Collections of Data

- **Arrays**
- **Stacks**
  - Like a real stack (of dishes, books, etc.)
  - Is a LIFO (last-in first-out) collection
  - Can only be accessed via the top
  - Stack class: `java.util.Stack` (look it up)
- **Queues (next week)**
  - A FIFO (first-in first-out) collection

Draw the state-of-memory diagram and give the output

```java
import java.util.*;
public class TestStack {
    public static void main(String args[]) {
        Stack theStack = new Stack();
        String name = new String("Joe");
        int pos;
        System.out.println("theStack.empty() is " + theStack.empty());
        // ignore what push returns
        theStack.push("Mike");
        theStack.push(name);
        theStack.push("Brian");
        System.out.println("theStack.empty() is " + theStack.empty());
        System.out.println("theStack.peek() returns " + theStack.peek());
        System.out.println("theStack.pop() returns " + theStack.pop());
        System.out.println("theStack.peek() returns " + theStack.peek());
        theStack.push(new String("XXX"));
        theStack.push("Brian");
        pos = theStack.search("Mike");
        System.out.println("position of Mike : " + pos);
        pos = theStack.search(name);
        System.out.println("position of Joe : " + pos);
        pos = theStack.search("Brian");
        System.out.println("position of Brian : " + pos);
        // print the contents of theStack (destructive)
        System.out.println("theStack: ");
        while (!theStack.empty()) {
            System.out.println(theStack.pop());
        }
        System.out.println("theStack.empty() is " + theStack.empty());
    }
}
```

Cont’d

```java
System.out.println("theStack.peek() returns " + theStack.peek());
System.out.println("theStack.pop() returns " + theStack.pop());
System.out.println("theStack.peek() returns " + theStack.peek());
theStack.push(new String("XXX"));
theStack.push("Brian");
```

What’s wrong with the following code?

```java
Stack s = new Stack();
System.out.println(s.peek());
```

Cont’d

```java
pos = theStack.search("Mike");
System.out.println("position of Mike : " + pos);
pos = theStack.search(name);
System.out.println("position of Joe : " + pos);
pos = theStack.search("Brian");
System.out.println("position of Brian : " + pos);
// print the contents of theStack (destructive)
System.out.println("theStack: ");
while (!theStack.empty()) {
    System.out.println(theStack.pop());
}
System.out.println("theStack.empty() is " + theStack.empty());
```
Casting

The compiler gives the following error message for the last line: Incompatible type for \( = \). Explicit cast needed to convert double to int.

\[
\text{double } d; \\
\text{int number } = 10; \\
d = 10; \\
\text{number } = d;
\]

What's wrong with the following code? Fix it!

Stack \( s = \text{new Stack}() \);
String \( \text{name} \); 
\( s.\text{push}(\text{"Kiel"}) \);
\( \text{name } = s.\text{pop}() \);

Stack \( s = \text{new Stack}() \);
\text{String } \text{name}; 
\text{s.push(\"Justin\")}; 
\text{name } = \text{s.peek}();

Stacks and Primitive Types

What's wrong with the following code?

The compiler gives the following error message for the last line: Incompatible type for method. Can't convert int to \text{java.lang.Object}.

\[
\text{Stack } s = \text{new Stack}(); \\
\text{int number } = 10; \\
\text{int top}; \\
\text{s.push(number); top } = \text{s.pop}();
\]

Wrapper Classes

Each of the primitive types has a type wrapper class. These classes enable you to manipulate primitive types as objects. The wrapper classes are \text{Byte}, \text{Short}, \text{Integer}, \text{Long}, \text{Float}, \text{Double}, \text{Char}, and \text{Boolean}.

Solution to the code from the previous slide:

\[
\text{Stack } s = \text{new Stack}(); \\
\text{int number } = 10; \\
\text{int top}; \\
\text{Integer numberObject } = \text{new Integer(number)}; \\
\text{s.push(numberObject); top } = \{(\text{Integer} ) \text{s.pop}() \}.\text{intValue}();
\]

Same effect

This will have the same effect as the code from the previous slide.

\[
\text{int top}; \\
\text{s.push(new Integer(10))}; \\
\text{top } = \{(\text{Integer} ) \text{s.pop}() \}.\text{intValue}();
\]
A Simple Stack Application

Stacks can be used to evaluate expressions.

We will do something closely related that’s simpler:
We will look at an expression and check if its brackets
match correctly (are balanced).

Our expression will contain two different kinds of brackets:
( ) and { }. Any symbol other than
these brackets is ignored.

For example,
1. \( (x + (y) * (z)) \) and \( (x + (y) * z) \) are balanced, and
2. \( (x + (y) * (z)) \) and \( (x + (y) * z) \) are not balanced.

A Stack of Characters

We can use an instance of the `java.util.Stack` class to
implement the algorithm from the previous page. That
will require using the wrapper class `Char` and casting.

The code will look much better if we use an instance of the
CharStack class. Later today, we will write
the CharStack class. For now, we will just use it.

The code on the next page is based on code from
Michael Main’s
Data Structures and Other Data Objects Using Java book.

The Algorithm

1. Go through the expression from left to right.
2. If you see a left bracket, push it on the stack.
3. If you see a right bracket, and there is a matching
   left bracket on top of the stack, then pop that
   left bracket off the stack.
4. If you see a right bracket, and there is not a matching
   left bracket on top of the stack, then the expression
   is unbalanced.
5. If you see a non-bracket, ignore it.

If all goes well, and the stack is empty after you have
processed the whole expression, the expression is
balanced; if there are still symbols on the stack, the expression
is unbalanced.

The Code

```java
// needs comments
public static boolean isBalanced(String expr) {
    CharStack leftBrackets = new CharStack();
    boolean failed = false;
    for (int i = 0; !failed && i < expr.length(); i++) {
        switch (expr.charAt(i)) {
        case '(':
        case '{':
            leftBrackets.push(expr.charAt(i));
            break;
        case ')':
            if (leftBrackets.empty() || leftBrackets.pop() != '(') {
                failed = true;
            }
            break;
        case '}':
            if (leftBrackets.empty() || leftBrackets.pop() != '{') {
                failed = true;
            }
            break;
        default:
            break;
        }
    }
    return (!failed && leftBrackets.empty());
}
```

Writing the CharStack
Class

Let’s use an array of char’s for storage of the data
in the CharStack class.

Since you do not yet know how to throw exceptions,
print an error message when something really bad
happens (like pop on an empty stack).

What about push when the underlying array is
completely filled? For now, you can just give an
error message.

If you like a challenge, change push so that the
stack can grow unbounded. You still have to use an
array for storage.
CharStack

```java
public class CharStack {
    // initial size of the array
    // that holds the stack elements
    private static final int INITIAL_SIZE = 100;
    // array that holds the stack elements
    private char[] storage;
    // number of items on the stack
    private int numberOfItems;
    public CharStack() {
        storage = new char[INITIAL_SIZE];
        numberOfItems = 0;
    }

    // naive version of push
    public void push(char item) {
        storage[numberOfItems] = item;
        numberOfItems++;
    }

    public char pop() {
        char topChar = '?';
        if (numberOfItems > 0) {
            topChar = storage[numberOfItems - 1];
            numberOfItems--;
        } else {
            System.out.println("Error: pop on an empty stack!");
        }
        return topChar;
    }

    public char peek() {
        char topChar = '?';
        if (numberOfItems > 0) {
            topChar = storage[numberOfItems - 1];
        } else {
            System.out.println("Error: peek on an empty stack!");
        }
        return topChar;
    }

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

CharStack: Push

```java
// naive version of push
public void push(char item) {
    storage[numberOfItems] = item;
    numberOfItems++;
}
```

CharStack: Pop

```java
public char pop() {
    char topChar = '?';
    if (numberOfItems > 0) {
        topChar = storage[numberOfItems - 1];
        numberOfItems--;
    } else {
        System.out.println("Error: pop on an empty stack!");
    }
    return topChar;
}
```

CharStack (peek & empty)

```java
public char peek() {
    char topChar = '?';
    if (numberOfItems > 0) {
        topChar = storage[numberOfItems - 1];
    } else {
        System.out.println("Error: peek on an empty stack!");
    }
    return topChar;
}
```

A Simple Stack Application

- Reversing things:
  - Finding palindromes or words/sentences that read the same forward or backward.
  - Example: A man, a plan, a canal, Panama!

The Algorithm

1. Go through a string from left to right.
2. If you see an alphabetic character, push it on the stack.
3. Go through the string again. If you see an alphabetic character, pop a character off the stack and see if they match.
3a. If they match, continue the search with the next character.
3b. If they don't match stop, because the word isn't a palindrome.

If all goes well, and the stack is empty after you have processed the whole string, then the string is a palindrome. If there are still symbols on the stack, the string is not a palindrome.