Programming with Data Structures

CMPSCI 187
Spring 2016

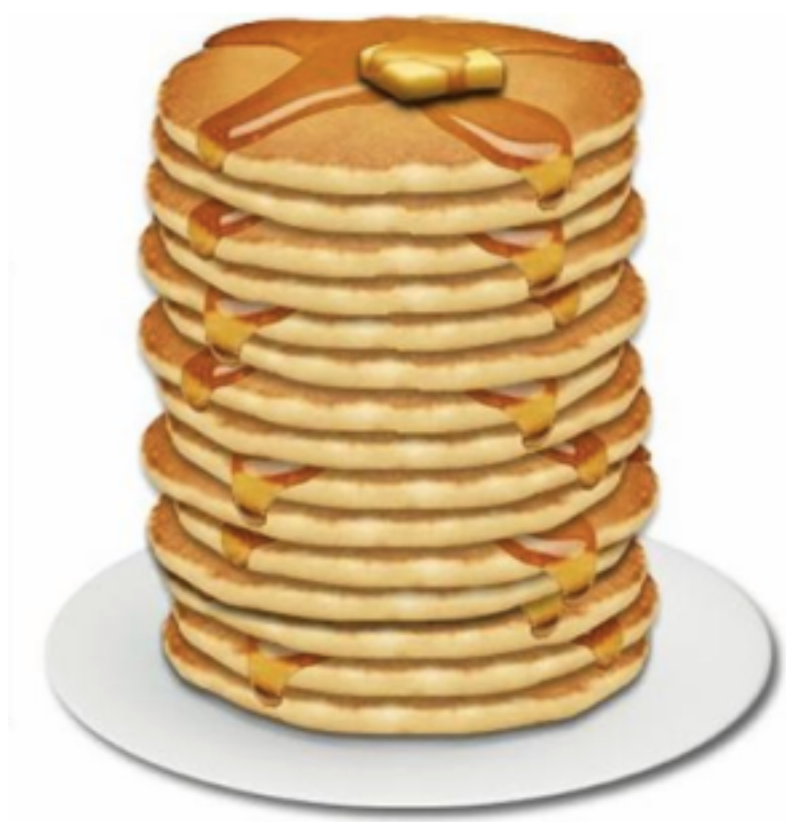
- **Please find a seat**
  - **Try to sit close to the center (the room will be pretty full!)**
- **Turn off or silence your mobile phone**
- **Turn off your other internet-enabled devices**

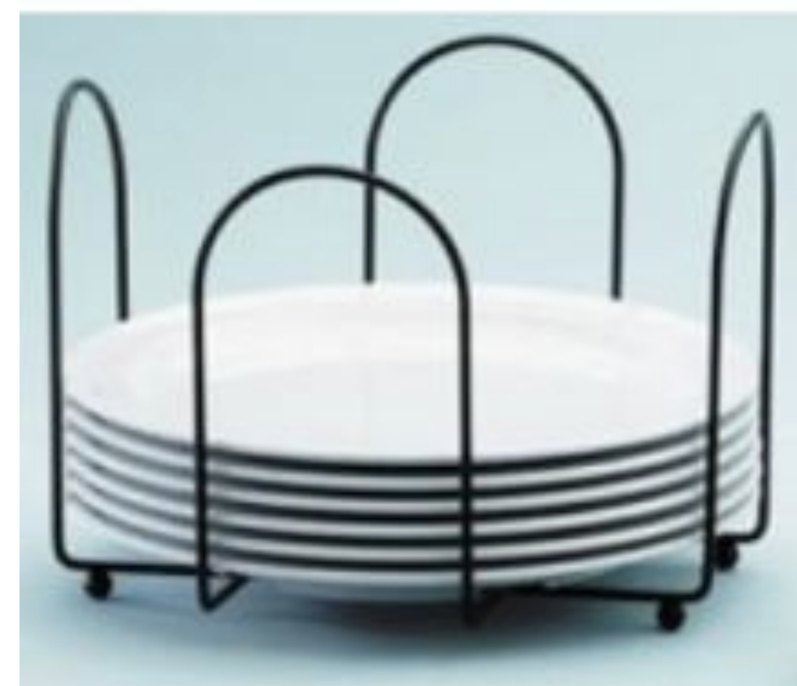
What is a Stack?

- A **stack** stores a collection of objects with the following restricted access:
  - You can only access the **top** element of the stack.
  - You can **push** a new object (to the top of the stack).
  - You can **pop** the top element of the stack (which removes it from the stack).
  - You can **peek**, which means looking at the **top** element without removing it.
  - No way to directly access other elements in the stack.

What is a Stack?

- Here are some real-world examples of stack:





What is a Stack?

- Pop always removes the last element that's pushed into the stack. This is called **LIFO** (last-in-first-out).
- It seems so restricted, why do we need it?
  - Turns out this is a great data structure for computer systems to manage method calls and returns.
  - Imagine you started cleaning your house (**task A**), but realized to do so you need to buy cleaning supplies (**task B**). To buy supplies you need to get cash first (**task C**). But your car is broken and you have to fix it first (**task D**). Think about the order these tasks come up and the order they are done.

Underflow and Overflow

- Popping from an empty stack causes an error condition called **stack underflow**. Before every pop, we need to check and make sure the stack is not empty. Otherwise throw a underflow exception.

- Some stacks are **bounded**, meaning that they have a fixed capacity. Pushing onto a full stack causes **stack overflow**, and we hence before every push we have to check and make sure the stack is not full (still has available spots).

Application 1 — Reversing a Word

- Say you are to write a program that accepts a word, and outputs the word in reverse order.
  - PROGRAM -> MARGORP
- This can be done easily using a stack:
  - Push each character one by one to stack
  - Pop the stack one by one

Application 1 — Reversing a Word

```
public void reverse(String word) {
  Stack s;
  for(int i=0;i<word.length;i++)
    s.push(word.charAt(i));
  while(!s.isEmpty())
    System.out.print(s.pop());
}
```

Application 2 — Delimiter Matching

- You want to write a program to make sure the parentheses in a math expression are balanced:
  - `(w * (x + y) / z – (p / (r – q) ) )`
- It may have several different types of delimiters: **`braces{}, brackets[], parentheses()`**
  - Each opening (left) delimiter must be matched by a closing (right) delimiter.
  - A delimiter that opens the last must be closed by a matching delimiter first. For example, [a * (b + c] + d) is wrong!

Application 2 — Delimiter Matching

- Think for a moment about how you would implement this algorithm.

Application 2 — Delimiter Matching

- Read characters one-by-one from the expression.
- Whenever you see a **left** (opening) delimiter, **push** it to stack.
- Whenever you see a **right** (closing) delimiter, **pop** from stack and check **match** (i.e. same type?)
- If they don’t match, report mismatch error.
- What happens if the stack is **empty** when you try to match a closing delimiter?
- What happens if the the stack is **non-empty** after you reach to the end of the expression?

Object Types in Class Definition

- So far we've learned to write StringLog and Linked List, but the type of data stored in these classes are hard-coded:

```
class ArrayStringLog {
  protected String[] log;
  protected int lastIndex;
}

class LLStringNode {
  private String info;
  private LLStringNode link;
}
```

Object Types in Class Definition

- What if we need to define classes to store other types of data, like ArrayIntegerLog, LLAppleNode, do we have to redefine the class over and over again?? That's awful!

```
class ArrayIntegerLog {
  protected Integer[] log;
  protected int lastIndex;
}

class LLAppleNode {
  private Apple info;
  private LLAppleNode link;
}
```

Generics

- Java (and many other languages) has a mechanism called **generics** to create entire families of classes (or interfaces) at once, where the object **type** is provided as a **parameter** to the class (or interface) definition.

- Each different type variable gives us a new class (or interface).

- In C++, generics are known as **templates**.

Generic Classes

- Let's start with an example of a generic class. Say we want to define a generic Log class that can be a StringLog, but can also be an IntegerLog or other types of logs:

```
public class Log<T> {
    private T[ ] log;
    private int lastIndex = -1;
    … …
}
```

- Think of this as a 'template' to create classes.

Generic Classes

```
public class Log<T> {
    private T[ ] log;
    private int lastIndex = -1;
    … …
}
```

- When using the generic class, you provide a specific type T inside the angle brackets:

```
Log<Integer> intLog = new Log<Integer>();
Log<String> strLog = new Log<String>();
```

- You **cannot** use primitive data types (int, float etc.), you have to use their wrapper classes (Interger, Float etc.)

Generic Classes

- When the compiler sees **`Log<Interger>`**, it basically creates a new class **`LogInterger`** (if it hasn't been created already) and substitutes every **`T`** in the 'template' with **`Integer`**.

```
public class LogInterger {
    private Interger[ ] log;
    private int lastIndex = -1;
    public void insert(Interger e);
    … …
}
```

Generic Classes

- Similarly, when it sees `Log<String>`, it basically creates a new class `LogString` (if it hasn't been created already) and substitutes every `T` in the 'template' with `String`.

```
public class LogString {
    private String[ ] log;
    private int lastIndex = -1;
    public void insert(String e);
    … …
}
```

- Generics is a deep topic: there are complications that we can ignore for the moment.

Exceptions and Error Handling

- We’ve mentioned two cases where the stack may cause an error, namely underflow and overflow. These are called **exceptions** (conditions that break the normal program execution and require special processing)
- We don’t want the program to simply crash or terminate when exceptions happen.
- Java provides exception handling through the **Exception** class, **throw**, and **try-catch-finally** statements.

Exceptions and Error Handling

- Method can **throw** an **Exception** object (which carries information about the exception) when it detects an error condition. Example:

```
public void push() {
   … …
   throw new Exception("Stack overflow!");
}
```

- You can extend the **Exception** class to define inherited exception classes with custom data.
- The caller method can use **try-catch** statement to handle the error condition and process accordingly.

Exceptions and Error Handling

- A lot of Java-provided methods, such as I/O related, require exception handling.
- The **`try-catch-finally`** statement:

```
try {
  // call methods
} catch(IOException e) {
  // handle IOException
  System.out.println(e.getMessage());
} finally {
  // this block is always executed
  whether exception or not
}
```

Exceptions and Error Handling

- If the caller doesn't handle the exception, it will be thrown further (i.e. deferred) to the upper-level caller. If it reaches the `main` method and not caught / handled there either, the program execution will terminate with error.

- Some exceptions are "**checked**", meaning that the compiler insists that it be told whenever a method may throw exceptions to its calling method.

```java
class Nested {

  public static void checkInteger(int i) throws Exception {
    if (i < 0)
      throw new Exception("Negative Number!!\n");
    System.out.println("Check passed.");
  }

  public static void main(String[] args) {
    try {
      checkInteger(-100);
    }
    catch(Exception e) {
      System.out.print( e.getMessage() );
    }
    System.out.println("Execution complete.");
  }
}
```

What is the output when executed ?

```java
class Nested {

  public static void checkInteger(int i) throws Exception {
    if (i < 0)
      throw new Exception("Negative Number!!\n");
    System.out.println("Check passed.");
  }

  public static void main(String[] args) {
    try {
      checkInteger(-100);
    }
    catch(Exception e) {
      System.out.print( e.getMessage() );
    }
    System.out.println("Execution complete.");
  }
}
```

2. So it is specified here

1. Exception can be raised here

3. It is caught and handled here

What is the output when executed ?

```java
class Code {

   public static void foo(int i) throws RuntimeException {
      System.out.print("1");
      if (i < 0)
         throw new RuntimeException("2");
      System.out.print("3");
   }

   public static void main(String[] args) {
      try {
         System.out.print("4");
         foo(-1);
         System.out.print("5");
      }
      catch(RuntimeException e) {
         System.out.print( e.getMessage() );
      }
      System.out.print("6");
   }
}
```

```java
class Code {

  public static void foo(int i) throws RuntimeException {
    System.out.print("1");
    if (i < 0)
      throw new RuntimeException("2");
    System.out.print("3");
  }

  public static void main(String[] args) {
    try {
      System.out.print("4");
      foo(-1);
      System.out.print("5");
    }
    catch(RuntimeException e) {
      System.out.print( e.getMessage() );
    }
    System.out.print("6");
  }
}
```

Output is: 4126