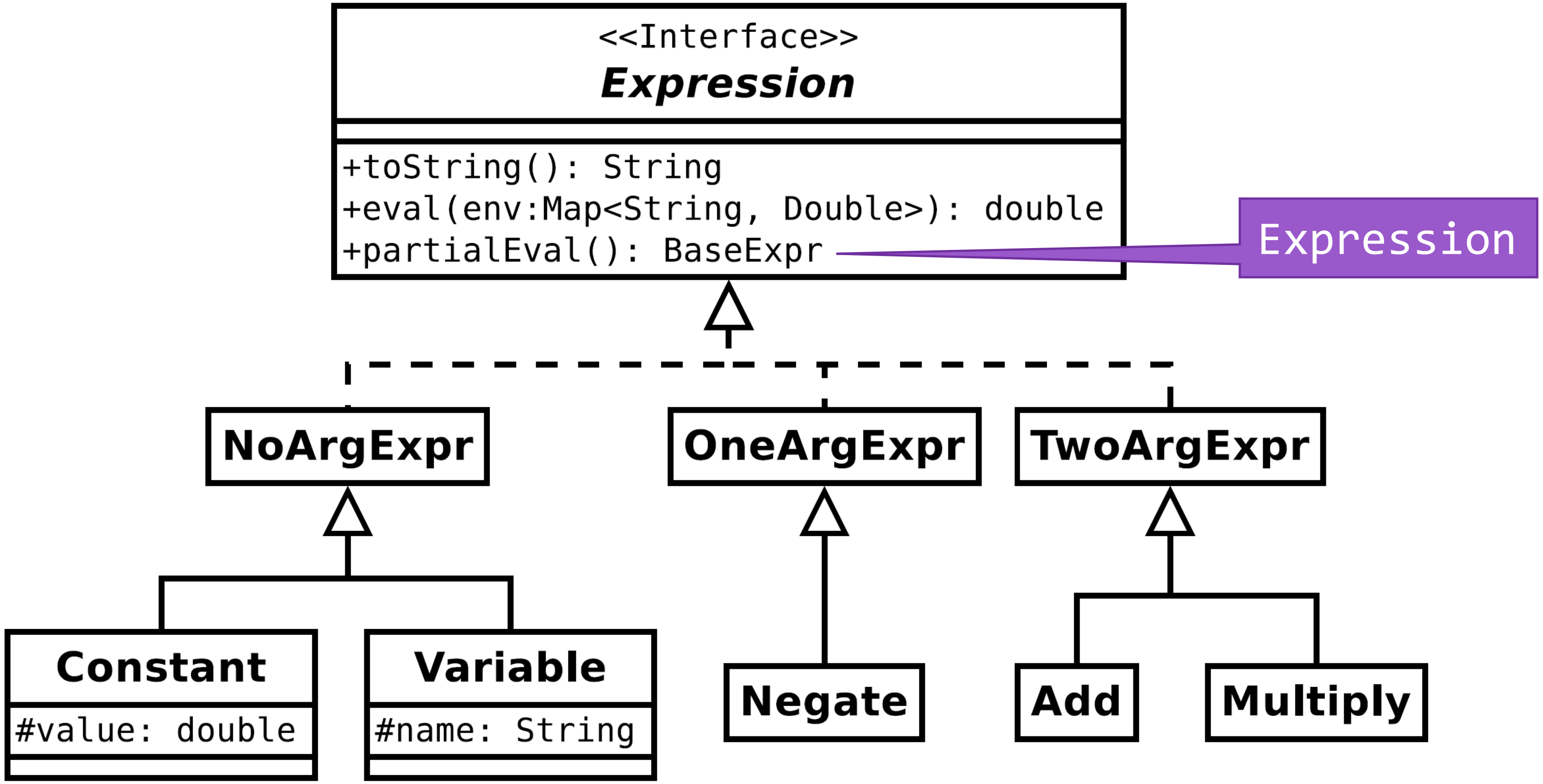
Datatype for expressions



operations:

- toString: `((2 + 4) * 5 + x)`
- evaluate: with x $\mapsto$ 12 this yields 42
- optimise (evaluate parts that do not contain variables)

Types for expressions



evaluation

- To evaluate an expression we need an environment assigning values to variables.
- The `eval` method has the environment as a parameter.
- An environment can be implemented in Java using a Map:
  - Map is a generic interface:

Interface Map<K,V>

Type Parameters:

`K - the type of keys maintained by this map`

`V - the type of mapped values`

interface Map<K,V>

Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
void	clear() Removes all of the mappings from this map (optional operation).
default V	compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
default V	computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction) If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null.
default V	computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
boolean	containsKey(Object key) Returns true if this map contains a mapping for the specified key.
boolean	containsValue(Object value) Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	entrySet() Returns a Set view of the mappings contained in this map.
boolean	equals(Object o) Compares the specified object with this map for equality.
default void	forEach(BiConsumer<? super K,? super V> action) Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
V	get(Object key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
default V	getOrDefault(Object key, V defaultValue) Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.
int	hashCode() Returns the hash code value for this map.
boolean	isEmpty() Returns true if this map contains no key-value mappings.
Set<K>	keySet() Returns a Set view of the keys contained in this map.
default V	merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction) If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
V	put(K key, V value) Associates the specified value with the specified key in this map (optional operation).
void	putAll(Map<? extends K,? extends V> m) Copies all of the mappings from the specified map to this map (optional operation).
default V	putIfAbsent(K key, V value) If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.
V	remove(Object key) Removes the mapping for a key from this map if it is present (optional operation).
default boolean	remove(Object key, Object value) Removes the entry for the specified key only if it is currently mapped to the specified value.
default V	replace(K key, V value) Replaces the entry for the specified key only if it is currently mapped to some value.
default boolean	replace(K key, V oldValue, V newValue) Replaces the entry for the specified key only if currently mapped to the specified value.
default void	replaceAll(BiFunction<? super K,? super V,? extends V> function) Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
int	size() Returns the number of key-value mappings in this map.
Collection<V>	values() Returns a Collection view of the values contained in this map.

Java map

- ```
  public interface Map<K,V> {
    V get( Object key );
    V put( K key, V value );
  }
  ```
- a commonly used implementation
  ```
  public class HashMap<K,V> implements Map<K,V> {
    …
  }
  ```
- Example
  ```
  Map<String, Double> env = new HashMap<>();
  env.put("x", 7.0);
  env.put("a", 42.0);
  double a = env.get("a");
  ```

Double is a class (reference type)
double is a primitive type
  ```

`double` vs. `Double`

- **`double`** `d = 3.14;`
- sometimes we need an object instead of a primitive value

  e.g. in the Map from names to values
- Java provides standard boxed variants of primitive (unboxed) types
- `Double db = new Double( 2.78 );`
- class `Double` contains also convenience methods (parsing, ..)
- instances of Double are immutable (like strings).
- Java performs *auto boxing* and *auto unboxing*: a silent conversion between `double` and `Double`

```
Double db = 2.78; // auto boxing
db = db * 2.0;    // which is equivalent to:
db = new Double( db.doubleValue() * 2.0 );
```

partial evaluation (optimization)

- evaluate the parts of an expression that do not depend on the values of variables
- rules:

$$
\begin{array}{rcl}
n+m & \rightarrow & o \text{ where } o = n+m \\
e+0 & \rightarrow & e \\
0+e & \rightarrow & e \\
n\times m & \rightarrow & o \text{ where } o = n\times m \\
0\times e & \rightarrow & 0 \\
1\times e & \rightarrow & e \\
e\times 0 & \rightarrow & 0 \\
e\times 1 & \rightarrow & e \\
\texttt{neg}(\, n\,) & \rightarrow & -n
\end{array}
$$

n, *m*: numerical constants
e: an arbitrary expression

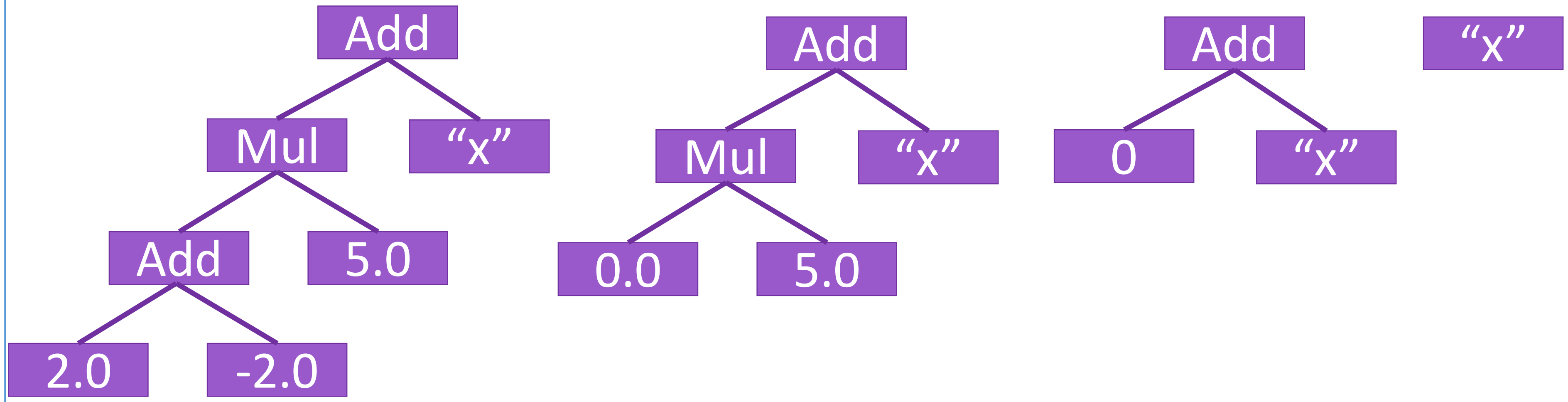
implementing partial evaluation

- `Expression` has a method partialEval for partial evaluation
  - arguments? — none
  - result? — optimized expression
  - abstract? — no, use **`return this;`** as default

- override this method to apply the rules (if this is possible)
  - how do we check if an argument is some constant?
  - polymorphism?
  - special method that yields value if it is there! what is the result if there is no value?

bottom-up or top-down?

add(n, m)	$\rightarrow n+m$
add(e, 0)	$\rightarrow$ e
add(0, e)	$\rightarrow$ e
mul(n, m)	$\rightarrow n\times m$
mul(0, e)	$\rightarrow$ 0
mul(1, e)	$\rightarrow$ e
mul(e, 0)	$\rightarrow$ 0
mul(e, 1)	$\rightarrow$ e
neg(n)	$\rightarrow -n$



- evalutate arguments first (bottom-up)!

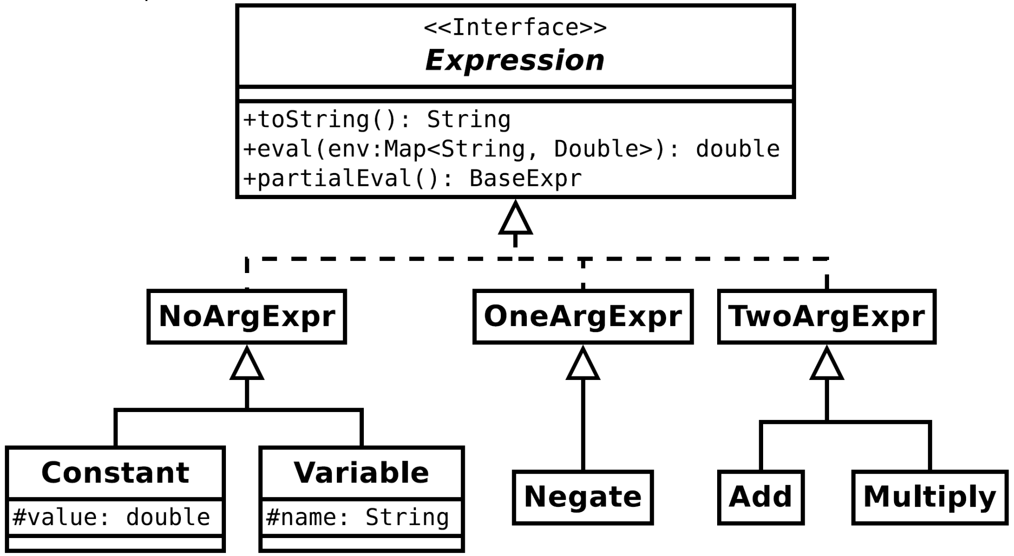
Representing binary expressions

```
public abstract class TwoArgExpr implements Expression {
  private final Expression x, y;

  public TwoArgExpr( Expression x, Expression y ) {
    this.x = x;
    this.y = y;
  }

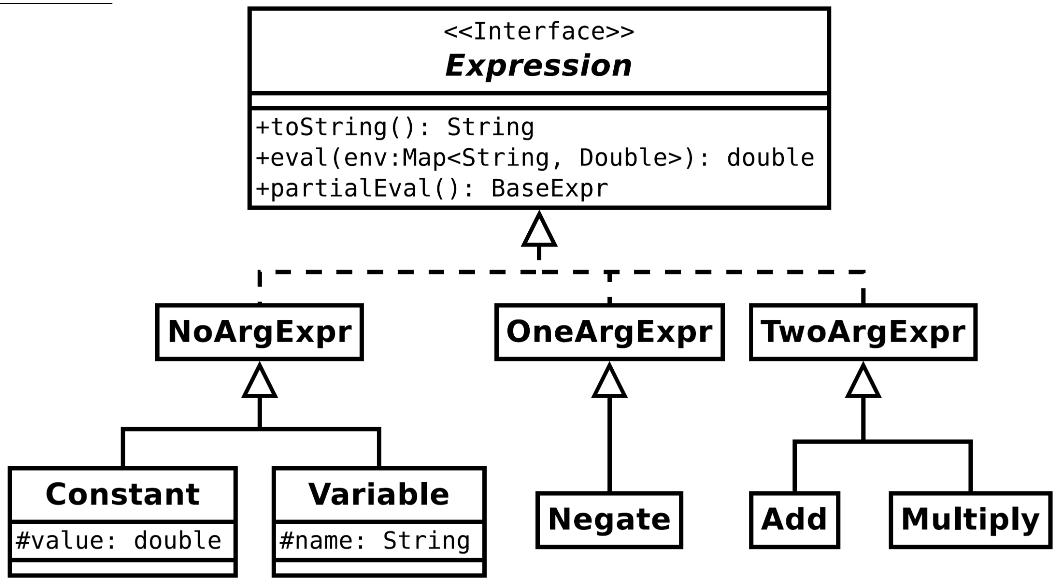
  public Expression getX() {
      return x;
  }

  public Expression getY() {
      return y;
  }
}
```



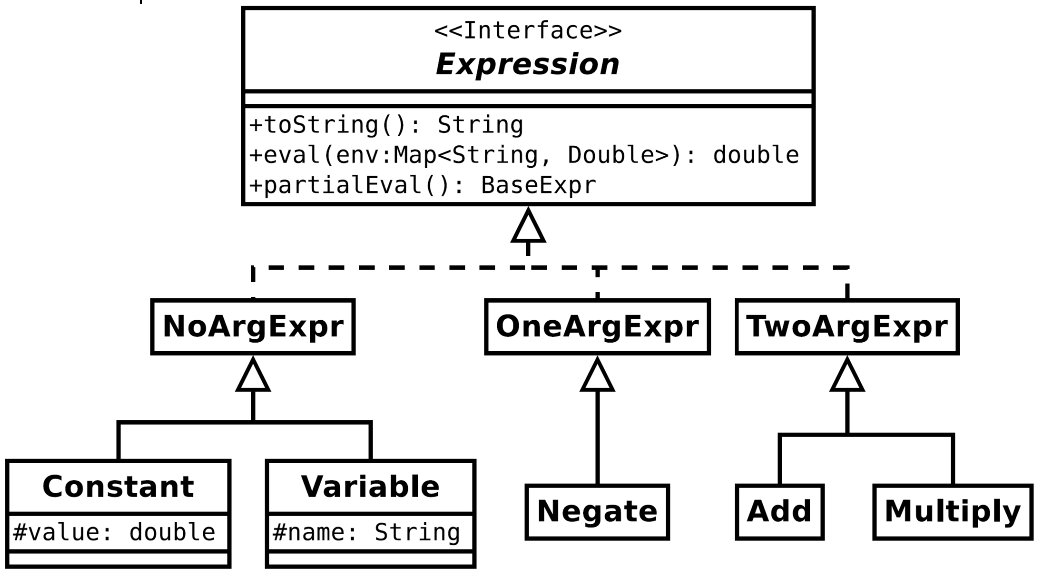
Binary expressions: Add

```java
public class Add extends TwoArgExpr {
  public Add( Expression x, Expression y ) {
    super( x, y );
  }

  @Override
  public String toString() {
    return "(" + getX() + "+" + getY() + ")";
  }

  @Override
  public double eval( Map<String, Double> env ) {
    return getX().eval(env) + getY().eval(env);
  }

  @Override
  public Expression partialEval() { … }
}
```

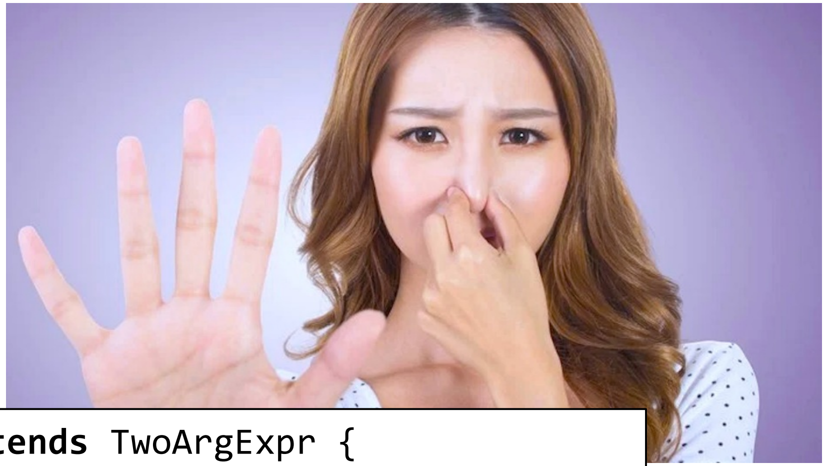


Binary expressions: Mul

```java
public class Mul extends TwoArgExpr {
  public Mul ( Expression x, Expression y ) {
    super ( x, y);
  }

  @Override
  public String toString() {
    return "(" + getX() + "*" + getY() + ")";
  }

  @Override
  public double eval( Map<String, Double> env ) {
    return getX().eval( env ) * getY().eval( env );
  }

  @Override
  public Expression partialEval() { … }
}
```



Code duplication:



```java
public class Add extends TwoArgExpr {
  public Add( Expression x, Expression y ) {
    super ( x, y );
  }

  @Override
  public String toString() {
    return "(" + getX() + "+" + getY() + ")";
  }

  @Override
  public double eval( Map<String, Double> env ) {
    return getX().eval(env) + getY().eval(env);
  }

  @Override
  public Expression partialEval() { … }
}
```

```java
public class Mul extends TwoArgExpr {
  public Mul ( Expression x, Expression y ) {
    super ( x, y );
  }

  @Override
  public String toString() {
    return "(" + getX() + "*" + getY() + ")";
  }

  @Override
  public double eval( Map<String, Double> env ) {
    return getX().eval(env) * getY().eval(env);
  }

  @Override
  public Expression partialEval() { … }
}
```

Solution: Pull the method up

Move duplicate code to the super class

- Use the *strategy pattern* to implement different behaviour

predefined generic interface
BinaryOperator<T>

T apply(T x,T y)

```
public abstract class TwoArgExpr implements Expression {
  private final Expression x, y;
  private final BinaryOperator<Double> evalOp;
  private final BinaryOperator<Expression> peOp;

  public TwoArgExpr( Expression x,Expression y,BinaryOperator<Double> evalOp,BinaryOperator<Expression> peOp ) {…}
  @Override
  public double eval( Map<String, Double> env ) {
      return evalOp.apply( x.eval(env), y.eval(env) );
  }

  @Override
  public Expression partialEval() {
      return peOp.apply( x.partialEval(), y.partialEval() );
  }

  @Override
  public String toString() {
      return "(" + x + getOperator() + y + ")";
  }

  protected abstract String getOperator();
}
```

Binary expressions: Add revisited

```
public class Add extends TwoArgExpr {
  public Add( Expression x, Expression y ) {
    super( x, y );
  }

  @Override
  public String toString() {
    return "(" + getX() + "+" + getY() + ")";
  }

  @Override
  public double eval( Map<String, Double> env ) {
    return getX().eval(env) + getY().eval(env);
  }

  @Override
  public Expression partialEval() { … }
}
```

```
public class Add extends TwoArgExpr {
  public Add( Expression x, Expression y ) {
    super( x, y, ???, ??? );
  }

  private Expression partialEvalAdd(…) { … }
}
```

Functional interfaces

- A functional interface is an interface with a *single abstract method*
  - It can be implemented by a lambda expression (more about lambda expressions next week)
- Instead of

```
public class AddEvaluator implements BinaryOperator<Double> {
    @Override
    public Double apply( Double t, Double u ) {
        return t + u;
    }
}

public Add( Expression x, Expression y ) {
     super ( x, y, new AddEvaluator (), … ),
}
```

Constructor of the class Add

Constructor of TwoArgExpr

lambda expression

- we can write (omitting the definition of class `AddEvaluator`)

```
public Add( Expression x, Expression y ) {
     super ( x, y, ( a1, a2 ) -> a1 + a2, … ),
}
```

Binary expressions: Add

```
public class Add extends TwoArgExpr {
    public Add( Expression x, Expression y ) {
        super ( x, y,
                ( a1, a2 ) -> a1 + a2,
                ( p1, p2 ) -> partialEvalAdd( p1, p2 ));
    }

    private static Expression partialEvalAdd( Expression x, Expression y ) {
        …
    }

    @Override
    protected final String getOperator() {
        return "+";
    }

}
```

Java peculiarities: static imports

- in Java we import a class as

```
import java.util.Scanner;
import java.lang.Math;
```

- sometimes it is boring/annoying to write class names
  e.g. `Math.PI` or `Math.sin`
- this can be prevented by static imports

```
import static java.lang.Math.PI;
```

- allows us to write `PI` instead of `Math.PI`
- static import of *all* public methods and attributes from Math:

```
import static java.lang.Math.*;
```

Expression factory

- To create an expression we have to write

```
Expression e1 = new Add( new Multiply( new Constant(2), new Constant(3) ),
                         new Variable("x"));
```

- hampers readability
- standard Java solution (design pattern): a *factory method*:

```
public static Add add( Expression x, Expression y ) {
   return new Add(x, y);
}
```

- Now we can use:

```
Expression e1 = add( mul( const(2), const(3) ), var( "x" ) );
```

typically we introduce a **factory class** containing this kind of methods

Finally