Essence of Object-Orientation

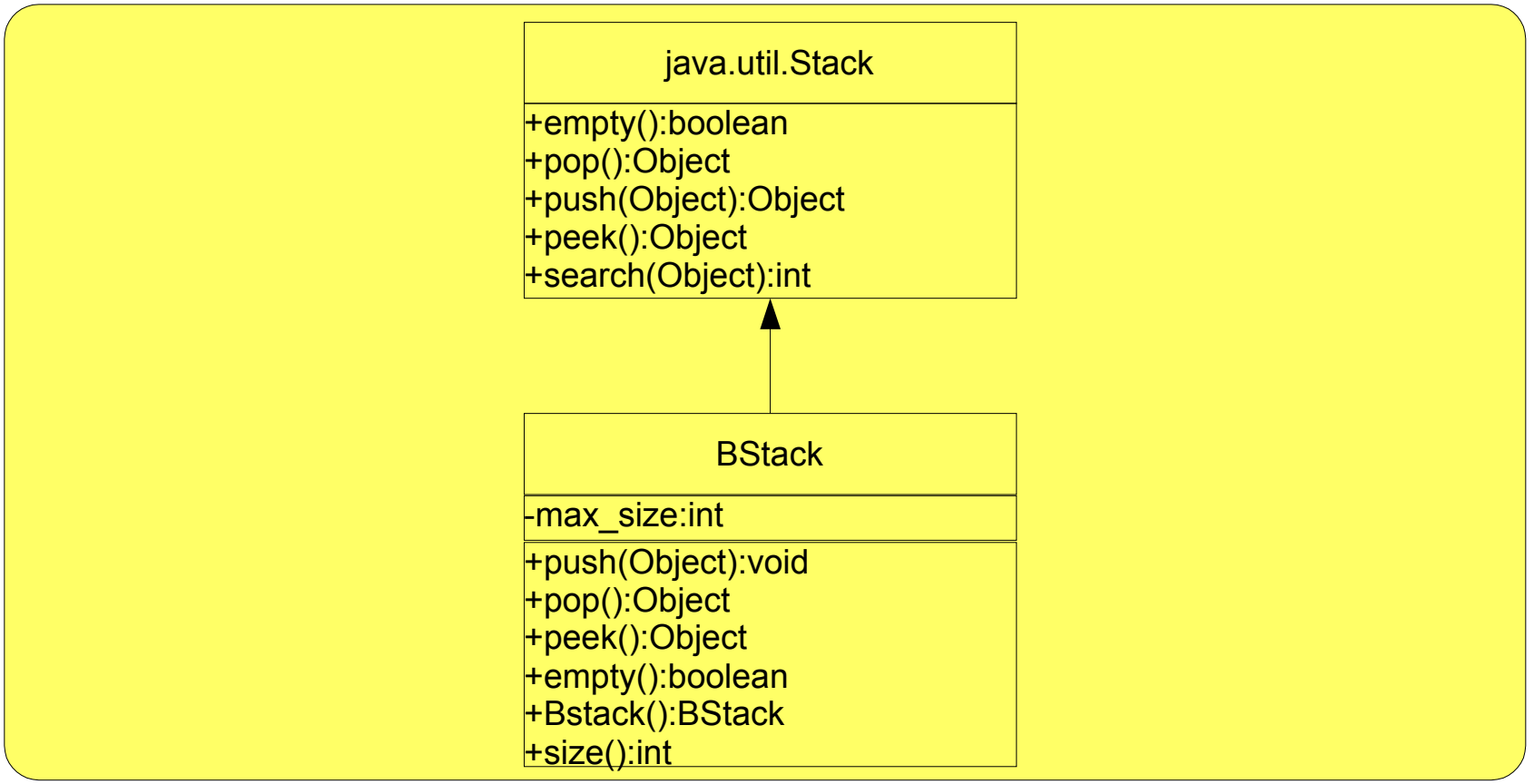
- Encapsulate data and its related operations as objects
  - members and methods
- Protection of individual objects against each other
  - externally (to the system) available operations and data
  - internally (to the object) available operations and data.

Inheritance: Code reuse or Behavioral specialization?

- What does it mean to extend a class through inheritance
  - simply reuse code?
  - specialize the behavior of the extended class?
  - both?
- Can we capture in "the language" all of the above meanings
  - not design rules!

What does Java do?

- Implement a bounded stack
  - There is a `Stack` implementation *java.util.Stack*
- Extend that one, half the code is already there!



What does Java do?

- Now `BStack` is a subtype of `Stack` !
- Can we freely substitute instances of `Stack` with instances of `BStack`?
  - Java compiler does not complain!
- Does it make sense to perform such a replacement?
- Inheritance between classes in Java denotes both code reuse **and** behavior specialization!

There are two things at play here

- The inheritance hierarchy denotes
  - classes that share code
  - classes that share behavior
- This treatment of the inheritance hierarchy is fixed in the language
  - e.g. `BStack` inherits code from `Stack` but is not a subtype of `Stack` cannot be expressed.
- Behavioral specialization is enforced by programmer
  - Documented (if at all) in the API.

Solutions ...

- Separate code inheritance from behavioral inheritance
  - POOL-I
  - allows the definition of 2 hierarchies, code inheritance and type inheritance
- Design By Contract
  - specification of behavior as assertions
  - runtime validation of assertions assists in inappropriate behavioral extensions

Design By Contract

- Assertions provide the assumptions and obligations of a method
  - *pre-condition*: Assumptions made on the the state of variables and/or object before execution can proceed to the method body
  - *post-condition*: Obligations expressed as conditions on variables, object and return value(s) the method should uphold to
- A method contract refers to the pre- and post-condition of a method's specification. (Eiffel 88)

Example

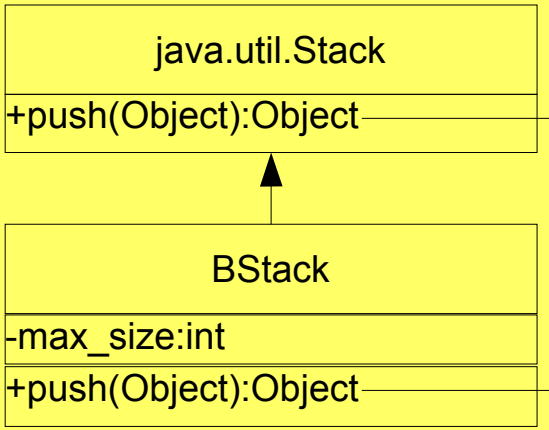
```
class Stack{
  ...
public void push(Object ele){
    @pre{true}(1)
  (3)@post{peek().equals(ele)}
    // method body (2)
}...
}

class Main{
  ...
public static void main(String args){
    // create an instance of Stack
    s.push(new Integer(8));
```

if false blame the caller (Main)

if false blame the method (push)

Contract Checking, Subtypes and Blame Assignment




```
@pre{true}
@post{peek().equals(ele)}
```

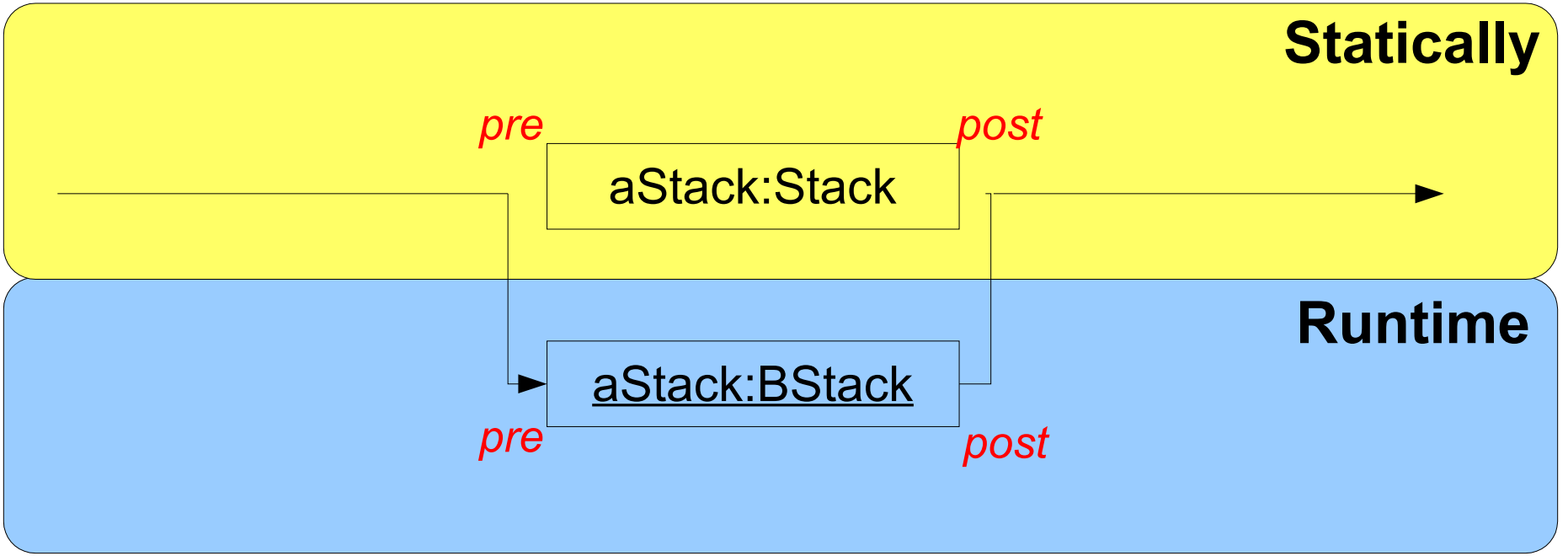
```
@pre{size()<max_size}
@post{peek().equals(ele)}
```

```
class Main{
  ...
public static void main(String args){
    Stack s = new BStack(1);
    s.push(new Integer(8));
    s.push(new Integer(9));
```

Which contract should be checked?

Who gets blamed?

Under the hood ...



Walkthrough (calling a method)

```
Stack s = new BStack(1);
s.push(new Integer(8));
s.push(new Integer(9));
```

(1) Stack pre

- its is being used as a Stack
- blame caller (main)

(2) Stack pre => BStack pre

- since BStack can appear in all places where Stack can it should be able to deal with the same input as Stack
- Blame **implementor** of BStack not a proper subtype!

Walkthrough (returning from a method)

```
Stack s = new BStack(1);
s.push(new Integer(8));
s.push(new Integer(9));
```

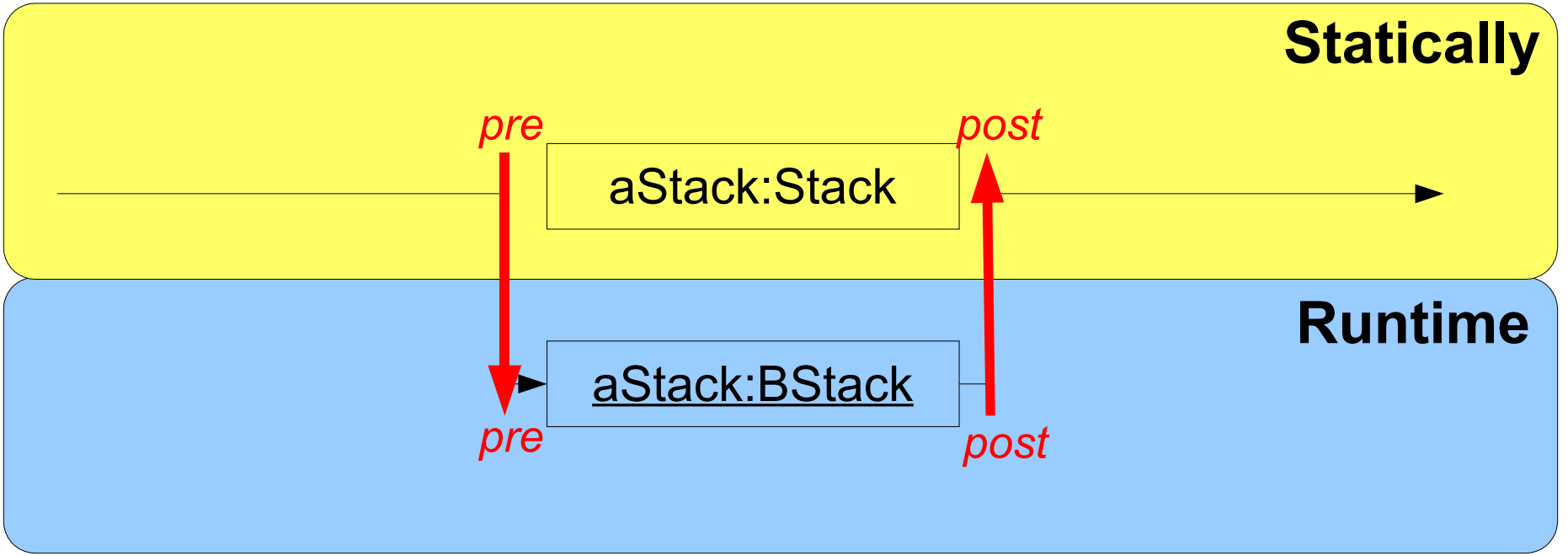
(1) BStack post

- Bstack's code is to execute, verify stated obligations
- blame implementation of BStack push

(2) BStack post => Stack post

- BStack can appear in all places where Stack can, return values have to also satisfy Stack's post
- Blame **implementor** of BStack not a proper subtype!

Under the hood ...



Contract Checking in OO

- Method pre-condition
  - blame caller
- Supertypes pre-condition implies subtypes pre-condition
  - blame subtype-not proper behavioral subtype
- Method post-condition
  - blame method implementation
- Subtypes post-condition implies supertypes post-condition
  - blame subtype-not proper behavioral subtype

Bringing Aspects into the picture

- Consider aspects in the AspectJ Language
- There are no types for aspects
  - as in the case of Classes in Java
- Aspects can
  - add new behavior
  - extend existing behavior
  - replace existing behavior
- How do you ensure that your aspects “play nice” with the existing program?

Using Aspects to extend behavior

- Base programmer does not control attachments of aspects (oblivious)
  - tracing, profiling etc.
- Addition of these aspects should **not** break the original system's assumptions
  - aspects should behave!
- Introduce “Behavioral Aspects”
  - extension of the ideas from behavioral subtyping

Example(1)

```
interface IBag{
public void add(int x);
  @post{\old(size())+1 == size()
public int remove();
  @pre{size() > 0}
  @post{\old(size())-1 == size()}
public int peek(int i);
  @pre{size()>i && i >= 0}
  @post{\old(size()) == size()}
public int size();
  @post{\result >= 0}}
```

```
aspect Default{
  pointcut rem(IBag o):call(* remove(..))&& target(o);

int around(IBag o):rem(o){
     if(o.size()==0) return 0;
     else return proceed(o);}}
```

Example(1)

```
class Test{
      IBag bag = ...;
  ...
public  void empty(){
      while(bag.size() > 0)
            bag.remove();
}
```

Compare the behavior observed when calling empty with and without the aspect Default attached.

```
aspect Default{
  pointcut rem(IBag o):call(* remove(..))&& target(o);

int around(IBag o):rem(o){
     if(o.size()==0) return 0;
     else return proceed(o);}}
```

Example(2)

```
interface IBag{
public void add(int x);
  @post{\old(size())+1 == size()
public int remove();
  @pre{size() > 0}
  @post{\old(size())-1 == size()}
public int peek(int i);
  @pre{size()>i && i >= 0}
  @post{\old(size()) == size()}
public int size();
  @post{\result >= 0}}
```

```
aspect Clever{
  int index = 0 ;
  pointcut rem(IBag o):call(* remove(..))&& target(o);

int around(IBag o):rem(o){
     int ret = o.peek(index);
     index = (index+1)% o.size();
     return ret;}}
```

Example(2)

```
class Test{
      IBag bag = ...;
  ...
public void empty(){
      while(bag.size() > 0)
            bag.remove();
}
```

Does not Terminate!

Compare the behavior observed when calling empty with and without the aspect Clever attached.

```
aspect Clever{
  int index = 0 ;
  pointcut rem(IBag o):call(* remove(..))&& target(o);

int around(IBag o):rem(o){
     int ret = o.peek(index);
     index = (index+1)% o.size();
     return ret;}}
```

Observations ...

- Programmers make mistakes
  - with the power provided by AOP its even easier
- How easy was it to find the bug in the example?
  - this was made up, simple and it fits on 4 slides
- AOP development tools help
  - but they do not help in reasoning about programs
  - visual help is good, but not good enough
- Contracts on aspects help in stating and enforcing the assumptions and obligations of aspect code.

Checking aspect contracts

- Aspect code extends behavior
- Should maintain the property

Added behavior via aspects does not break the original system's behavior but only specializes it.

- The same principle as with proper behavioral subtype
  - only here there is no static check, everything is done at runtime

Checking aspect contracts

- At a join point
  - the join points (method call or execution) pre-condition implies the advice pre-condition
  - the advice post-condition implies the join points (method call or execution) post-condition
- There is no explicit caller for advice
  - its implicit when the pointcut descriptor matches.
- In the case of multiple advice on the same join point
  - Predecessor is also “caller” of current advice.

Blame assignment for around advice

- Pre-conditions

Method Pre \ Advice Pre	True	False
True	✔	✗ Bad Extension
False	✔	✗ PCD

Blame assignment for around advice

- Post-conditions

Advice Post \ Method Post	True	False
True	✔	✗ Bad Extension
False	✔	✗ PCD

Example(1) Revisited

```
interface IBag{
public void add(int x);
  @post{\old(size())+1 == size()
public int remove();
  @pre{size() > 0}
  @post{\old(size())-1 == size()}
public int peek(int i);
  @pre{size()>i && i >= 0}
  @post{\old(size()) == size()}
public int size();
  @post{\result >= 0}}
```

```
aspect Default{
  pointcut rem(IBag o):call(* remove(..))&& target(o);
@pre{true}
@post{\old(\target(size()))-1 == \target(size()) ||
      \old(\target(size())) == 0}
int around(IBag o):rem(o){
     if(o.size()==0) return 0;
     else return proceed(o);}}
```

Example(2) Revisited

```
interface IBag{
public void add(int x);
  @post{\old(size())+1 == size()
public int remove();
  @pre{size() > 0}
  @post{\old(size())-1 == size()}
public int peek(int i);
  @pre{size()>i && i >= 0}
  @post{\old(size()) == size()}
public int size();
  @post{\result >= 0}}
```

```
aspect Clever{
  int index = 0 ;
  pointcut rem(IBag o):call(* remove(..))&& target(o);
@pre{true}
@post{\old(\target(size()))== \target(size())}
int around(IBag o):rem(o){
     int ret = o.peek(index);
     index = (index+1)% o.size();
     return ret;}}
```

Example(2) Revisited

```
class Test{
      IBag bag = ...;
  ...
public void empty(){
      while(bag.size() > 0)
            bag.remove();
}
```

- During evaluation the following implication is validated

```
\old(\target(size())) == \target(size()) =>  \old(size())-1 == size()
```

- If LHS of => is TRUE then => is FALSE
  - error is signaled, `Clever` is a bad extension