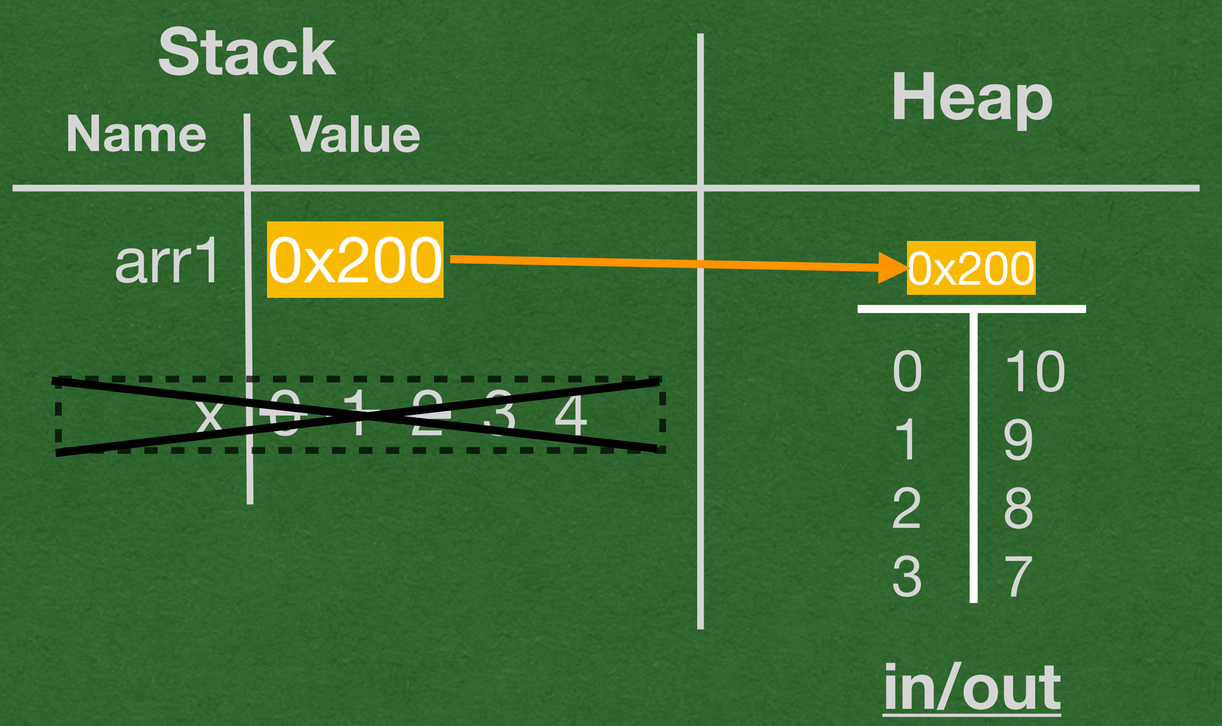
Objects and Classes

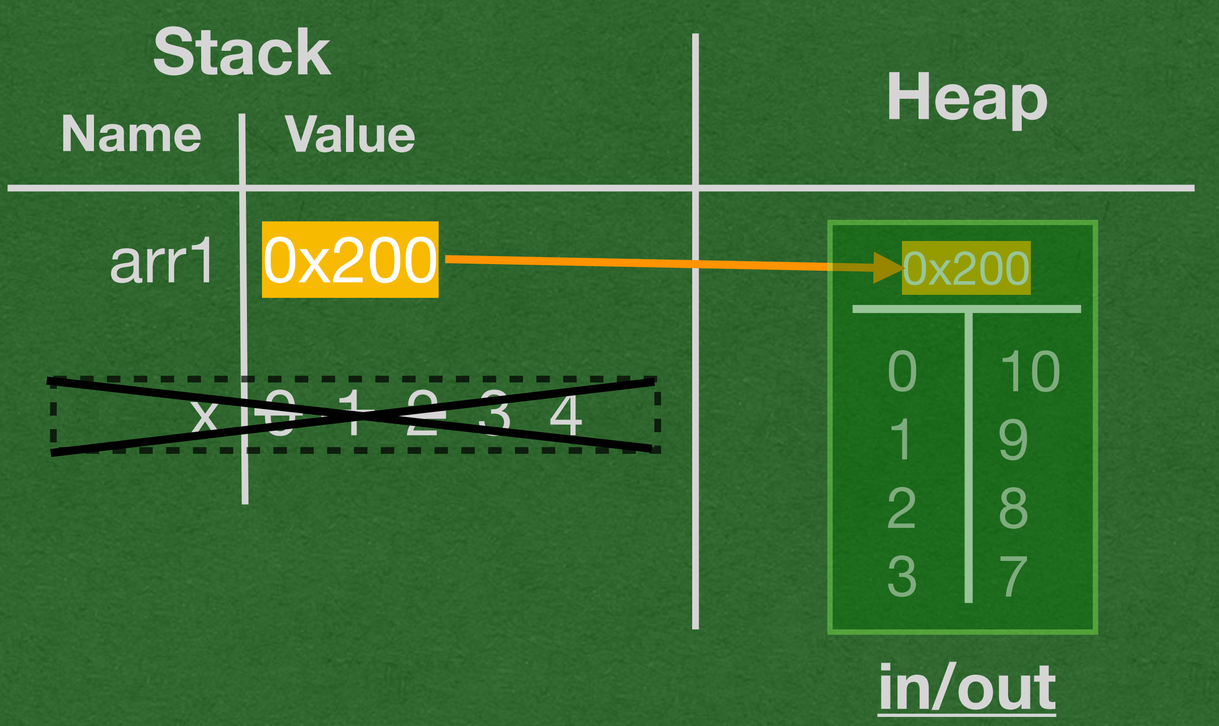
```
ArrayList<Integer> arr1 = new ArrayList<>();
for (int x=0; x<4; x++) {
    arr1.add(10-x);
}
```



Recall this ArrayList example

- When we use the **new** keyword, we are creating a new ***object*** of ***type*** ArrayList
- Using **new** calls a special method called a constructor

```
ArrayList<Integer> arr1 = new ArrayList<>();
for (int x=0; x<4; x++) {
    arr1.add(10-x);
}
```



Recall this ArrayList example

- Objects are stored on the heap
- Only a ***reference*** to the location of the object is stored in variables
- We use the dot operator to follow the reference and access the objects methods

```java
package java.util;

/*** This code is significantly reduced for the slide!          ***/
/*** To see the full code, ctrl+click on ArrayList in IntelliJ  ***/
public class ArrayList<E> {

}
```

```java
package week2;

import java.util.ArrayList;

public class ArrayListExample {

    public static void main(String[] args) {
        ArrayList<Integer> arr1 = new ArrayList<>();
        for (int x=0; x<4; x++) {
            arr1.add(10-x);
        }
    }

}
```

Classes

- Classes are templates used to create objects
- A class tells java how to create our objects
- Defining a class allows us to create many objects of the same type
  - We can create many ArrayLists objects from a single ArrayList class

```java
package java.util;

/*** This code is significantly reduced for the slide!          ***/
/*** To see the full code, ctrl+click on ArrayList in IntelliJ  ***/
public class ArrayList<E> {

    public ArrayList() {

    }

}
```

```java
package week2;

import java.util.ArrayList;

public class ArrayListExample {

    public static void main(String[] args) {
        ArrayList<Integer> arr1 = new ArrayList<>();
        for (int x=0; x<4; x++) {
            arr1.add(10-x);
        }
    }

}
```

Classes

- Using **new** calls a special method called a constructor
- A constructor is a method that has the same name as the class
- Constructors are not static

```java
package java.util;

/*** This code is significantly reduced for the slide!          ***/
/*** To see the full code, ctrl+click on ArrayList in IntelliJ  ***/
public class ArrayList<E> {

    public ArrayList() {

    }


    public ArrayList(Collection<? extends E> c) {
        /* removed for slides */
    }

}
```

```java
package week2;

import java.util.ArrayList;
import java.util.Arrays;

public class ArrayListExample {

    public static void main(String[] args) {
        ArrayList<Integer> arr1 = new ArrayList<>(Arrays.asList(10, 9, 8, 7));

    }
}
```

Classes

- You can have multiple constructors with **different** parameter lists
  - This is called method overloading
  - True for all methods, not just constructors
- This ArrayList constructor can be convenient

```java
package java.util;

/*** This code is significantly reduced for the slide!          ***/
/*** To see the full code, ctrl+click on ArrayList in IntelliJ  ***/
public class ArrayList<E> {

    private Object[] elementData;
    private int size;

    public ArrayList() {

    }

    public ArrayList(Collection<? extends E> c) {
        /* removed for slides */
    }

    public int size() {
        return this.size;
    }

}
```

Classes

- Objects have both *state* and *behavior*
- *State:* Any variables declared outside all the classes methods become part of the state of objects
  - We call these *instance* variables
- Behavior: Any non-static methods define the behavior of an object

```java
package java.util;

/*** This code is significantly reduced for the slide!          ***/
/*** To see the full code, ctrl+click on ArrayList in IntelliJ  ***/
public class ArrayList<E> {

    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

    private Object[] elementData;
    private int size;

    public ArrayList() {
        this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
    }

    public ArrayList(Collection<? extends E> c) {
        /* removed for slides */
    }

    public int size() {
        return this.size;
    }

}
```

Classes

- The constructor is used to initialize the ***state*** of the new object
- Set the instance variables to their initial values
- If the constructor takes parameters, set the instance variables based on those parameters

```java
package java.util;

/*** This code is significantly reduced for the slide!          ***/
/*** To see the full code, ctrl+click on ArrayList in IntelliJ  ***/
public class ArrayList<E> {

    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

    private Object[] elementData;
    private int size;

    public ArrayList() {
        this.elementData = ArrayList.DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
    }

    public ArrayList(Collection<? extends E> c) {
        /* removed for slides */
    }

    public int size() {
        return this.size;
    }

    public E get(int index) {
        return (E) this.elementData[index];
    }

    private void add(E e, Object[] elementData, int s) {
        if (s == elementData.length)
            elementData = grow();
        elementData[s] = e;
        this.size = s + 1;
    }
    public boolean add(E e) {
        add(e, this.elementData, this.size);
        return true;
    }
}
```

Classes

- Add more non-static methods to define more behavior for the objects we create
- Behavior often depends on the current state of the object (Values stored in its instance variables)
- In this example, the add method is overloaded with a private add method
  - This is called a *helper* method

```java
package java.util;

/*** This code is significantly reduced for the slide!          ***/
/*** To see the full code, ctrl+click on ArrayList in IntelliJ  ***/
public class ArrayList<E> {

    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

    private Object[] elementData;
    private int size;

    public ArrayList() {
        this.elementData = ArrayList.DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
    }

    public ArrayList(Collection<? extends E> c) {
        /* removed for slides */
    }

    public int size() {
        return this.size;
    }

    public E get(int index) {
        return (E) this.elementData[index];
    }

    private void add(E e, Object[] elementData, int s) {
        if (s == elementData.length)
            elementData = grow();
        elementData[s] = e;
        this.size = s + 1;
    }
    public boolean add(E e) {
        add(e, this.elementData, this.size);
        return true;
    }
}
```

Classes

- The "this" keyword is a variable containing a reference to the object that called a method
- For constructors, it's a reference to the object being created
- *More detail when we get to a memory diagram*

```java
package java.util;

/*** This code is significantly reduced for the slide!          ***/
/*** To see the full code, ctrl+click on ArrayList in IntelliJ  ***/
public class ArrayList<E> {

    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

    private Object[] elementData;
    private int size;

    public ArrayList() {
        this.elementData = ArrayList.DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
    }

    public ArrayList(Collection<? extends E> c) {
        /* removed for slides */
    }

    public int size() {
        return this.size;
    }

    public E get(int index) {
        return (E) this.elementData[index];
    }

    private void add(E e, Object[] elementData, int s) {
        if (s == elementData.length)
            elementData = grow();
        elementData[s] = e;
        this.size = s + 1;
    }
    public boolean add(E e) {
        add(e, this.elementData, this.size);
        return true;
    }
}
```

Encapsulation

- Encapsulation is when we hide data and details not relevant to the outside user
- Any state/behavior we want others to use: Make it **public**
- Any implementation details not relavent to your user: Make it **private**

```java
package java.util;

/*** This code is significantly reduced for the slide!          ***/
/*** To see the full code, ctrl+click on ArrayList in IntelliJ  ***/
public class ArrayList<E> {

    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

    private Object[] elementData;
    private int size;

    public ArrayList() {
        this.elementData = ArrayList.DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
    }

    public ArrayList(Collection<? extends E> c) {
        /* removed for slides */
    }

    public int size() {
        return this.size;
    }

    public E get(int index) {
        return (E) this.elementData[index];
    }

    private void add(E e, Object[] elementData, int s) {
        if (s == elementData.length)
            elementData = grow();
        elementData[s] = e;
        this.size = s + 1;
    }
    public boolean add(E e) {
        add(e, this.elementData, this.size);
        return true;
    }
}
```

Encapsulation

- As a user of ArrayLists
  - You don't care how the underlying state is stored (As a plain array)
  - You don't care that the add method is overloaded
- Hide the details we don't need to care about

```java
package java.util;

/*** This code is significantly reduced for the slide!          ***/
/*** To see the full code, ctrl+click on ArrayList in IntelliJ  ***/
public class ArrayList<E> {

    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

    private Object[] elementData;
    private int size;

    public ArrayList() {
        this.elementData = ArrayList.DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
    }

    public ArrayList(Collection<? extends E> c) {
        /* removed for slides */
    }

    public int size() {
        return this.size;
    }

    public E get(int index) {
        return (E) this.elementData[index];
    }

    private void add(E e, Object[] elementData, int s) {
        if (s == elementData.length)
            elementData = grow();
        elementData[s] = e;
        this.size = s + 1;
    }
    public boolean add(E e) {
        add(e, this.elementData, this.size);
        return true;
    }
}
```

Encapsulation

- All of the public state and behavior defines your public interface
- This is how the outside world uses your code
- These are the methods you call when using an ArrayList
- We call this an API (Application Programming Interface)

```java
package java.util;

/*** This code is significantly reduced for the slide!          ***/
/*** To see the full code, ctrl+click on ArrayList in IntelliJ  ***/
public class ArrayList<E> {

    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

    private Object[] elementData;
    private int size;

    public ArrayList() {
        this.elementData = ArrayList.DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
    }

    public ArrayList(Collection<? extends E> c) {
        /* removed for slides */
    }

    public int size() {
        return this.size;
    }

    public E get(int index) {
        return (E) this.elementData[index];
    }

    private void add(E e, Object[] elementData, int s) {
        if (s == elementData.length)
            elementData = grow();
        elementData[s] = e;
        this.size = s + 1;
    }
    public boolean add(E e) {
        add(e, this.elementData, this.size);
        return true;
    }
}
```

Static

- Static variables and methods can be accessed through the ***class***
- Non-static methods are accessed through ***objects*** (instances of the class)
- If we just say "method" we mean "non-static method"

Classes

```java
package java.lang;

/*** This code is significantly reduced for the slide!    ***/
/*** To see the full code, ctrl+click on Math in IntelliJ ***/
public class Math {

    public static final double E = 2.718281828459045;
    public static final double PI = 3.141592653589793;
    public static final double TAU = 2.0 * PI;

    private static final double DEGREES_TO_RADIANS = 0.017453292519943295;
    private static final double RADIANS_TO_DEGREES = 57.29577951308232;

    /**
     * Don't let anyone instantiate this class.
     */
    private Math() {}

    public static int abs(int a) {
        return (a < 0) ? -a : a;
    }

}
```

- Some classes are only used for their static state and behavior
- It doesn't make sense to create a new Math object
- Use the static variables and methods from the Math class

Creating Our Own Class

Classes

Let's create a Player class with this state and behavior

- State
  - Maximum hit points
  - Current hit points
  - Name
  - Attack power - *Next Lecture*
- Behavior - *Next Lecture*
  - Can take damage - *Next Lecture*
  - Can attack other Players - *Next Lecture*

Player Class

```
public class Player {
    private int maxHP;
    private int hp;
    private String name;

    public Player(String name, int maxHP) {
        this.maxHP = maxHP;
        this.hp = maxHP;
        this.name = name;
    }

}
```

- We create the Player class
- Add the instance variables needed for all the state of a Player
- Write a constructor matching the name of the class
  - Takes any parameters needed for initialization
  - Initializes the instance variables

Player Class

```
public class Player {
    private int maxHP = 10;
    private int hp = 10;
    private String name;

    public Player(String name) {
       this.name = name;
    }

}
```

- You can also initialize instance variables when they are declared
- Use this if you want every object to have the same initial value for a variable

Player Class

```
public class Player {
    private int maxHP;
    private int hp;
    private String name;

    public Player(String name, int maxHP) {
        this.maxHP = maxHP;
        this.hp = maxHP;
        this.name = name;
    }

}
```

- Our instance variables are all private
  - Very common in Java
- Leverage encapsulation
  - Hide the details of your code
  - Expose public methods for others to interact with your code
- So how does anyone use this state?...

```java
public class Player {
    private int maxHP;
    private int hp;
    private String name;

    public Player(String name, int maxHP) {
        this.maxHP = maxHP;
        this.hp = maxHP;
        this.name = name;
    }

    public int getMaxHP() {
        return maxHP;
    }
    public void setMaxHP(int maxHP) {
        this.maxHP = maxHP;
    }

    public int getHP() {
        return hp;
    }
    public void setHP(int hp) {
        this.hp = hp;
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

}
```

Player Class

- Getters and Setters!

- Write public methods that allow access to your state
- Getters - Return the value of the requested variable
- Setters - Takes a value and reassigned the instance variable

```java
public class Player {
    private int maxHP;
    private int hp;
    private String name;

    public Player(String name, int maxHP) {
        this.maxHP = maxHP;
        this.hp = maxHP;
        this.name = name;
    }

    public int getMaxHP() {
        return maxHP;
    }
    public void setMaxHP(int maxHP) {
        this.maxHP = maxHP;
    }

    public int getHP() {
        return hp;
    }
    public void setHP(int hp) {
        if (hp <= this.maxHP) {
            this.hp = hp;
        } else {
            this.hp = this.maxHP;
        }
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

}
```

Player Class

- Why???
  - It would be easier to just make our variables public!
- Control.
  - If we want to sanitize values, add code to the setter
  - If you want to format output, add code to the getter
- If others write code to access your variables directly, you do not have this option!

```java
public class Player {
    private int maxHP;
    private int hp;
    private String name;

    public Player(String name, int maxHP) {
        this.setMaxHP(maxHP);
        this.setHP(maxHP);
        this.setName(name);
    }

    public int getMaxHP() {
        return maxHP;
    }
    public void setMaxHP(int maxHP) {
        this.maxHP = maxHP;
    }

    public int getHP() {
        return hp;
    }
    public void setHP(int hp) {
        if (hp <= this.maxHP) {
            this.hp = hp;
        } else {
            this.hp = this.maxHP;
        }
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

}
```

Player Class

- You should call your setters in your constructor
- Ensures your checks are ran when an object is created

```java
public class Player {
    private int maxHP;
    private int hp;
    private String name;

    public Player(String name, int maxHP) {
        this.setMaxHP(maxHP);
        this.setHP(maxHP);
        this.setName(name);
    }

    public int getMaxHP() {
        return maxHP;
    }
    public void setMaxHP(int maxHP) {
        this.maxHP = maxHP;
    }

    public int getHP() {
        return hp;
    }
    public void setHP(int hp) {
        if (hp <= this.maxHP) {
            this.hp = hp;
        } else {
            this.hp = this.maxHP;
        }
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

}
```

Player Class

- Classes define new **types**
  - The ArrayList *class* defines the ArrayList *type*
  - Our Player *class* defines the Player *type*
- We can use Player wherever we could use any other type
  - As variable types
  - As parameter types in methods
  - As the return type of methods
  - As type parameters of data structures