



<http://algs4.cs.princeton.edu>

1.3 BAGS, QUEUES, AND STACKS

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

Parameterized stack

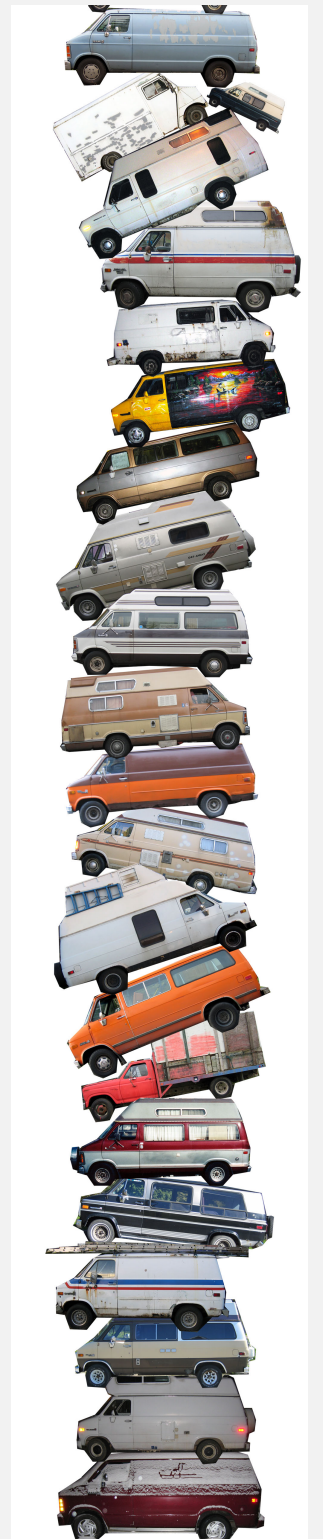
We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfInts`, `StackOfVans`,

Attempt 1. Implement a separate stack class for each type.

- Rewriting code is tedious and error-prone.
- Maintaining cut-and-pasted code is tedious and error-prone.

@#*\$! most reasonable approach until Java 1.5.



Parameterized stack

We implemented: StackOfStrings.

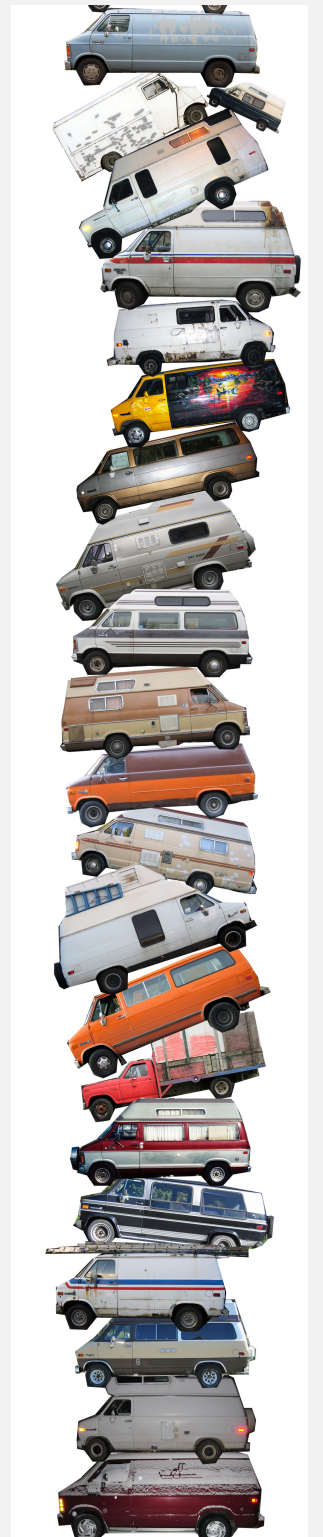
We also want: StackOfURLs, StackOfInts, StackOfVans,

Attempt 2. Implement a stack with items of type Object.

- Casting is required in client.
- Casting is error-prone: run-time error if types mismatch.

```
StackOfObjects s = new StackOfObjects();  
Apple a = new Apple();  
Orange b = new Orange();  
s.push(a);  
s.push(b);  
a = (Apple) (s.pop());
```

run-time error



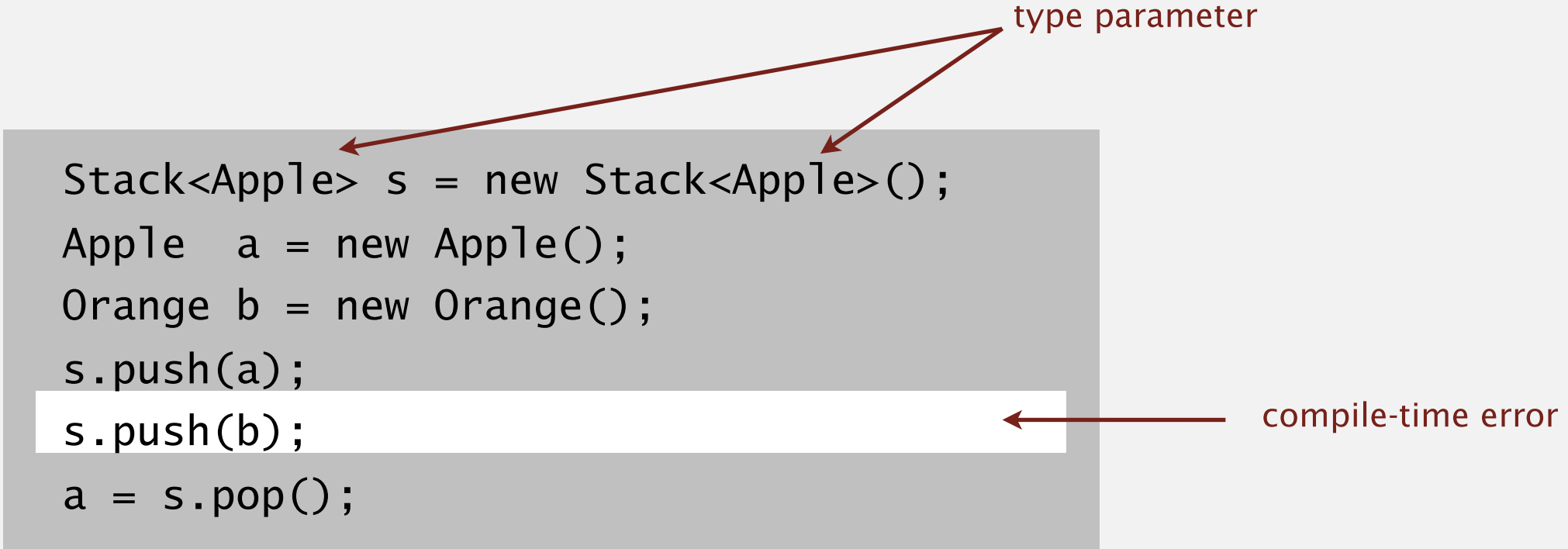
Parameterized stack

We implemented: StackOfStrings.

We also want: StackOfURLs, StackOfInts, StackOfVans,

Attempt 3. Java generics.

- Avoid casting in client.
- Discover type mismatch errors at compile-time instead of run-time.



```
Stack<Apple> s = new Stack<Apple>();
Apple a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = s.pop();
```

Guiding principles. Welcome compile-time errors; avoid run-time errors.

Generic stack: linked-list implementation

```
public class LinkedStackOfStrings
{
    private Node first = null;

    private class Node
    {
        String item;
        Node next;
    }

    public boolean isEmpty()
    { return first == null; }

    public void push(String item)
    {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public String pop()
    {
        String item = first.item;
        first = first.next;
        return item;
    }
}
```

```
public class Stack<Item>
{
    private Node first = null;

    private class Node
    {
        Item item;
        Node next;
    }

    public boolean isEmpty()
    { return first == null; }

    public void push(Item item)
    {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public Item pop()
    {
        Item item = first.item;
        first = first.next;
        return item;
    }
}
```

generic type name

Generic stack: array implementation

the way it should be

```
public class FixedCapacityStackOfStrings
{
    private String[] s;
    private int N = 0;

    public ..StackOfStrings(int capacity)
    { s = new String[capacity]; }

    public boolean isEmpty()
    { return N == 0; }

    public void push(String item)
    { s[N++] = item; }

    public String pop()
    { return s[--N]; }
}
```

```
public class FixedCapacityStack<Item>
{
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity)
    { s = new Item[capacity]; }

    public boolean isEmpty()
    { return N == 0; }

    public void push(Item item)
    { s[N++] = item; }

    public Item pop()
    { return s[--N]; }
}
```

@#\$*! generic array creation not allowed in Java

Generic stack: array implementation

the way it is

```
public class FixedCapacityStackOfStrings
{
    private String[] s;
    private int N = 0;

    public ..StackOfStrings(int capacity)
    { s = new String[capacity]; }

    public boolean isEmpty()
    { return N == 0; }

    public void push(String item)
    { s[N++] = item; }

    public String pop()
    { return s[--N]; }
}
```

```
public class FixedCapacityStack<Item>
{
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity)
    { s = (Item[]) new Object[capacity]; }

    public boolean isEmpty()
    { return N == 0; }

    public void push(Item item)
    { s[N++] = item; }

    public Item pop()
    { return s[--N]; }
}
```

the ugly cast



Unchecked cast

```
% javac FixedCapacityStack.java
```

```
Note: FixedCapacityStack.java uses unchecked or unsafe operations.
```

```
Note: Recompile with -Xlint:unchecked for details.
```

```
% javac -Xlint:unchecked FixedCapacityStack.java
```

```
FixedCapacityStack.java:26: warning: [unchecked] unchecked cast
```

```
found   : java.lang.Object[]
```

```
required: Item[]
```

```
    a = (Item[]) new Object[capacity];
```

```
        ^
```

```
1 warning
```

Q. Why does Java make me cast (or use reflection)?

Short answer. Backward compatibility.

Long answer. Need to learn about **type erasure** and **covariant arrays**.



Generic data types: autoboxing

Q. What to do about primitive types?

Wrapper type.

- Each primitive type has a **wrapper** object type.
- Ex: Integer is wrapper type for int.

Autoboxing. Automatic cast between a primitive type and its wrapper.

```
Stack<Integer> s = new Stack<Integer>();  
s.push(17);           // s.push(Integer.valueOf(17));  
int a = s.pop();      // int a = s.pop().intValue();
```

Bottom line. Client code can use generic stack for **any** type of data.