What is a Stack?

- A stack is a data structure, we mean something that is meant to hold data and provides certain operations on that data.

- An array is an example of simple data structure. Its ‘operations’ are using offsets with brackets.
  - ex. someArray[2];

- “Hold data and provide certain operations”.
  - Does this sound familiar?
What is a Stack?

- A stack is a bit more sophisticated than an array

- The two important things...
  - Its primary operations are “push” and “pop”
  - Its LIFO (last-in, first-out)

- New items can be added to the top of the stack.

- Items can be removed only from the top of the stack.
What is a Stack?

- Imagine a stack of plates…
- You can “**push**” a new plate onto the stack of plates by sticking a plate on top
- You can “**pop**” a plate off the stack by taking the top one
Operations

- **void push(T value)**
  - Puts the value of type T at the top of the stack.

- **T pop()**
  - Removes the value of type T from the top of the stack and returns it.
  - If stack is empty, returns null.

- **T peek()**
  - Returns the value of type T from the top of the stack (without removing it).
  - If stack is empty, returns null.

- **boolean empty()**
  - Returns true if the stack is empty, otherwise it returns false.

- **int getSize()**
  - Returns the number of elements in the stack.

- **int getCapacity()**
  - Returns the maximum number of characters allowed in the stack.
Motivation

- There are many data structures. Trees, Graphs, Sets and so on.

- In fact, if you continue in CS you will take a whole course on them.

- Why are there so many?
  - They are generalized solutions to common problem types in programming.
  - Each data structure has characteristics that make it suited to certain types of problems.
Motivation

- A stack is suited to problems where that have some element of LIFO behavior.
- We saw this in the call stack memory region.
- Another example, suppose we want to reverse a String?
  - We could push each character onto a stack, then pop them all off, creating a new string.
  - We would need a stack of characters.
/**
 * A method that takes a string returns its reverse.
 * ex "army" => "ymra", "4321" => "1234"
 */

private static String reverse(String string) {
    Stack<Character> stack = new Stack<>();
    for (int i = 0; i < string.length(); ++i) {
        stack.push(string.charAt(i));
    }

    String reverse = "";
    for (int i = 0; i < string.length(); ++i) {
        reverse += stack.pop();
    }

    return reverse;
}
Implementation

- The previous slide is using the Stack data structure included in the Java standard library.
- When learning data structures (and OOP!) it's useful to implement it yourself.
- See `stack/StackOfCharacters.java`
Inheritance
What Is Inheritance?

- Different kinds of physical objects often have a certain amount in common with each other.

- Mountain bikes, road bikes, and tandem bikes all share the characteristics of bicycles (current speed, current gear, etc.)

- Yet each also defines additional features that make them different:
  - Ex. tandem bicycles have two seats and two sets of handlebars
Superclasses & Subclasses

- Object-oriented programming allows classes to *inherit* commonly used state and behavior from other classes.

- In our example, Bicycle is the *superclass* of MountainBike, RoadBike, and TandemBike

- In Java, each class is allowed to have one direct *superclass*, and each superclass has the potential for an *unlimited* number of *subclasses*. 
Superclasses & Subclasses

- Bicycle
  - Mountain Bike
  - Road Bike
  - Tandem Bike
Superclasses & Subclasses

- When two classes have a superclass/subclasses relationship the subclass “is-a” superclass. Literally. A mountain bike is a bicycle.

- Moreover, if you create a class that inherits from another class, the subclass can do all the things the superclass can do….
  - It has all its non-private methods
  - It has all its non-private variables

- However the subclass can add additional behavior that is specific to its type.

- For example, a MountainBike might have an extra variable “suspension”, which is a specialized characteristic of only mountain bikes.
‘Extends’ Keyword

- The syntax for creating a subclass is simple.
- At the beginning of your class declaration, use the extends keyword, followed by the name of the class to inherit from.

```java
public class MountainBike extends Bicycle {
    // new fields and methods defining
    // a mountain bike would go here
    // .. and MountainBike 'inherits' some
    // methods and data from Bicycle
}
```
Extensibility

- This gives MountainBike the same fields and methods as Bicycle, yet allows its code to focus exclusively on the features that make it unique.

- In this way, we get ‘code reuse’ and ‘extensibility'

- It allows us to find a class which implements roughly what we need, and “extend” it to add and modify it to our specific purpose.
Subclasses of Subclasses

- We can even extend our MountainBike.

```java
public class CrossCountryMountainBike extends MountainBike {
    // inherits Bicycle methods and any methods
    // that MountainBike has added.
}
```

- So we can build ‘hierarchies’ of inheritance.
In fact, if you do not specify an ‘extends’ declaration in your code the compiler will add one for you.

By default, all objects extend `java.lang.Object`

All classes extend Object directly or indirectly.

The `java.lang.Object` class is the ‘root of the class hierarchy’ in Java.
java.lang.Object

class A {}

public class SimpleClass {

    public SimpleClass() {
        A a = new A();
    }

}
More on Superclasses & Subclasses

- In an inheritance hierarchy, classes are known as “superclasses” and “subclasses”

- For some class B that extends A
  - A is a superclass of B
  - B is a subclass of A
  - A is a subclass of Object
  - B is also a subclass of Object (via A)

- Sometimes also called ‘parent’ and ‘child’ classes
More on Superclasses & Subclasses

- A subclass inherits certain properties (data) and methods from the superclass.
- Moreover, it has all the non-private behaviors and data of the superclass.
- But the subclass can also …
  - Add new instance variables
  - Add new methods
  - *Override* the methods of the superclass
Suppose we extend a class and its method does not do exactly what we need?

For example, suppose we extended Bicycle and created MotorizedBicycle.

Conceptually, a ‘pedal’ method for Bicycle would not work properly for a MotorizedBicycle.
Overriding

- This is called method overriding.

- We declare a pedal method in MotorizedBicycled that knows how to function for a bike with a motor.

- When users call the pedal method on our MotorizedBicycled, it calls our special “overridden” implementation.

- We are effectively ‘hiding’ Bicycle’s pedal method with our own.
We’ve seen an example of a method override in the Snape exercise with `toString()`.

Potion had a `toString()` method.

`java.lang.Object` is Potion’s superclass, it has a `toString()` method.

And since the one for Object is useful to exactly nobody ever in the history of programming we implemented our own.

This is an example of overriding.
Summary

- We can extend classes to reuse their powers.
- We can specialize our extensions to have behaviors specific to our new class by overriding.
Processing
What is Processing?

- A ‘library’ of code that can be used to create graphical programs.
- A teaching tool for programming.
- Developed in part here at NYU’s ITP program.
  - https://processing.org/overview/
  - https://processing.org/tutorials/gettingstarted/
- The documentation is targeted at the beginner
How to Install Processing?

- You need to set it up in each Eclipse project you want to use it in.
- Instructions are listed on the site.
How to Use Processing

- ‘import’ the PApplet
- Just like importing Scanner.
- Declares your intention to use something from the Processing library.

```java
import processing.core.PApplet;

public class ProcessingExample00 extends PApplet {

    public void draw() {
        rect(random(0, 500), random(0, 500), 10, 10);
    }
}
```
How to Use Processing

- ‘extend’ the PApplet
- Thus inheriting all of PApplet’s methods (including a main method)

```java
import processing.core.PApplet;

public class ProcessingExample00 extends PApplet {

    public void draw() {
        rect(random(0, 500), random(0, 500), 10, 10);
    }
}
```
How to Use Processing

- Then we ‘override’ its draw() method.

- We are going to ‘draw’ to the ‘canvas’ whatever we would like.

```java
import processing.core.PApplet;

public class ProcessingExample00 extends PApplet {

    public void draw() {
        rect(random(0, 500), random(0, 500), 10, 10);
    }
}
```
How to Use Processing

- The in our draw() implementation we use other methods of PApplet to write things to the canvas.

- Note that draw gets invoked in a loop by the PApplet class.

```java
import processing.core.PApplet;

public class ProcessingExample00 extends PApplet {

    public void draw() {
        rect(random(0, 500), random(0, 500), 10, 10);
    }
}
```

- See processing/*.java