Programming and Algorithms II

Lecture 6: Abstract Data Types

Nicolas Wu

nicolas.wu@bristol.ac.uk

Department of Computer Science
University of Bristol

Abstract Data Types

Abstract Data Types

- An abstract data type (ADT) provides some abstract mathematical object
- The specification is in terms of properties with respect to that object
- Despite their name, these are closer to interfaces than abstract classes

Abstract Data Types

- To model an ADT in Java, we need a *state* and some *operations*, which we *specify* in terms of some mathematical object
- The specification is in the form of *comments* before the class, and before each method
- We must specify the initial state, for the class and the preconditions, postconditions, and return value for each method

Stacks

Stacks

- A *stack* is an abstract data type that comes up very often
- It allows elements to be put onto and removed from the top of the stack
- The natural model for this is to think of a stack as a *list* with a suitably limited vocabulary

Stacks

- First we give the state space and initial value of the ADT

```
// state:  xs : [int]
// init:   xs = []
interface IntStack {
  …
}
```

A Haskell-like notation is a fine tool for specifications

- With these in place, we define each of the operations we want to model by making reference to this state both before and after execution of the method

Stacks

- We provide *preconditions* and *postconditions* before methods as comments

```
// pre: true
// post: xs = x:xs0
public void push(int x);
```

- The preconditions are things that *must* be true before we enter the code
- The postconditions are promises we make about the state after the code executes
- Variables subscripted with 0 are the value of the variable *before* the code is executed

Q. What happens if the pre is false and we run the method anyway?

Stacks

- We provide *preconditions* and *postconditions* before methods as comments

```
// pre: true
// post: xs = x:xs0
public void push(int x);
```

- The preconditions are things that *must* be true before we enter the code
- The postconditions are promises we make about the state after the code executes
- Variables subscripted with 0 are the value of the variable *before* the code is executed

Stacks

Q. What happens if the pre is false and we run the method anyway?

A. Anything can happen. Missiles could get launched. The world might end.

- We provide *preconditions* and *postconditions* thods as comments

```
/ pre: true
// post: xs = x:xs0
public void push(int x);
```

onditions are things that *must* be true enter the code

onditions are promises we make state after the code executes

ariables subscripted with 0 are the value of the variable *before* the code is executed

Stacks

Q. What happens if the pre is false and we run the method anyway?

Q. What if the post condition is always true?

- We provide *preconditions* and *postconditions*

thods as comments

A. Anything can happen. Missiles could get launched. The world might end.

```
/ pre: true
// post: xs = x:xs0
public void push(int x);
```

onditions are things that *must* be true
enter the code

onditions are promises we make
state after the code executes

ariables subscripted with 0 are the value of the variable *before* the code is executed

Stacks

Q. What happens if the pre is false and we run the method anyway?

A. Anything can happen. Missiles could get launched. The world might end.

Q. What if the post condition is always true?

A. Then our code can do anything we want it to. Missiles could get launched.

- We provide *preconditions* and *postconditions*
  thods as comments

```
/ pre: true
// post: xs = x:xs0
public void push(int x);
```

onditions are things that
enter the code

onditions are promise
state after the code e

ariables subscripted with 0 are the value of the variable *before* the code is executed

Q. What if a post condition is always false?

Q. What happens if the pre is false and we run the method anyway?

Q. What if the post condition is always true?

Stacks

- We provide *preconditions* and *postconditions*
- thods as comments

A. Anything can happen. Missiles could get launched. The world might end.

A. Then our code can do anything we want it to. Missiles could get launched.

```
/ pre: true
// post: xs = x:xs0
public void push(int x);
```

onditions are things that
enter the code

onditions are promise
state after the code e

ariables subscripted with 0 are the value of the variable *before* the code is executed

Q. What happens if the pre is false and we run the method anyway?

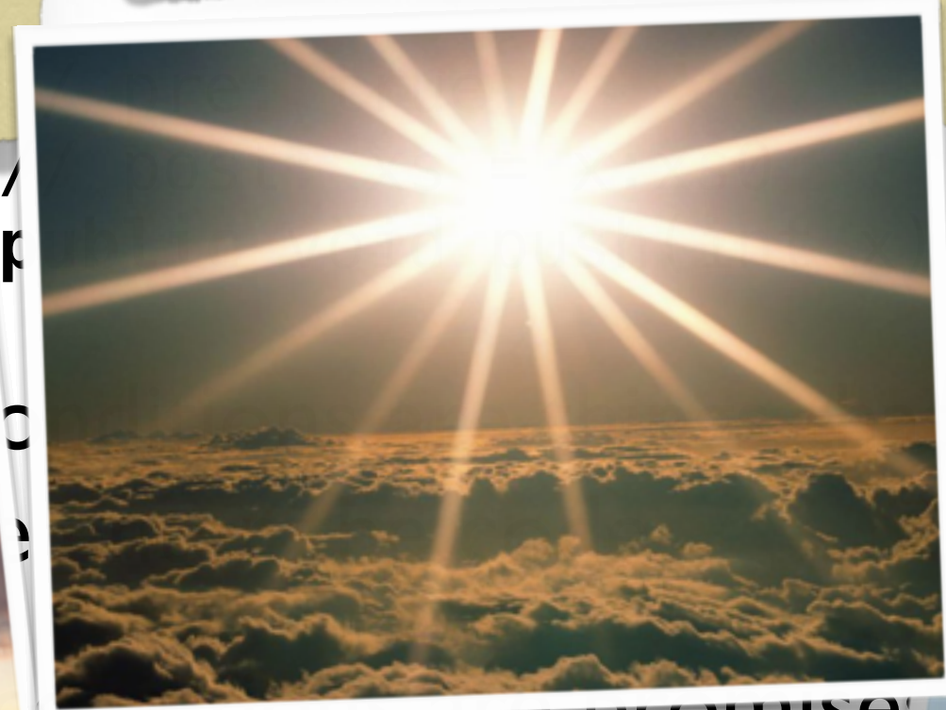
A. Anything can happen. Missiles could get launched. The world might end.

Q. What if a post condition is always false?

A. Then we are claiming our code is miraculous: it makes true become false

Q. What if the post condition is always true?

A. Then our code can do anything we want it to. Missiles could get launched.



- We provide
- Conditions are promise
- state after the code e

Variables subscripted with 0 are the value of the variable *before* the code is executed

Stacks

- The *peek* operation returns the element at the top, without affecting the stack

```
// pre:  xs ≠ []
// post: xs = xs0
// return: head xs
public int peek();
```

- The *pop* operation returns the element at the top, and removes it from the stack

```
// pre:    xs ≠ []
// post:   x:xs = xs0
// return: x
public int pop();
```

x was introduced in the post step and reused in return

Stacks

- The *empty* operation returns true when the stack is empty

```
// pre: true
// post: xs = xs0
// return: xs ≡ []
public boolean empty();
```

- When the precondition is *true* it is customary to remove it
- Similarly, if the return value is void, it is usually left empty

Stacks

```
// state: xs : [int]
// init:  xs = []
interface IntStack {

  // post: xs = x:xs0
  public void push(int x);

  // pre:    xs ≠ []
  // post:   x:xs = xs0
  // return: head xs
  public int pop();

  // pre:  xs ≠ []
  // post: xs = xs0
  // return: head xs
  public int peek();

  // post: xs = xs0
  // return: xs ≡ []
  public boolean empty();
}
```

Concrete Stacks

Concrete Stacks

- So far we have only described what a stack ADT should be
- We haven't implemented a concrete stack
- Many different implementations exist
- We need to tie the abstract and concrete together with an *abstraction function (ABS)*
- Concrete implementations have limitations set out by the *data type invariant (DTI)*

Concrete Stacks

- The abstraction function is a function from the concrete state to the abstract state

```
abs : ArrayIntStack → [int]
abs this = [ this.values[this.size - i]
           | i <- [1 .. this.size] ]
```

- Constraints on the concrete are given by the datatype invariant

```
dti : ArrayIntSet → Bool
dti this = this.size < this.N
```

Concrete Stacks

- Implement an ArrayIntStack based on the Stack ADT, using the abstraction function and datatype invariant

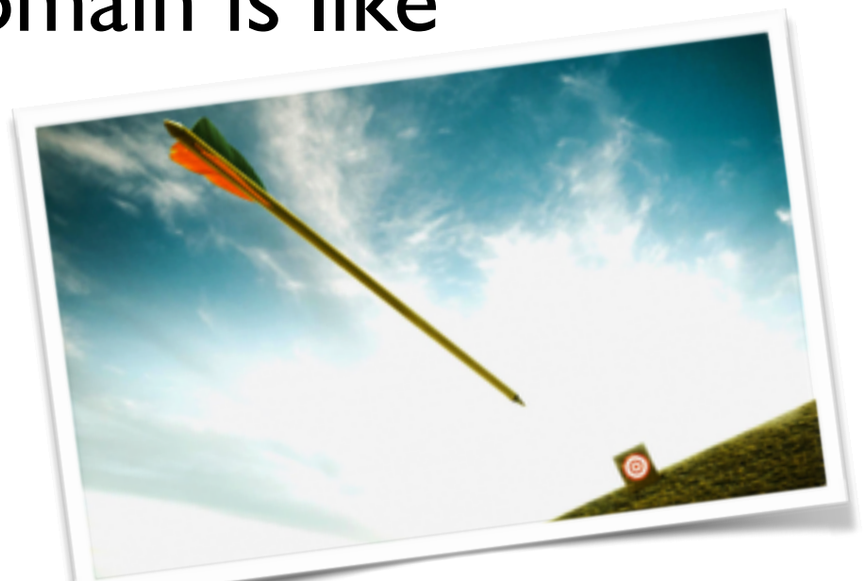
Concrete Stacks

```
// abs : (int[], int, int) → [int]
// abs (stack, size, N) = [ stack[size - i] | 0 < i ≤ size ]
// dti (stack, size, N) = stack.length ≡ N ∧ size ≤ N
class ArrayStack implements Stack {
  int[] stack;
  int size;
  final int N;

  ArrayStack() {
    N     = 100;
    stack = new int[N];
    size  = 0;
  }

  public void push(int x) {
    assert (size > 0);
    stack[size] = x;
    size = size + 1;

  }

  public int pop() {
    assert (size > 0);
    int result = stack[size];
    size = size - 1;
    return result;
  }

  public int peek() {
    int result = stack[size];
    return result;
  }

  public boolean empty() {
    return (size == 0);
  }
}
```

NB. We use assert() to ensure the preconditions hold. Assertions can get erased at runtime

Concrete Stacks

- This might all have seemed far-fetched
- But if we don't model our datatypes on mathematical objects, what else is there?
- Mathematical objects give us a *semantic domain*: they give *meaning* to our computation
- Hacking without a semantic domain is like shooting without a target



Downcasting

Downcasting

- The IntStack we introduced was very rigid in its values: it could only contain ints

```
IntStack stack = new IntArrayStack();
stack.push(42);
stack.push(314);
int n = stack.pop();
```

- In general, we want to deal with objects

Downcasting

- All objects in Java inherit from Object

```
class Object {
   protected Object clone();
   boolean equals(Object obj);
   …
}
```

- We could use this idea to give a broader definition of Stacks

Downcasting

Q. How would we modify this code?

```
// state: xs : [int]
// init:  xs = []
interface IntStack {

  // post: xs = x:xs0
  public void push(int x);

  // pre:    xs ≠ []
  // post:   x:xs = xs0
  // return: head xs
  public int pop();

  // pre:  xs ≠ []
  // post: xs = xs0
  // return: head xs
  public int peek();

  // post: xs = xs0
  // return: xs ≡ []
  public boolean empty();
}
```

Downcasting

Q. How would we modify this code?

```
// state: xs : [int]
// init:  xs = []
interface IntStack {

  // post: xs = x:xs0
  public void push(int x);
  // pre:    xs ≠ []
  // post:   x:xs = xs0
  // return: head xs
  public int pop();

  // pre:  xs ≠ []
  // post: xs = xs0
  // return: head xs
  public int peek();

  // post: xs = xs0
  // return: xs ≡ []
  public boolean empty();
}
```

A. Replace int with Object

```
// state: xs : [Object]
// init:  xs = []
interface ObjectStack {

  // post: xs = x:xs0
  public void push(Object x);
  // pre:    xs ≠ []
  // post:   x:xs = xs0
  // return: head xs
  public Object pop();

  // pre:  xs ≠ []
  // post: xs = xs0
  // return: head xs
  public Object peek();

  // post: xs = xs0
  // return: xs ≡ []
  public boolean empty();
}
```

Downcasting

- Using the ObjectStack is now very similar to before
- We just need to do some *downcasting* to turn an Object into an Integer

```
ObjectStack stack = new ObjectArrayStack();
stack.push(42);
stack.push(314);
Integer n = (Integer)stack.pop();
```

Downcasting

- Let's see how we can abuse downcasting

```
ObjectStack stack = new ObjectArrayStack();
stack.push(new Dog("Milou"));
```

So far, so good ...

Downcasting

- Let's see how we can abuse downcasting

```
ObjectStack stack = new ObjectArrayStack();
stack.push(new Dog("Milou"));
Integer number = (Integer)stack.pop()
```

Milou is not a number

Downcasting

- Downcasting allowed us to turn an *Object* into an *Integer*
- But it only works when the *Object* really was an *Integer*!
- We must rely on the programmer to make sure that the downcasting is applied sensibly
- Unfortunately, if a mistake is made we can only see this problem at runtime
- Furthermore, downcasting costs time!

Generics

Generics

- The downcasting problem was solved by the introduction of *generics* in Java 5
- Generics allow the *compiler* to keep track of object types at *compile time*, rather than relying on the programmer
- With generics, we can parameterise classes with *types*, which removes the need for downcasting for the consumer of ADTs

Generics

- Generics allow us to replace this code:

```
ObjectStack stack = new ObjectArrayStack();
stack.push(314);
Integer n = (Integer)stack.pop();
```

- With something more sensible:

```
Stack<Integer> stack = new ArrayStack<Integer>();
stack.push(314);
Integer n = stack.pop();
```

No more downcasting!

Generics

- Show the generic version of Stack and ArrayStack

Generics

- Making the generic version of Stack involves sprinkling a type parameter
- This is clean!

```
// state: xs : [V]
// init:  xs = []
interface Stack<V> {
  // post: xs = x:xs0
  public void push(V x);

  // pre:    xs ≠ []
  // post:   x:xs = xs0
  // return: head xs
  public V pop();

  // pre:  xs ≠ []
  // post: xs = xs0
  // return: head xs
  public V peek();

  // post: xs = xs0
  // return: xs ≡ []
  public boolean empty();
}
```

Generics

```
// abs : (Object[], int, int) → [V]
// abs (stack, size, N) = [ stack[size - i] | 0 < i ≤ size ]
// dti (stack, size, N) = stack.length ≡ N ∧ size ≤ N
class ArrayStack<V> implements Stack<V> {
  private Object[] stack;
  private int size;
  private final int N;

  ArrayStack() {
    N     = 100;
    stack = new Object[N];
    size  = 0;
  }

  public void push(V x) {
    assert (size > 0);
    stack[size] = x;
    size = size + 1;
  }

  public V pop() {
    assert (size > 0);
    @SuppressWarnings("unchecked")
    V result = (V) stack[size];
    size = size - 1;
    return result;
  }

  public V peek() {
    @SuppressWarnings("unchecked")
    V result = (V) stack[size];
    return result;
  }

  public boolean empty() {
    return (size == 0);
  }
}
```

Q. Did you spot the fishy bit of code?

Generics

```
// abs : (Object[], int, int) → [V]
// abs (stack, size, N) = [ stack[size - i] | 0 < i ≤ size ]
// dti (stack, size, N) = stack.length ≡ N ∧ size ≤ N
class ArrayStack<V> implements Stack<V> {
  private Object[] stack;
  private int size;
  private final int N;

  ArrayStack() {
    N     = 100;
    stack = new Object[N];
    size  = 0;
  }

  public void push(V x) {
    assert (size > 0);
    stack[size] = x;
    size = size + 1;
  }

  public V pop() {
    assert (size > 0);
    @SuppressWarnings("unchecked")
    V result = (V) stack[size];
    size = size - 1;
    return result;
  }

  public V peek() {
    @SuppressWarnings("unchecked")
    V result = (V) stack[size];
    return result;
  }

  public boolean empty() {
    return (size == 0);
  }
}
```

Q. Did you spot the fishy bit of code?

A. There's a downcast!

Generics

- We just got hit with a downcast again, but was it problematic?
- Well, it was *ugly*
- However, it wasn't unsafe: we did not expose the underlying Object[] to the world, nor did the downcast leak out
- But it *was* ugly
- Yes, generics do have limitations, in fact, there's a whole list of them

Generics

- While generics seem like a magical solution, there are a number of sometimes painful restrictions:
  - No primitive type instantiation of generic type
  - No new instances of type parameters
  - No static fields with type parameters
  - No arrays of parameterized types
  - No generic exceptions
  - No overloading to common raw type

http://docs.oracle.com/javase/tutorial/java/generics/restrictions.html