Stacks of 'Known Type'

```
class ArrayStack<V> implements Stack<V> {
  Object[] stack;
  int size;
  final int N;

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

  public void push(V x) {
    assert (size < 100);
    stack[size] = x;
    size = size + 1;
  }

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

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

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

classes and interfaces can be parameterised with types, which can be used as consistent, but unknown (when writing your class) types within them

```
interface Stack<V> {
  void push(V x);
  V pop();
  V peek();
  boolean empty();
}
```

these casts are now under control – we know the casts will always work locally since 'push' only allows objects of type V on the stack in the first place

- if a mistake is made with downcasting we can only see this problem at runtime
- 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, using <…>, parameterise classes with types, which removes the need for uncontrolled downcasting

cast-free!

```
class StackWorld {
  public static void main (String[] args) {
    ...
    Stack<Robot> stack = new ArrayStack<Robot>();
    stack.push(new Robot("C3PO"));
    stack.push(new TranslationRobot("a"));
    stack.push(new CarrierRobot());
    System.out.println(stack.pop().name);
    System.out.println(stack.pop().name);
    System.out.println(stack.peek().name);
} }
```