Queue

What is a queue

A **queue** is a data structure that organizes data into a Black Friday line type of data structure. The first element to enter the queue will be the first element into the store. Similarly, in a Java program, the first element to enter the queue will be the first element to be returned. The Java utility that utilizes a queue is `Interface Queue<E>`. Notice the `<E>`, which allows generic programming with queues.

Notable methods

- `add(E item)` - Adds the item into the queue.
- `remove()` - Deletes and returns the head of the queue (the first element).

API

http://docs.oracle.com/javase/7/docs/api/java/util/Queue.html

Stack

What is a stack

A **stack** is like as it sounds: a stack. Like a stack of plates or a deck of cards, once we put a plate on the top, we naturally take the top plate off. Unlike a queue, a **stack pops off the top element** instead of the bottom (first) element. In conclusion, stack follows **FILO** (first in, last out). To use the stack, we call the `Class Stack<E>`

Notable methods

- `push(E item)` - "Pushes" an item onto the stack (puts the plate on the top)
- `pop()` - "Pops" an item off the stack. Returns + removes the top (removes the top plate).

API

http://docs.oracle.com/javase/7/docs/api/java/util/Stack.html

Call Stacks

Call stacks is the progression to the program in the order of resolution. For example, in the `main()` class:

```
public static void main(String[] args) {
  Dog fido = new Dog(4);
  fido.setNumLegs(2);
}

public class Dog {

  int numLegs;

  public Dog(int numLegs) {
    if (isValid(numLegs)) this.numLegs = numLegs;
    else this.numLegs = 4;
  } //End constructor

  private static boolean isValid(int num) {
    if (num > 0) return true;
    return false;
  } //End isValid()

} //End Dog()
```

In this program, when the `main()` executes, it puts `main()` onto the call stack. When `main()` encounters `new Dog(4)`, we put that on the call stack, meaning it'll do that call before it continues the `main()` method. Inside the execution for `Dog()`, we encounter `isValid()`, which will put yet another method onto our call stack. This leads to a stack like the following:

1. `isValid()`
2. `Dog()`
3. `main()`

Since stacks are FILO (first in last out), `main()` will resolve last, and `isValid()` will resolve first, then return to it's **saved location** in the next method in the stack (after `isValid()`, `Dog()`) and continue it's program from there. When the method ends, we **pop the method off the call stack**.

Cleaning up unused memory

When we finish a method, what happens to the variables and other types of memory units? When we pop off stacks, the stack memory is cleared. However, for memory for the **heap**, or the main storage (that doesn't get erased on a stack pop), we must rely on the **garbage collector** to delete these unused data slots.

Stack Variables

Remember that when we use variables in methods, we are manipulating stack variables. When we call a method, we push on **copies of the variables, not the acutal variables**. For example:

```
main() {
  int a = 3;  //main stack variable a
  int b = 10; //main stack variable b
  increment(3, 10);
  System.out.println(x + " " + y);
} //End main()

public void increment(int a, int b) {
  a++; //increment stack variable a
  b++; //increment stack variable b
} //End increment()
```

Would print `3 10`. This is because the variable `increment()` manipulated was **it's stack variable**. To manipulate a variable outside it's stack location, we should make the function `return` a value, or pass an address.

The Heap

The **heap** is where objects (like the dogs we make) are stored. When we make new objects, we store those in the heap. When we run methods, we are storing method-related data in the call stack. Returning to our previous example, we pass an object `Integer` instead of an int variable to get `increment()` to work:

```
main() {
  Integer x = new Integer(3);
  increment(x);
  System.out.println(x);
} //End main()

public void increment(Integer x) {
  x = x + 1;
} //End increment
```

Would output `4`. This is because we **passed an object (address)** instead, which allows the method to work directly on the object.