# Today's topics

- The **Stack** Data Structure
- Exceptions
- Generics
- Iterable and Iterator

# What is a Stack?

- A **stack** stores a collection of objects with the following restricted access:
  - 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.
  - **In sum, you can only operate the top element**, there is no way to 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.

# Clicker Question #1

```
1. I have a pen, I have an apple -> apple pen

2. I have a pen, I have a pineapple -> pineapple pen

3. apple pen, pineapple pen -> pen pineapple apple pen
```

Which is not LIFO

(a) Line 1
(b) Line 2
(c) Line 3
(d) None of them

# Underflow and Overflow

- Popping from an empty stack causes an error condition called **stack underflow exception**. Before every pop, we need to check and make sure the stack is not empty. Otherwise it will disrupt the normal program execution.

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

# Exceptions and Error Handling

- Underflow and overflow are conditions that break the normal program execution and require special processing. These are called **exceptions.**

- We don’t want the program to simply crash or terminate when exceptions happen.

- Java provides ways for handling exceptions through the **Exception** class, **throw** statement, and the **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. Program flow will jump to exception handling.

```
public void pop() {
  … …
  throw new Exception(“Stack underflow!”);
  // code below is skipped if exception is thrown above
}
```

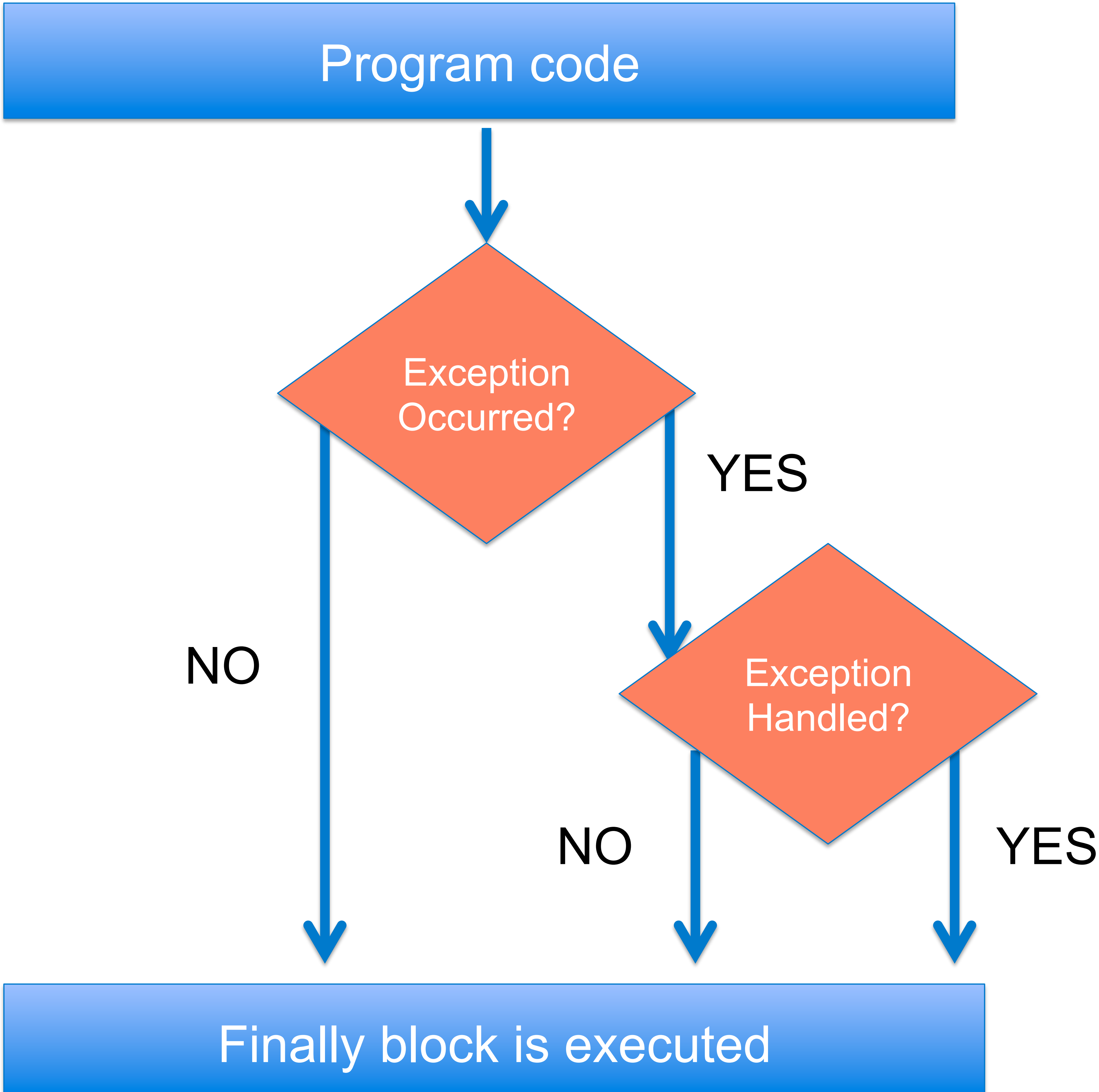
- You can also extend the **`Exception`** class to define a customized exception class. For example, Java’s **`IOException`** extends the **`Exception`** class, and most I/O related methods require handling **`IOException`**.

# Exceptions and Error Handling

- The caller method can use try-catch statement to handle the error condition and process accordingly.

  The **try-catch-finally** sequence:

```
try {
  // Call some method, e.g. pop()
} catch(IOException e) {
  // handle IOException error
} catch(Exception e) {
    // System.out.println(e.getMessage());
} finally {
  // this block always executes
  // generally for clean-up work
}
```



Program code
Exception
Occurred?
YES
NO
Exception
Handled?
NO
YES
Finally block is executed

# Checked Exceptions

- The **`Exception`** class itself, and most class extending are called a **checked** exception, meaning the compiler insists that it be told whenever a method may throw such exceptions.

- Methods that throw such exceptions must be explicitly handled, meaning callers of such methods must either wrap these methods in **try-catch** statement, or use the **throws** clause to defer the handling further up.

- One common checked exception is **`IOException`**.

```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 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.");
    }
}
```

**Negative number.**
**Execution complete.**

# Clicker Question #2

```
class Code {

    public static void foo(int i) throws Exception {
        System.out.print("a");
        if (i < 0)  throw new Exception("b");
        System.out.print("c");
    }

    public static void main(String[] args) {
        try {
            System.out.print("d");
            foo(-1);
            System.out.print("e");
        }
        catch(Exception e) {
            System.out.print(e.getMessage());
        }
        System.out.print("f");
    }
}
```

The output is:

(a) dabef

(b) dabf

(c) dabcef

(d) dabcf

(e) daebf

# Unchecked Exceptions

- Classes extending Java’s **`RuntimeException`** class are called **unchecked exceptions**.
- Methods that throw such exceptions do **<u>not</u>** need to declare them in the signature, and callers are **<u>not</u>** required to `try-catch` them.
- Some common unchecked exceptions include `NullPointerException`, `IndexOutOfBoundException`, `ClassCastException`, `ArithmeticException`
- As confusing as it sounds, `RuntimeException` itself is a subclass of `Exception`.

# The Stack Interface

```
public interface StackInterface<T> {
    void push(T element) throws StackOverflowException;
    T pop()  throws StackUnderflowException;
    T peek() throws StackUnderflowException;
    boolean isEmpty();
    boolean isFull();
}
```

- Methods that throw non-checked exceptions can **optionally** declare such exceptions using the **throws** clause.
- Let’s assume these exceptions are runtime exceptions.
- T is a generic type (we will cover this next).

# How many stack?

- Here are some real-world examples of stack:

# Object Types in Class Definition

So far we've learned to write data structures like `StringLog`, `LLStringNode`, 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 `IntegerLog`, `LLAppleNode`, do we have to re-write 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. Here T represents the generic type:

```
public class ArrayLog<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 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 (Integer, Float etc.)

# Generic Classes

**Conceptually*** when the compiler sees `Log<Integer>`, it basically creates a new class, say, called `LogInteger`, and substitutes every T in the 'template' with `Integer`.

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

* This is how generics are handled by C++, but in Java it's actually not handled the same way.

# Generic Classes

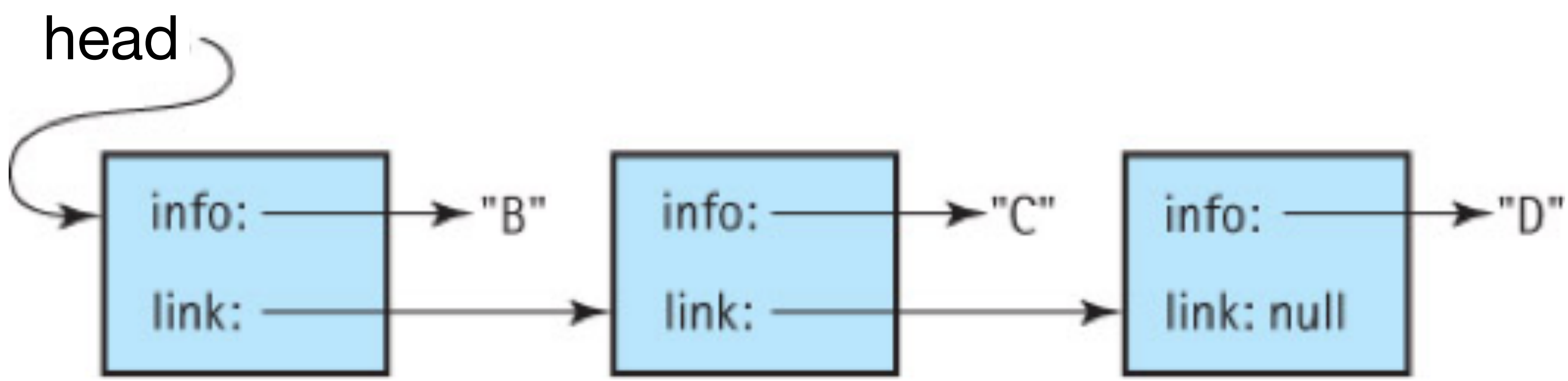
Similarly, when it sees `Log<String>`, it basically creates a new class `LogString` and substitutes every T in the 'template' with `String`.

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

- This greatly reduces programmer's work if you want to reuse the same data structure but just changing data type.
- Java imposes complications with generics that we can ignore for the moment.

# Linked Stack Implementation

- Recall linked list:




- To implement stack using a linked list, we first define a **generic node class `LLNode<T>`**. Its `info` variable points to an object of generic type `T`, and its `link` variable points to another `LLNode<T>` object.

# Linked Stack Implementation

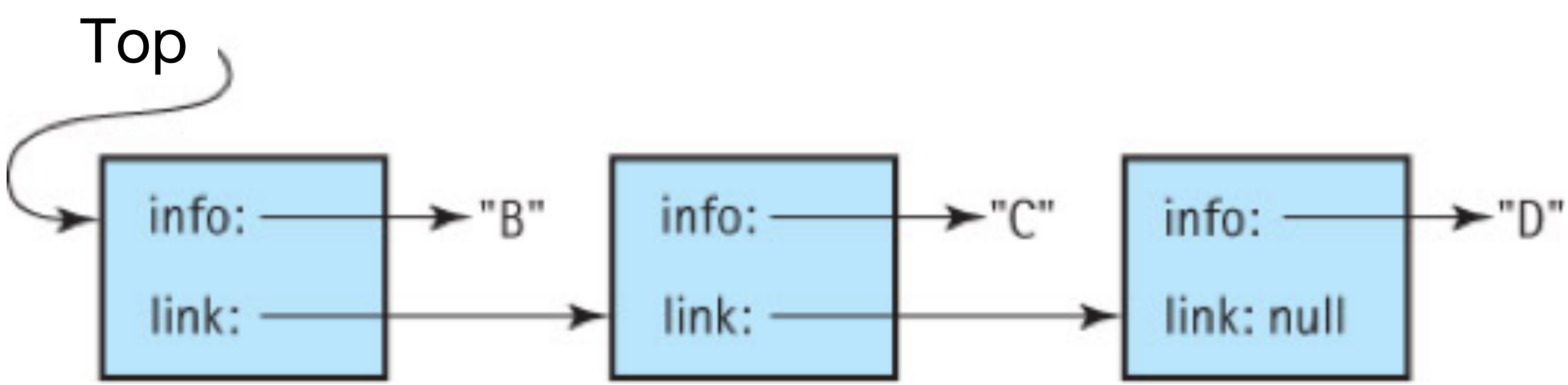
```
public class LLNode<T> {
  public LLNode<T> link;
  public T info;

  public LLNode(T info) { this.info = info; }
}
```

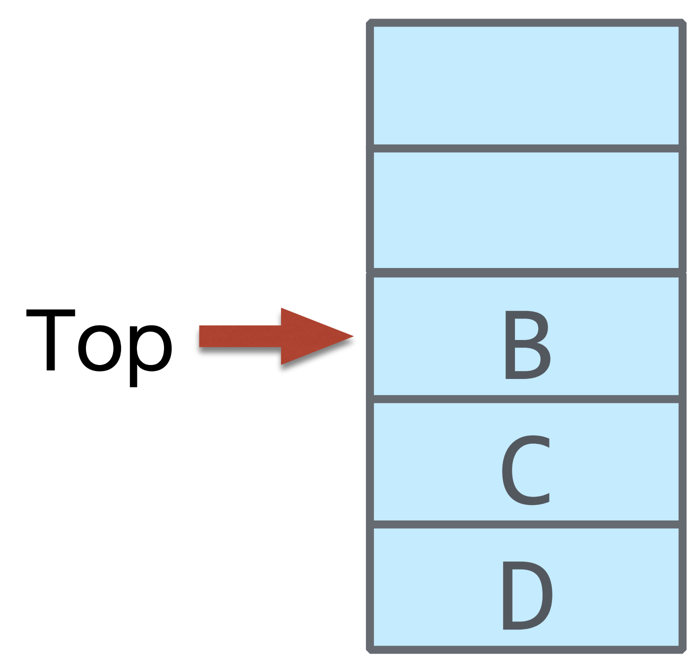
- For convenience, here we have declared both link and info as public, thus any class can directly access and modify these two variables without setters and getters.
- Note that **`LLNode`** and **`<T>`** must go together (i.e. can’t write **`LLNode`** without **`<T>`**), that is **except the constructor**!

# LinkedStack<T>

- To implement a generic Stack using linked list, the only variable we need is a pointer to the **top** of the stack; and we can use front insertion for pushing elements.




Conceptually this is what the stack look like:



# LinkedStack<T>

```
public interface StackInterface<T> {
    void push(T element) throws StackOverflowException;
    T pop()  throws StackUnderflowException;
    T peek() throws StackUnderflowException;
    boolean isEmpty();
    boolean isFull();
}
```

```
public class LinkedStack<T>
       implements StackInterface<T> {
    private LLNode<T> top;
    public  LinkedStack() { top = null; }
}
```

# LinkedStack Methods

```
public boolean isEmpty( ) {
   return (top == null);
}

public T peek() {
   if (isEmpty())
      throw new StackUnderflowException("underflow");
   return top.info;
}
```

- Note that **`top`** is a node, and we need to return its content (`top.getInfo()`) rather than the node itself.

# Linked Stack Methods

- What about the `push()` and `pop()` methods?
- The `push()` method accepts a type `T` object, creates a new node, and adds that to the linked list.
  - The node will be inserted at the beginning of the linked list (i.e. front insertion).
  - Here `push()` will never throw an exception, why?

# push() and pop()

```
public void push (T element) {
   LLNode<T> newNode = new LLNode<T>(element);
   newNode.link = top;
   top = newNode;
}
```

# push() and pop()

```
public void push (T element) {
   LLNode<T> newNode = new LLNode<T>(element);
   newNode.link = top;
   top = newNode;
}

public T pop() {
   if (isEmpty())
      throw new StackUnderflowException(“underflow”);
   T element = top.info;
   top = top.link;
   return element;
}
```

# push() and pop()

Can we simplify the pop() method to the following:?

```
public T pop() {
   if (isEmpty())
      throw new StackUnderflowException(“underflow”);
   top = top.link;
   return top.info;
}
```

# Iterator

- An **`iterator`** is an object that allows you to traverse the elements in a collection one-by-one, regardless of how the collection is implemented.
- Java's **`iterator<T>`** interface has three methods:
  - **hasNext()**: returns true if the collection has more elements to traverse, false otherwise.
  - **next()**: returns the next element. To get all elements, call this repeatedly. If there is no more element (`hasNext()` returns `false`), this method throws `NoSuchElementException`. (unchecked)
  - **remove()**: removes the last returned element.

# Iterator<T> and Iterable<T>

Example (let’s say object **list** stores a collection of **type T** objects, doesn’t matter how data is stored internally)

```
Iterator<T> iter = list.iterator();
while (iter.hasNext()) {
  T element = iter.next();
  … …
}
```

# Putting it Together

Let's say List<T> is a generic class that stores a collection of type T objects, and it implements the Iterable<T> interface:

```
class List<T> implements Iterable<T> {
  public Iterator<T> iterator() {
    return new ListIterator<T>(…);
  }
  // variables, constructors, other methods
}
```

Note the iterator() above returns a ListIterator<T> object, which implements the Iterator<T> interface. It might look like this:

# Putting it Together

```
class ListIterator<T> implements Iterator<T>
{
  public boolean hasNext() {…}
  public T next() {…}
  public void remove() {…}
  // variables, constructors, other methods
}
```

- Here `ListIterator<T>` is aware of the specific implementation details of `List<T>` and provides the above methods to traverse the collection of objects.
- Reference to the `List<T>` object is typically passed in through the constructor.

# Putting it Together

```
List<String> list = new List<String>();
… …
Iterator<String> ite = list.iterator();
// ite is of class ListIterator<String>
while (iter.hasNext()) {
  String e = iter.next();
  … …
}
```

In fact, there is an easier way (special for loop) to traverse the list:

```
for(String e : list) {
  … …
}
```

# Summary

```
List<String> list = new List<String>();
… …
for(String e : list) {
   … …
}
```

- A class (e.g. List) can implement the `Iterable<T>` interface. If so it must implement the `iterator()` method that returns an `iterator` object.
- The `iterator` class implements the `Iterator<T>` interface, and must provide `hasNext()`, `next()`, and `remove()` methods.
- To traverse the elements in an iterable object, you can use the special for loop as shown above.

# Clicker Question #3

```
Log<Integer> A = new Log<Integer>();
… …
for(Integer i : A) {
  System.out.println(i);
}
```

For the above code to compile, class `Log<T>` <u>is required</u> to implement which of the following?

(a) the `hasNext()`, `next()` and `remove()` methods

(b) the `iterator()` method

(c) all four methods in (a) and (b)

(d) none of the above methods