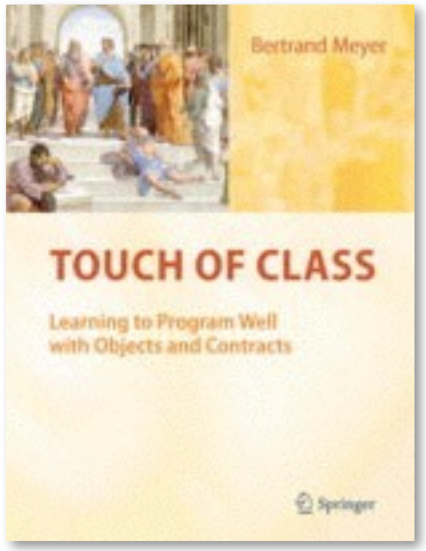
3. Design by Contract

Oscar Nierstrasz

Design by Contract

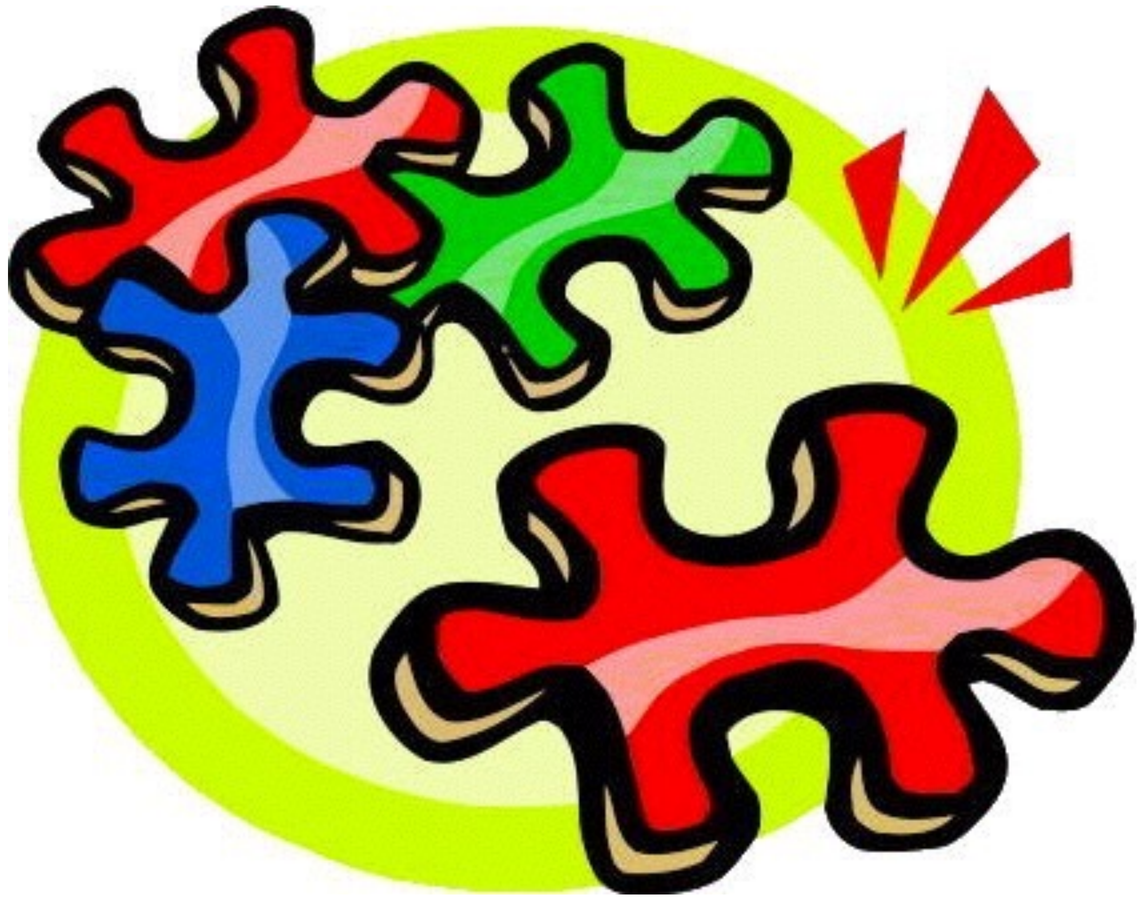
Bertrand Meyer, *Touch of Class — Learning to Program Well with Objects and Contracts*, Springer, 2009.



Bertrand Meyer is a French computer scientist who was a Professor at ETH Zürich (successor of Niklaus Wirth) from 2001-2015. He is best known as the inventor of “Design by Contract”, and as the designer of the Eiffel programming language, which provides built-in for DbC.

Who’s to blame?

The components fit but the system does not work. Who’s to blame? The component developer or the system integrator?



DbC makes clear the “contract” between a supplier (an object or “component”) and its client. When something goes wrong, the contract states whose fault it is. This simplifies both design and debugging.

Roadmap

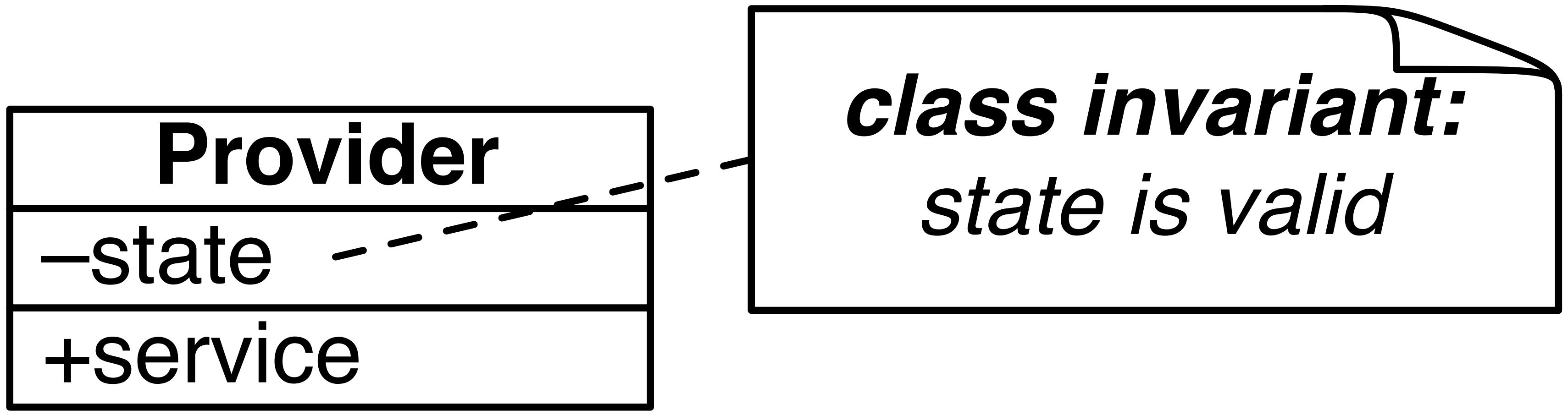
> Contracts
> Stacks
> Design by Contract
> A Stack Abstraction
> Assertions
> Example: balancing parentheses

Roadmap

> **Contracts**
> Stacks
> Design by Contract
> A Stack Abstraction
> Assertions
> Example: balancing parentheses

Class Invariants

An invariant is a predicate that *must hold* at certain points in the execution of a program



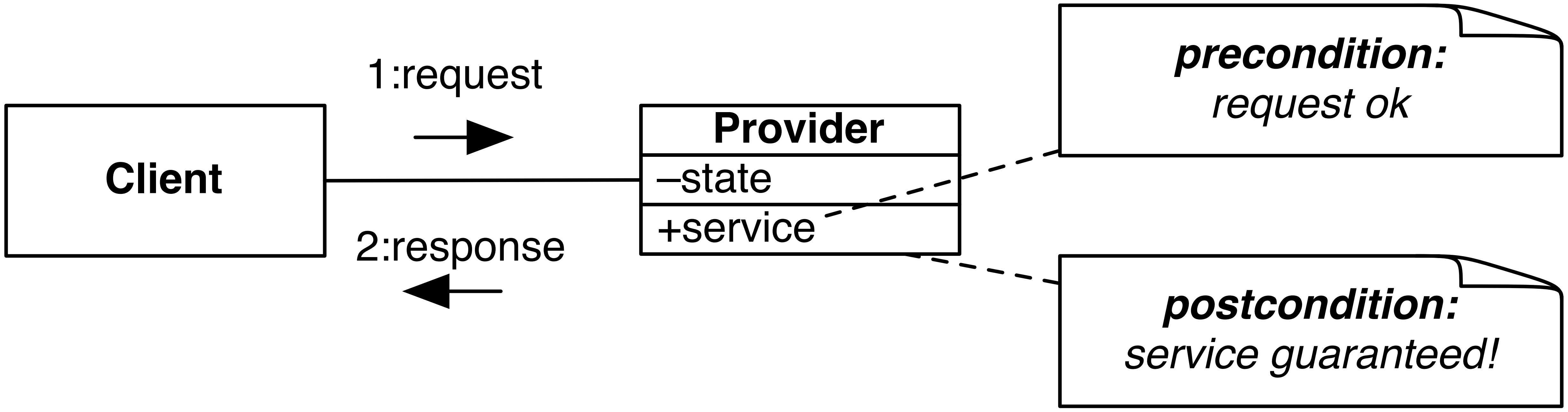
A class invariant characterizes the *valid states of instances*
It must hold:

1. *after construction*
2. *before and after every public method*

The word “invariant” means “never changing” — so, it refers to a predicate that is “always true” about instances of a class. In the case of DbC, “always” means “when it matters”, i.e., whenever a client could be affected. In practice this means *before or after any public methods*.

Contracts

A contract *binds the client* to pose valid requests, and *binds the provider* to correctly provide the service.

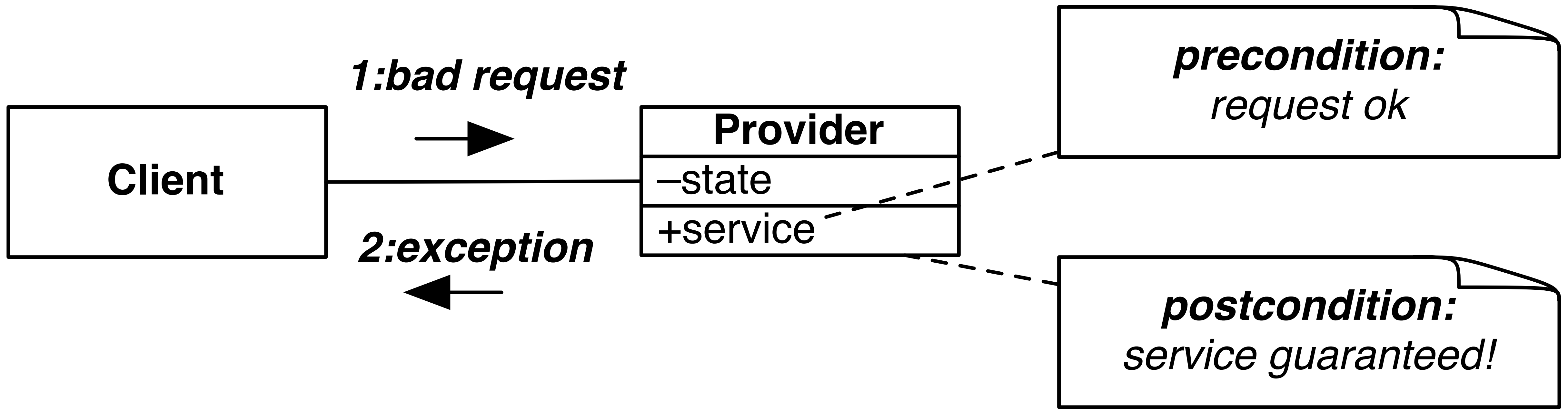


A contract formalizes what *preconditions* must hold before the client may request a service. If the preconditions do not hold, *then the client is at fault*, and the supplier is not required to do anything.

The contract further formalizes what *postconditions* must hold. If the preconditions hold, then the supplier is bound to fulfil the service, after which the postconditions should hold. If they do not, *then the supplier is at fault*.

Contract violations

If either the client or the provider violates the contract, an *exception* is raised.



NB: The service does not need to implement any special logic to handle errors — it simply raises an exception!

What to do if something goes wrong?

What you want to avoid is complicated logic to deal with every eventuality. Instead, specify clearly the contract, and *raise an exception* if something goes wrong. Handle the exception at the next highest level (typically abort or retry). More on this later.

Exceptions, failures and defects

> An exception is the occurrence of an *abnormal condition during the execution of a software element.*

> A failure is the *inability of a software element to satisfy its purpose*.

> A defect (AKA “bug”) is the *presence in the software of some element not satisfying its specification.*

Exceptions are a fundamental programming language mechanism to signal failure of a service. By specifying *exceptions* and *exception handlers*, we avoid the need to encode special error codes as return values from functions, and to check these values in client code. Exception handlers can be defined at an arbitrary level to simplify software design.

Note the distinction between what happens during execution (*exception*, *failure*) and what is in the source code (*defect*, *bug*).

Later in the context of unit testing, we will also talk about *failures* (of tests) and *errors* (unexpected exceptions).

Disciplined Exceptions

> There are only two reasonable ways to react to an exception:

1. clean up the environment and *report failure* to the client ("organized panic")
2. attempt to *change the conditions* that led to failure and *retry*

A failed assertion often indicates presence of a software defect, so "organized panic" is usually the best policy.

At the very least, an object that catches an exception should *reestablish its class invariant*, and then re-throw the exception.

In other words, “organized panic” means, “leave myself in a clean state and pass the buck.”

Usually with DbC, exceptions are raised in the preconditions, before any state is modified, so there is nothing to do to reestablish the class invariant. However it may be that some work has been done, and a service requested of another object fails. Then some cleanup may be necessary before throwing an exception.

Retrying only makes sense if there is a subsequent chance of success, e.g., trying a different algorithm, trying a fixed number of times to obtain a resource …

Roadmap

- Contracts
- **Stacks**
- Design by Contract
- A Stack Abstraction
- Assertions
- Example: balancing parentheses

Stacks

A Stack is a classical data abstraction with many applications in computer programming.

Stacks support two mutating methods: push and pop.

Operation	***Stack***	***isEmpty()***	***size()***	***top()***
		TRUE	0	(error)
push(6)	$\boxed{6}$	FALSE	1	6
push(7)	$\boxed{6}\boxed{7}$	FALSE	2	7
push(3)	$\boxed{6}\boxed{7}\boxed{3}$	FALSE	3	3
pop()	$\boxed{6}\boxed{7}$	FALSE	2	7
push(2)	$\boxed{6}\boxed{7}\boxed{2}$	FALSE	3	2
pop()	$\boxed{6}\boxed{7}$	FALSE	2	7

A stack holds a number of elements in the order in which they are *pushed* and returns them in reverse order through the *pop* operation. Stack behavior is known as *last in, first out (LIFO)*. (The last element pushed is the first one popped. Conversely, the first one pushed is the last one to be popped.)

Stacks are a fundamental construct in the implementation of programming languages: The *run-time stack* keeps track of operations that are executed. Every operation that is called pushes a *stack frame* onto the run-time stack holding its local variables, and this frame is popped when the operation returns, placings its callers frame on the top of the stack.

More on this in the lecture on Debugging.

Stack pre- and postconditions

Stacks should respect the following contract:

service	***pre***	***post***
`isEmpty()`	-	*no state change*
`size()`	-	*no state change*

`push(Object item)`	-	not empty,
		`size == old size + 1,`
		`top == item`

`top()`	not empty	*no state change*
`pop()`	not empty	`size == old size - 1`

Stack invariant

> The only thing we can say about the Stack class invariant is that the size is always ≥ 0
>
> —we don't know anything yet about its state!

Roadmap

- > Contracts
- > Stacks
- > **Design by Contract**
- > A Stack Abstraction
- > Assertions
- > Example: balancing parentheses

Design by Contract

When you design a class, each service S provided must specify a clear contract.

> *"If you promise to call S with the precondition satisfied, then I, in return, promise to deliver a final state in which the post-condition is satisfied."*

Consequence:

—if the precondition does not hold, *the object is not required to provide anything!* (in practice, an exception is raised)

In other words …

Design by Contract =

Don’t accept anybody else’s garbage!

DbC *simplifies design* by clearly separating responsibilities.

Pre- and Post-conditions

The pre-condition binds clients:

- —it defines what the data abstraction *requires* for a call to the operation to be legitimate
- —it may involve *initial state and arguments*
- —example: *stack is not empty*

The post-condition, in return, binds the provider:

- —it defines the conditions that the data abstraction *ensures* on return
- —it may only involve the *initial and final states, the arguments and the result*
- —example: *size = old size + 1*

Benefits and Obligations

A contract provides *benefits and obligations* for both clients and providers:

	Obligations	***Benefits***
Client	Only call pop() on a non-empty stack!	Stack size decreases by 1. Top element is removed.
Provider	Decrement the size. Remove the top element.	No need to handle case when stack is empty!

Who’s to blame?

If preconditions are violated, the client is to blame.
If invariants or postconditions fail, the component is to blame!

Roadmap

> Contracts
> Stacks
> Design by Contract
> **A Stack Abstraction**
> Assertions
> Example: balancing parentheses

StackInterface

Interfaces let us *abstract* from concrete implementations:

```
public interface StackInterface<E> {
    public boolean isEmpty();
    public int size();
    public void push(E item);
    public E top();
    public void pop();
}
```

✎ How can clients accept multiple implementations of a data abstraction?

✔ *Make them depend only on an interface or an abstract class.*

This interface is *generic*, that is it takes elements of an arbitrary type E. This is useful to avoid losing type information and cluttering your code with *downcasts* when you retrieve elements from the stack and want to use them.

Interfaces in Java

Interfaces *reduce coupling* between objects and their clients:

- A class can implement multiple interfaces
  - … but can only extend one parent class
- Clients should depend on an interface, not an implementation
  - … so implementations don’t need to extend a specific class

Define an interface for any data abstraction that will have more than one implementation

As a rule, you should *avoid depending on concrete classes* as much as possible in your code. Dependencies are evil and make your code fragile!

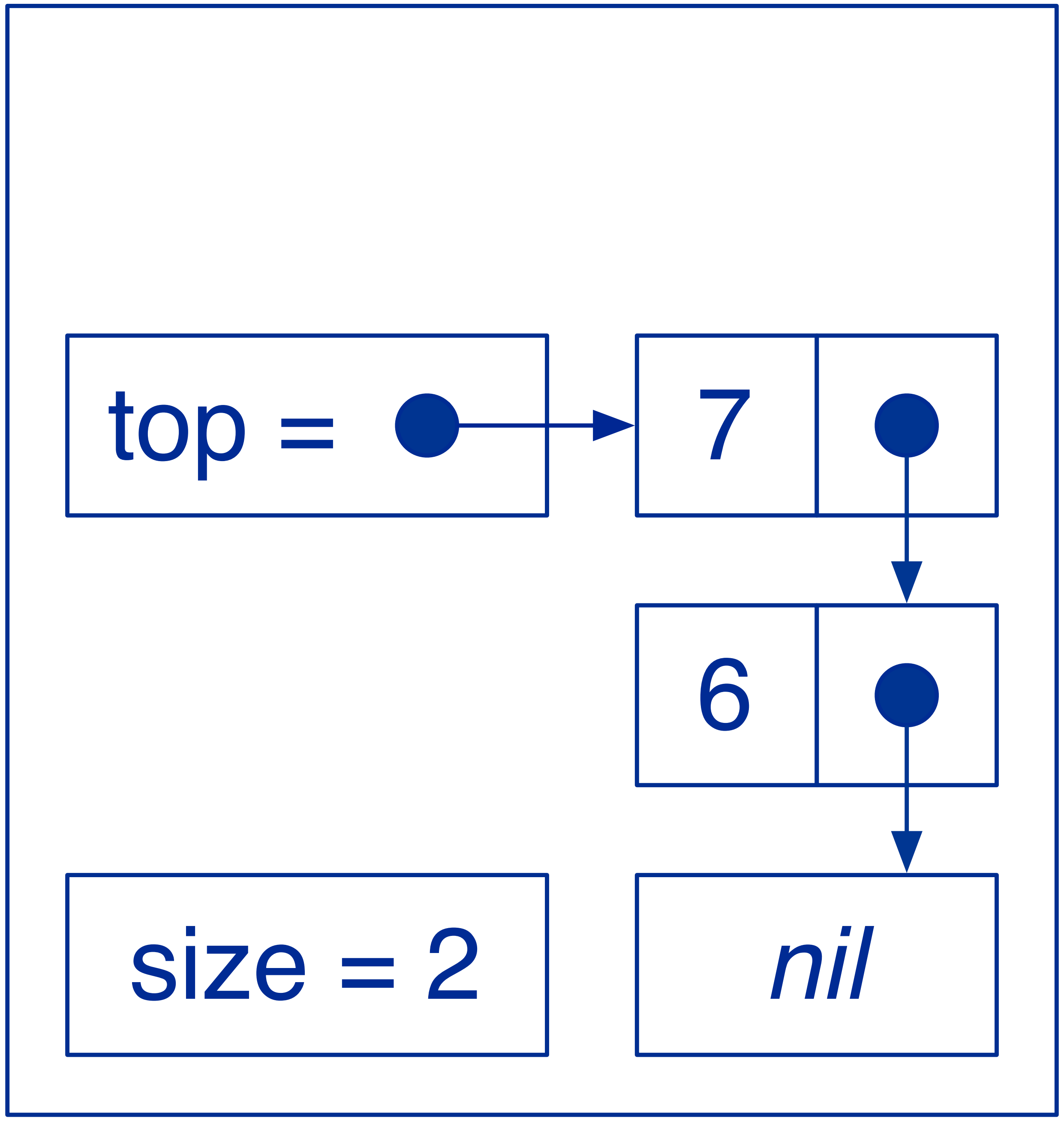
Instead, introduce interfaces wherever you expect objects of more than one possible class to be used. This will leave your design free to accommodate new classes in the future.

[On the other hand, if it is clear that only one class is ever needed, don’t clutter your design with unneeded interfaces.]

Stacks as Linked Lists

A Stack can easily be implemented by a linked data structure:

```
stack = new Stack();
stack.push(6);
stack.push(7);
stack.push(3);
stack.pop();
```



LinkStack Cells

We can define the Cells of the linked list as an *inner class* within `LinkStack`:

```
public class LinkStack<E> implements StackInterface<E> {
   private Cell top;
   private class Cell {
      E item;
      Cell next;
      Cell(E item, Cell next) {
         this.item = item;
         this.next = next;
      }
   }
   ...
}
```

Inner classes are useful when you have a complex object with a nested object that is very simple and will not be used anywhere else in your code.

Private vs Public instance variables

✎ When should instance variables be public?

✔ *Always make instance variables private or protected.*

The Cell class is a special case, since its instances are strictly private to LinkStack!

Note that Java has a fourth category of visibility: if you do not declare a method or a variable to be one of `public`, `protected` or `private`, then the default is *package scope*.

Features with package scope can be accessed by any code within the same package, but not outside. Package scope can be especially useful if you want features of a particular exposed to other related classes (e.g., test classes), but not to the world at large.

LinkStack abstraction

The constructor must construct a *valid initial state*:

```
public class LinkStack<E> implements StackInterface<E> {
   ...
   private int size;
   public LinkStack() {
      // Establishes the class invariant.
      top = null;
      size = 0;
   }
   ...
```

Class Invariants

A class invariant is any condition that expresses the *valid states* for objects of that class:

> it must be *established* by every constructor

> every public method

—may *assume* it holds when the method starts

—must *re-establish* it when it finishes

Stack instances must satisfy the following invariant:

> size ≥ 0

> ...

LinkStack Class Invariant

A valid `LinkStack` instance has an integer `size`, and a `top` that points to a sequence of linked `Cells`, such that:

- `size` is always ≥ 0
- When `size` is zero, `top` points nowhere (== `null`)
- When `size` > 0, `top` points to a `Cell` containing the top item

When to check invariants?

> In principle, check invariants:
>
> —at the end of each *constructor*
>
> —at the end of every *public mutator*

If a method does not change the state of the object, then there is no need to check the invariant at its completion.

Roadmap

- Contracts
- Stacks
- Design by Contract
- A Stack Abstraction
- **Assertions**
- Example: balancing parentheses

Assertions

> An assertion is a declaration of a *boolean expression* that the programmer believes *must hold* at some point in a program.
>
> —Assertions should not affect the logic of the program
>
> —If an assertion fails, an *exception* is raised

```
x = y*y;
assert x >= 0;
```

If an assertion is false, there is a bug in the program!

Assertions

Assertions have four principle applications:

1. Help in writing correct software
   — formalizing invariants, and pre- and post-conditions
2. Documentation aid
   — specifying contracts
3. Debugging tool
   — testing assertions at run-time
4. Support for software fault tolerance
   — detecting and handling failures at run-time

Assertions have applications beyond DbC, particularly in the context of proving that a non-trivial algorithm is correct.

Assertions can be used to document and check your assumptions at various points in the execution of your code. This is especially useful for debugging your code (you will get an exception at the point where your assertion fails, not later when the code breaks).

DbC is just a special case, where the assertions correspond to pre- and post-conditions, and invariants.

Assertions in Java

`assert` is a keyword in Java since version 1.4

```
assert expression;
```

will raise an `AssertionError` if *`expression`* is false.

—*NB:* Throwable *Exceptions* must be declared; *Errors* need not be!

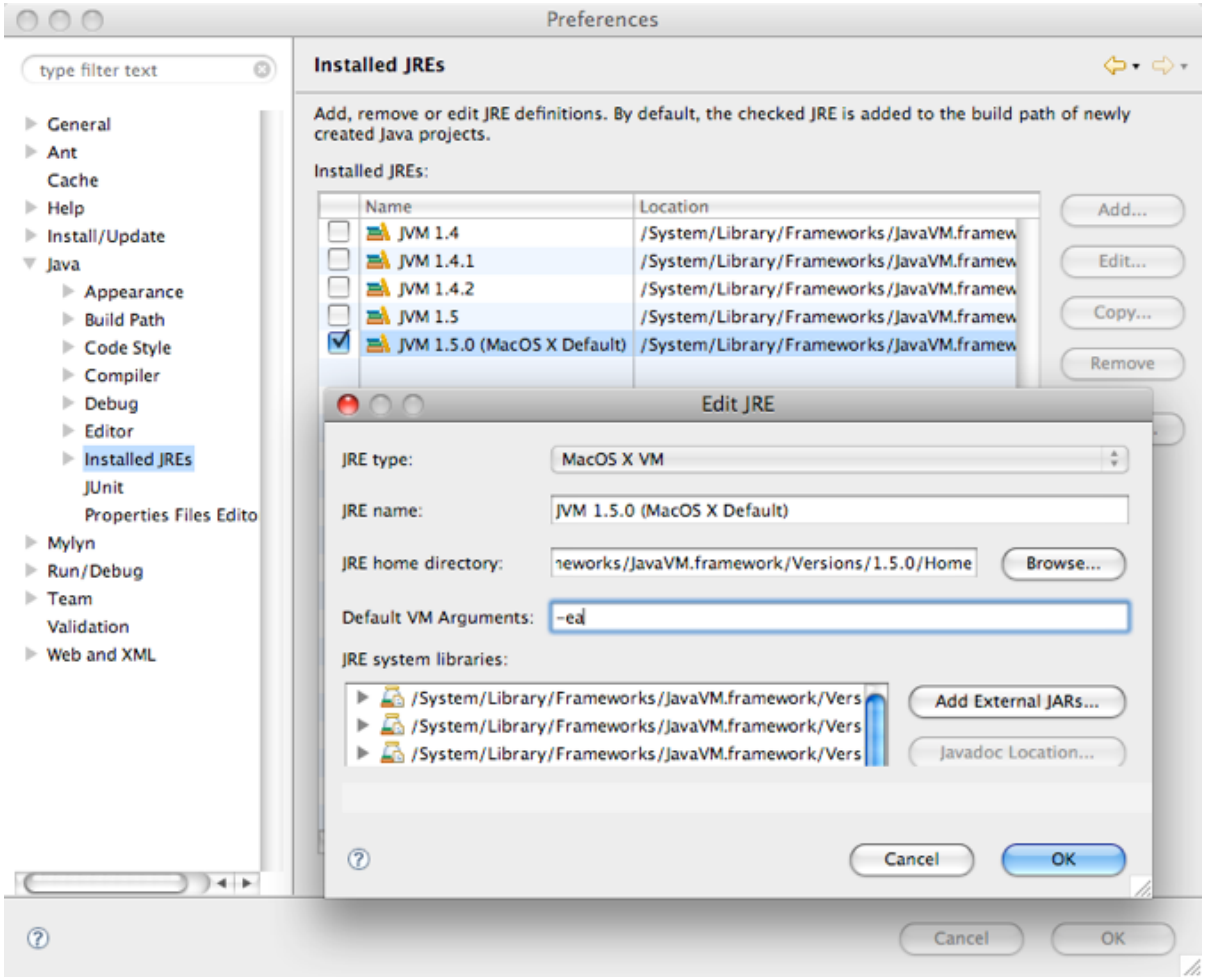
✔ *Be sure to enable exceptions in eclipse! (And set the vm flag -enableassertions [-ea])*

Note the relationship between assertions and exceptions.

An *exception* is a programming language mechanism to signal that the normal flow of control of the program cannot proceed because some *exceptional event* has taken place (e.g., bad input, unavailable resource etc.).

An *assertion* is a mechanism to state that a particular predicate should hold at a particular point in the code. If this predicate fails, this is an *exceptional event*, and an exception is raised.

Enabling assertions in eclipse



Preferences

Installed JREs

Add, remove or edit JRE definitions. By default, the checked JRE is added to the build path of newly created Java projects.

Installed JREs:

	Name	Location
☐	JVM 1.4	/System/Library/Frameworks/JavaVM.framew
☐	JVM 1.4.1	/System/Library/Frameworks/JavaVM.framew
☐	JVM 1.4.2	/System/Library/Frameworks/JavaVM.framew
☐	JVM 1.5	/System/Library/Frameworks/JavaVM.framew
✓	JVM 1.5.0 (MacOS X Default)	/System/Library/Frameworks/JavaVM.framew

Edit JRE

JRE type: MacOS X VM

JRE name: JVM 1.5.0 (MacOS X Default)

JRE home directory: ᴧeworks/JavaVM.framework/Versions/1.5.0/Home

Default VM Arguments: -ea

JRE system libraries:

/System/Library/Frameworks/JavaVM.framework/Vers
/System/Library/Frameworks/JavaVM.framework/Vers
/System/Library/Frameworks/JavaVM.framework/Vers

Checking pre-conditions

Assert pre-conditions to inform clients when *they* violate the contract.

NB: This is all you have to do!

```
public E top() {
    assert !this.isEmpty();     // pre-condition
    return top.item;
}
```

✎ When should you check pre-conditions to methods?

✔ *Always check pre-conditions, raising exceptions if they fail.*

Note how elegant this is – you do not need to write any special code to handle the error! DbC says that, when a pre-condition fails, *it's not your fault*, so you don't have to do anything special.

Since pre-conditions are checked before any other code is run, you are also sure that the state of the object has not changed, so you do not have to worry about cleaning up your state to reestablish the invariant.

NB: pre-conditions, post-conditions and invariants should never use code that modifies the state of your object.

Checking class invariants

Every class has its own invariant:

```
protected boolean invariant() {
   return (size >= 0) &&
      ( (size == 0 && this.top == null)
      || (size > 0 && this.top != null));
}
```

Why protected and not private?

Checking post-conditions

Assert post-conditions and invariants to inform yourself when *you* violate the contract.

```
public void push(E item)  {
    top = new Cell(item, top);
    size++;
    assert !this.isEmpty();            // post-condition
    assert this.top() == item;         // post-condition
    assert invariant();
}
```

NB: This is all you have to do!

✎ When should you check post-conditions?

✔ *Check them whenever the implementation is non-trivial.*

Debugging faulty code can be very hard, particularly if that code corrupts the state of the program but does not cause a the program to immediately fail. In this case the error will occur elsewhere in the program, when the corrupted data is used.

By explicitly stating and checking pre- and post-conditions, as well as invariants, you increase the chance that exceptions will be raised precisely where the fault code lies, and bugs can be eliminated much more efficiently.

Roadmap

> Contracts
> Stacks
> Design by Contract
> A Stack Abstraction
> Assertions
> **Example: balancing parentheses**

Example: Balancing Parentheses

Problem:

> Determine whether an expression containing parentheses ( ), brackets [ ] and braces { } is correctly balanced.

Examples:

> balanced:

```
if (a.b()) { c[d].e(); }
else { f[g][h].i(); }
```

> not balanced:

```
((a+b())
```

A simple algorithm

Approach:

> when you read a *left* parenthesis, *push the matching parenthesis* on a stack

> when you read a *right* parenthesis, *compare it* to the value on top of the stack

—if they *match*, you *pop and continue*

—if they *mismatch*, the expression is *not balanced*

> if the *stack is empty* at the end, the whole expression is *balanced*, otherwise not

Stacks are the perfect data abstraction for keeping track of your position while navigating through a nested hierarchy. As you navigate down, you push your location, and as you return upwards, you pop. This is exactly what happens in the run-time stack, which keeps track of the hierarchy of method calls.

In exactly the same way, a stack can keep track of our position in the tree of nested parentheses.

Using a Stack to match parentheses

Sample input: “([{ }]]”

Input	*Case*	*Op*	*Stack*
(	left	push)	)
[	left	push]	)]
{	left	push }	)]}
}	match	pop	)]
]	match	pop	)
]	mismatch	^false	)

The ParenMatch class

A `ParenMatch` object *uses a stack* to check if parentheses in a text String are balanced:

```
public class ParenMatch {
   private String line;
   private StackInterface<Character> stack;

   public ParenMatch(String aLine,
                     StackInterface<Character> aStack) {
   {
      line = aLine;
      stack = aStack;
   }
```

A declarative algorithm

We implement our algorithm at a high level of abstraction:

```
public boolean parenMatch() {
   for (int i=0; i<line.length(); i++) {
      char c = line.charAt(i);
      if (isLeftParen(c)) { // expect matching right paren later
         stack.push(matchingRightParen(c)); // Autoboxed to Character
      } else {
         if (isRightParen(c)) {
            // empty stack => missing left paren
            if (stack.isEmpty()) { return false; }
            if (stack.top().equals(c)) { // Autoboxed
               stack.pop();
            } else { return false; } // mismatched paren
         }
      }
   }
   return stack.isEmpty(); // not empty => missing right paren
}
```

“Declarative” here means we say *what we do*, rather than *how to do it*. At this level of abstraction we do not care how we check if `c` is a left parenthesis or is a matching right parenthesis, we only care that we need to do this. The details are available at the next level of abstraction down.

Well-written declarative code is *self-documenting*.

By writing a declarative algorithm we *enhance readability* and *program comprehension*, and thus we *improve maintainability* of our code.

Ugly, procedural version

```
public boolean parenMatch() {
   char[] chars = new char[1000]; // ugly magic number
   int pos = 0;
   for (int i=0; i<line.length(); i++) {
      char c = line.charAt(i);
      switch (c) { // what is going on here?
      case '{' : chars[pos++] = '}'; break;
      case '(' : chars[pos++] = ')'; break;
      case '[' : chars[pos++] = ']'; break;
      case ']' : case ')' : case '}' :
         if (pos == 0) { return false; }
         if (chars[pos-1] == c) { pos--; }
         else { return false; }
         break;
      default : break;
      }
   }
   return pos == 0; // what is this?
}
```

This code does exactly what the previous version did, but is not declarative. It requires extra effort to understand what is going on.

Helper methods

The helper methods are trivial to implement, and their details only get in the way of the main algorithm.

```
    private boolean isLeftParen(char c) {
       return (c == '(') || (c == '[') || (c == '{');
    }

    private boolean isRightParen(char c) {
       return (c == ')') || (c == ']') || (c == '}');
    }
```

By choose *intention-revealing names* for helper methods, we simultaneously promote self-documenting, declarative code in clients, and we also make the intent of the method body clear.

Running parenMatch

```
public static void parenTestLoop(StackInterface<Character> stack) {
    BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));
    String line;
    try {
        System.out.println("Please enter parenthesized expressions to test");
        System.out.println("(empty line to stop)");
        do {
            line = in.readLine();
            System.out.println(new ParenMatch(line, stack).reportMatch());
        } while(line != null && line.length() > 0);
        System.out.println("bye!");
    } catch (IOException err) {
    } catch (AssertionException err) {
        err.printStackTrace();
    }
}
```

Running ParenMatch.main ...

```
Please enter parenthesized expressions to test
(empty line to stop)
(hello) (world)
"(hello) (world)" is balanced
()
"()" is balanced
static public void main(String args[]) {
"static public void main(String args[]) {" is not balanced
()
"()" is not balanced
}
"}" is balanced

"" is balanced
bye!
```

Which contract has been violated?

We have been careful about specifying the contracts for the Stack abstraction, but we were sloppy with the ParenMatch class.

Which implicit contract have we failed to formalize and respect?

Which assertion(s) need to be added, and how do we correct the bug?

NB: There are several possible solutions …

What you should know!

- ✎ *What is an abstract data type?*
- ✎ *What is the difference between encapsulation and information hiding?*
- ✎ *How are contracts formalized by pre- and post-conditions?*
- ✎ *What is a class invariant and how can it be specified?*
- ✎ *What are assertions useful for?*
- ✎ *What situations may cause an exception to be raised?*
- ✎ *How can helper methods make an implementation more declarative?*

Can you answer these questions?

- ✎ *When should you call super() in a constructor?*
- ✎ *When should you use an inner class?*
- ✎ *What happens when you pop() an empty java.util.Stack? Is this good or bad?*
- ✎ *What impact do assertions have on performance?*
- ✎ *Can you implement the missing LinkStack methods?*