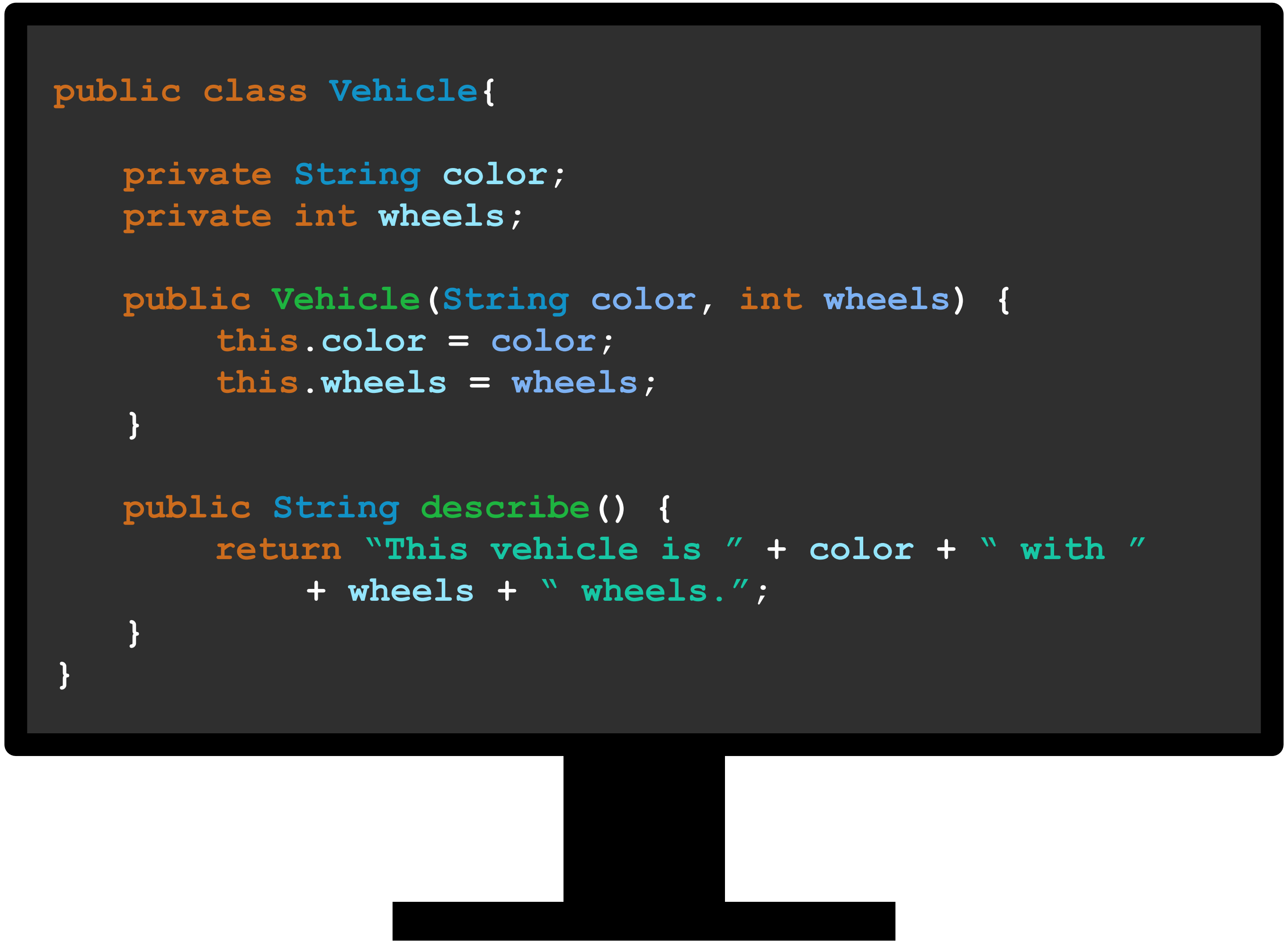


```
public class Vehicle{

    private String color;
    private int wheels;

    public Vehicle(String color, int wheels) {
         this.color = color;
         this.wheels = wheels;
    }

    public String describe() {
         return “This vehicle is ” + color + “ with ”
              + wheels + “ wheels.”;
    }
}
```

Messages and Methods

To instruct a class or an object to perform a task, we send a ***message*** to it.

You can send a message only to the classes and objects that understand the message you sent to them.

A class or an object must possess a matching ***method*** to be able to handle the received message.

Messages and Methods

A method defined for a class is called a ***class method***, and a method defined for an object is called an ***instance method***.

A value we pass to an object when sending a message is called an ***argument*** of the message.

Access Modifiers

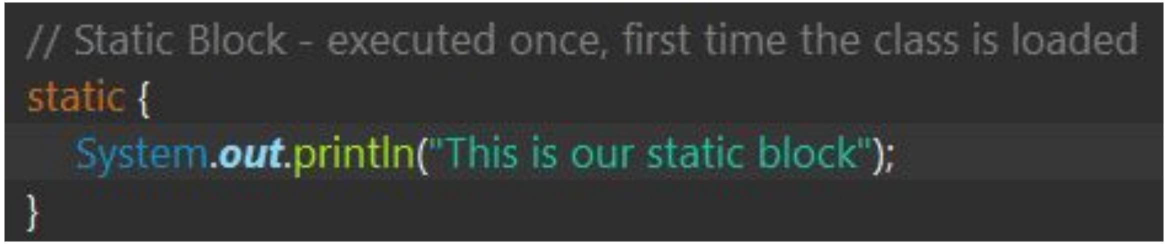
Modifier	Class	Package	Subclass	Global
Public	Allowed	Allowed	Allowed	Allowed
Protected	Allowed	Allowed	Allowed	Denied
Default	Allowed	Allowed	Denied	Denied
Private	Allowed	Denied	Denied	Denied

Static Keyword

- Means the method or attribute is bound to the entire Class
  - No object needs creation to call static methods
  - Static attributes are reflected for all objects of a class
    - E.g. - a count int, that counts all objects of a class is static
- To call a static method
  - Use: *ClassName.methodName();*

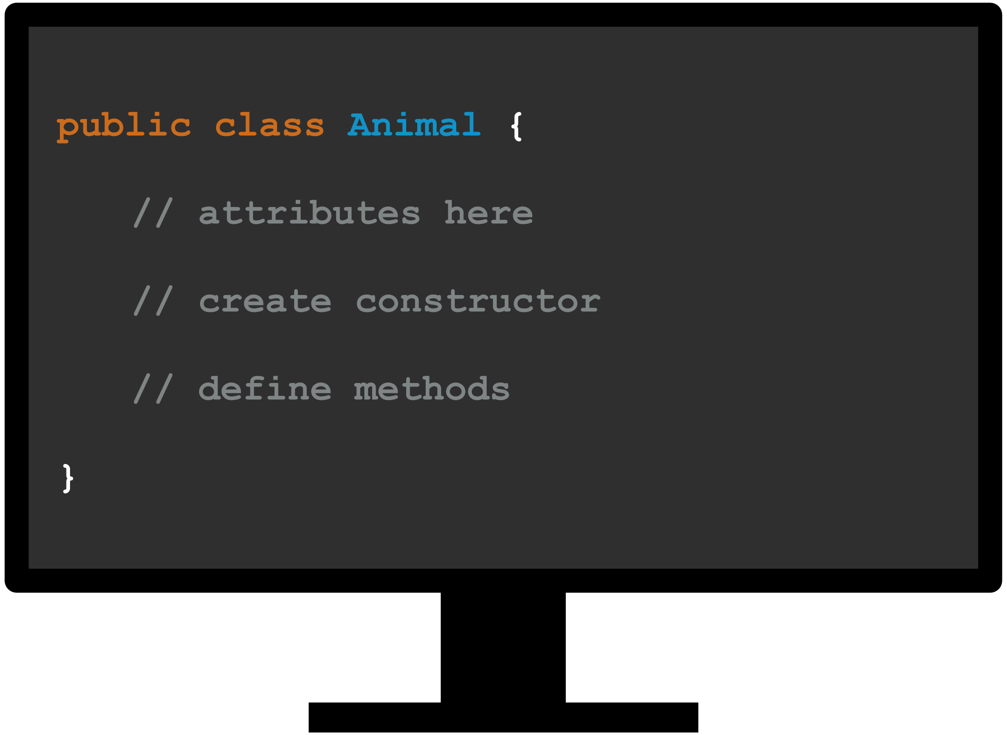
Static Keyword

- Static Block
  - Block of code that executes once, when a class is loaded into the program heap.
  - Will execute *BEFORE* a constructor, but only once.



```
// Static Block - executed once, first time the class is loaded
static {
    System.out.println("This is our static block");
}
```

Class Example



```
public class Animal {

    // attributes here

    // create constructor

    // define methods

}
```

White Board Exercise

Creating a Class Diagram

Create Class	Class Properties	Child Class	Polymorphism	Explain
Choose a topic and create a class for this, draw it up on the board	Create attributes and methods for this class.	Create a child class that can inherit from this original class. Come up with attributes and methods for this child class.	Create a method that will override one of the methods from the parent.	What is happening in this diagram? Is there encapsulation?

Naming Conventions

➔ *Classes* - should be nouns, in mixed case with the first letter of each internal word capitalized

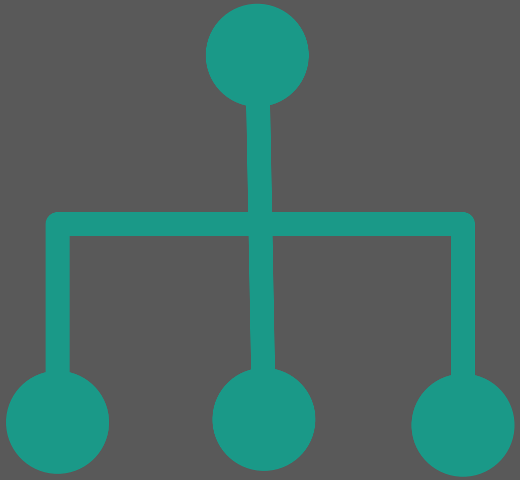
➔ *Interfaces* - should be adjectives, in mixed case with the first letter of each internal word capitalized

➔ *Methods* - should be verbs, in mixed case with the first letter of each internal word capitalized

➔ *Variables* - should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. Lowercase first letter and camelcased

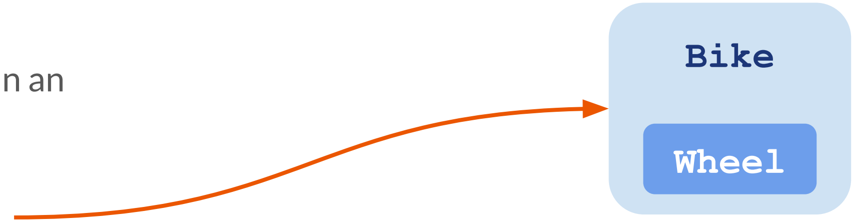
➔ *Final Variables and Enums* - should be all uppercase with words separated by underscores ("_")

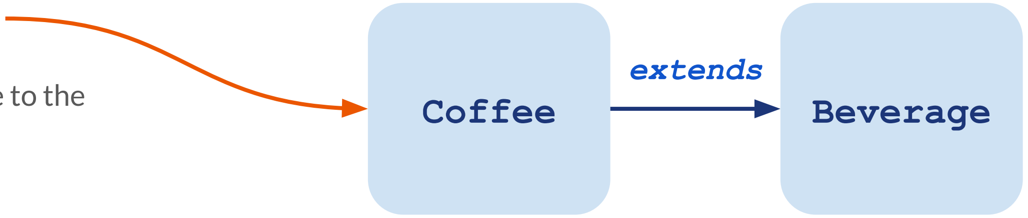
Composition & Inheritance



Has-A vs Is-A Relationship

- **Composition** - a class can contain an instance of another class
  - *Has-A* relationship
    - Ex: a bike ***has a*** wheel
- **Inheritance** - a child class can obtain the properties of its parent class
  - *Is-A* relationship
    - Ex: a coffee ***is a*** beverage
  - Access modifiers determine what attributes/methods are accessible to the child class



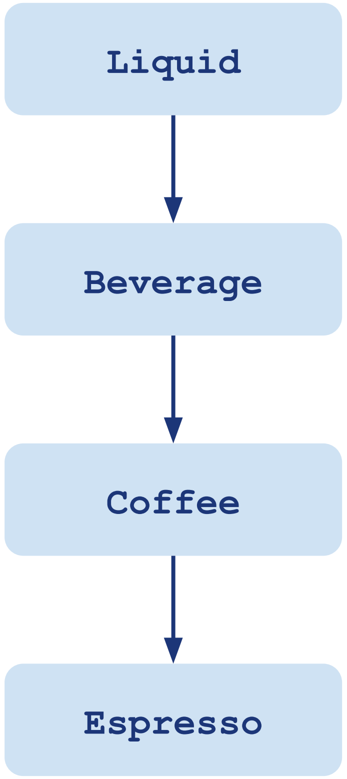


Inheritance Hierarchy

A child class will inherit from its parent class and all the classes its parent inherits from. Because an Espresso is a Liquid, a variable of type Liquid can be assigned an Espresso object. However, a Liquid cannot be an Espresso because a Liquid does not inherit from Espresso.

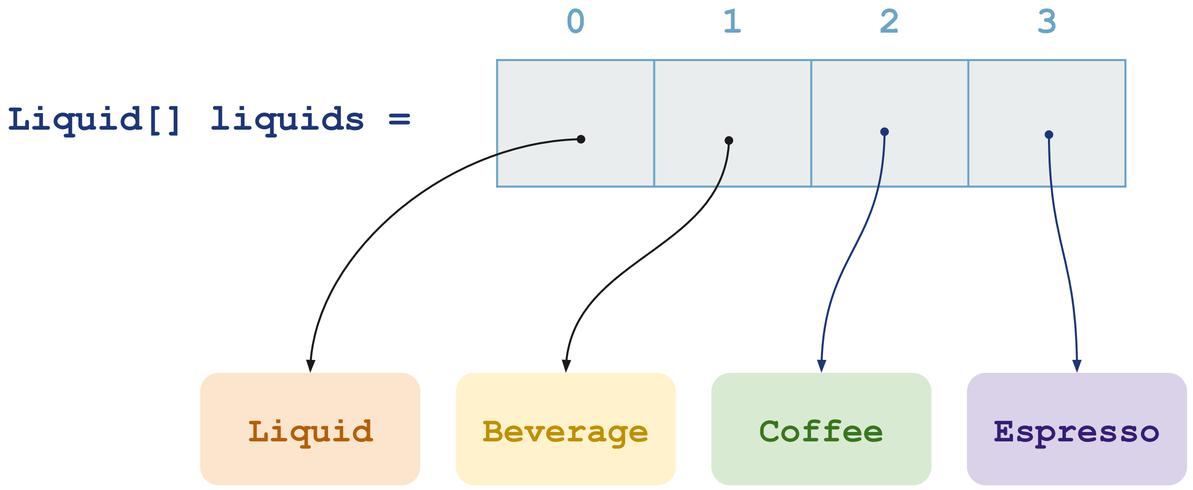
✓
```
Liquid liquid = new Espresso();
```

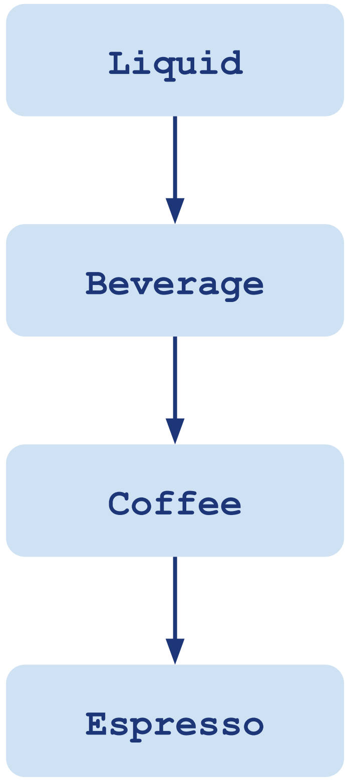
✗
```
Espresso espresso = new Liquid();
```



Inheritance Hierarchy

An array of Liquids can take in any objects that inherit from it. So Beverage, Coffee, and Espresso objects can be placed in this array.





Final Classes and Inheritance

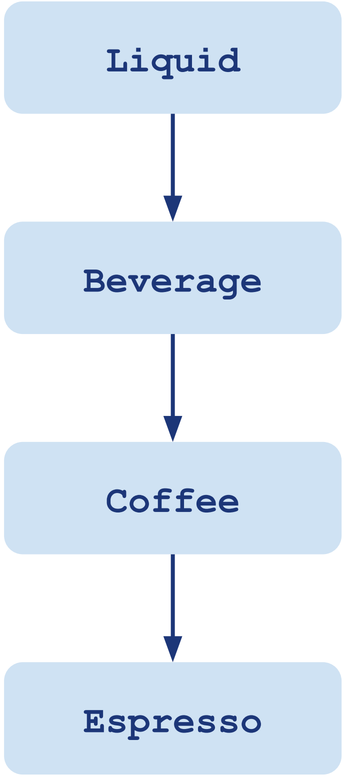
A class declared final cannot be extended by another class. Classes like String are final and use this implementation so no other classes can inherit and use their functionality.

```
public final class Espresso {
     ...
}
```

❌

```
public class Latte extends Espresso {
     ...
}
```

Will cause error

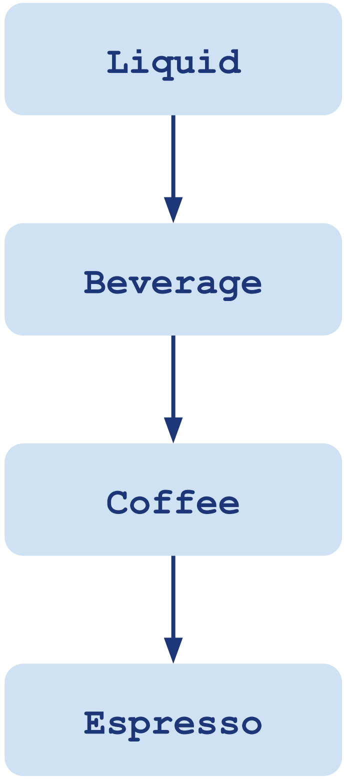


Polymorphism

Polymorphism is the ability for an object to take on different forms. Like how an Espresso is an Espresso, but also a Coffee.

```
Espresso espresso = new Espresso();
```

```
Coffee espresso = new Espresso();
```

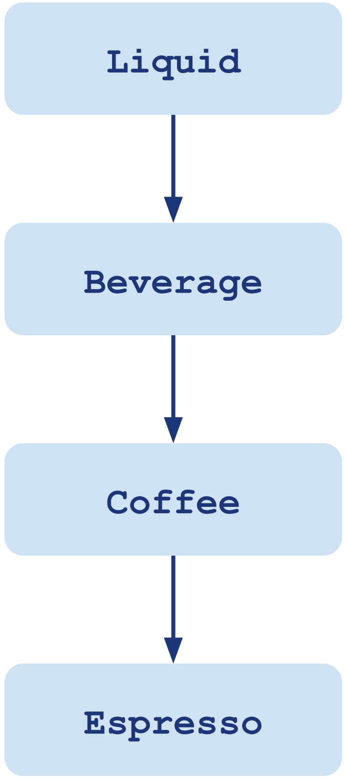


Polymorphism: Methods

RunTime Polymorphism is when method overriding is used to redefine a method with the same method signature from a parent class in your child class. Also known as ***dynamic polymorphism***.

```
Espresso espresso = new Espresso();
espresso.whatAmI(); // prints: I am an
                    // espresso
```

```
Coffee espresso = new Espresso();
espresso.whatAmI(); // prints: I am an
                    // espresso
```

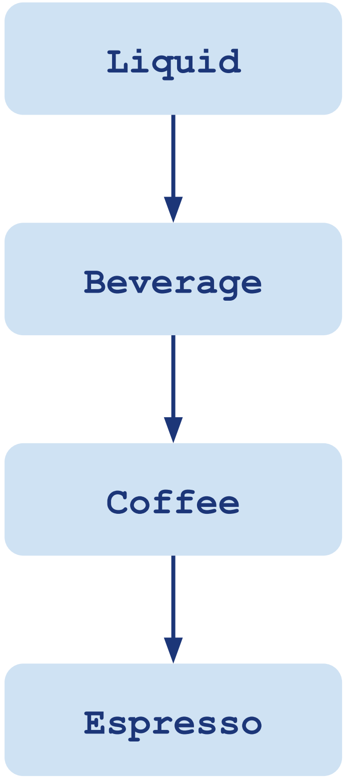


Note: Can’t override static methods.

Polymorphism: Methods

Only time you won't be able to override a method is if it is **final** or you try to override a **static** method.

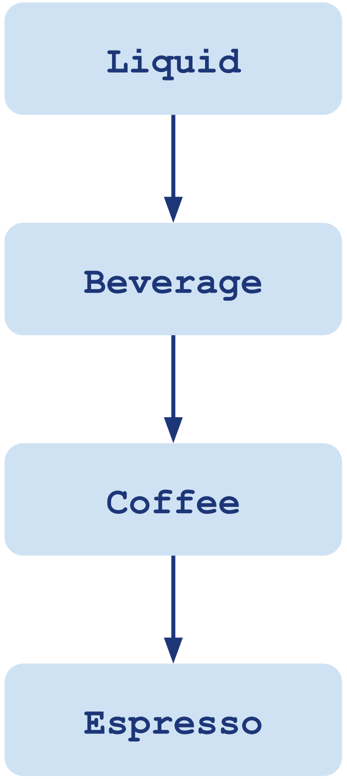
```
public class Espresso {
    // methods cannot be overridden
    public final void whatAmI() {
        ...
    }
    public static void hello() {
        ...
    }
}
```



Polymorphism: Methods

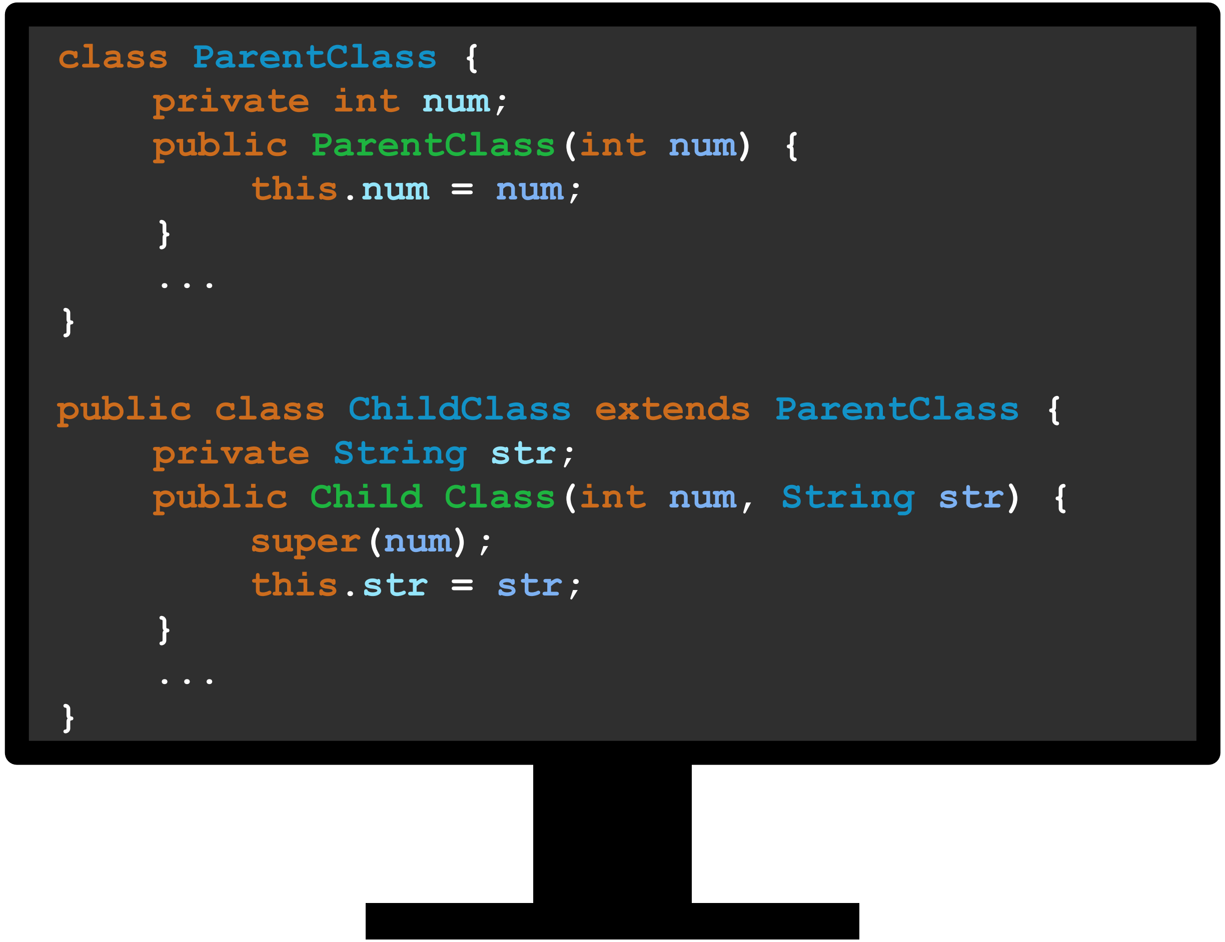
Compile Time Polymorphism is when method overloading is used to define multiple methods with the same name, but different parameters. Also known as ***static polymorphism***.

```
Espresso espresso = new Espresso();

espresso.whatAmI();        // prints: I am an
                           // espresso

espresso.whatAmI("Sam"); // prints: I am an
                         // espresso prepared
                         // by Sam

espresso.whatAmI(3);     // prints: I am 3
                         // espressos
```



Super Keyword

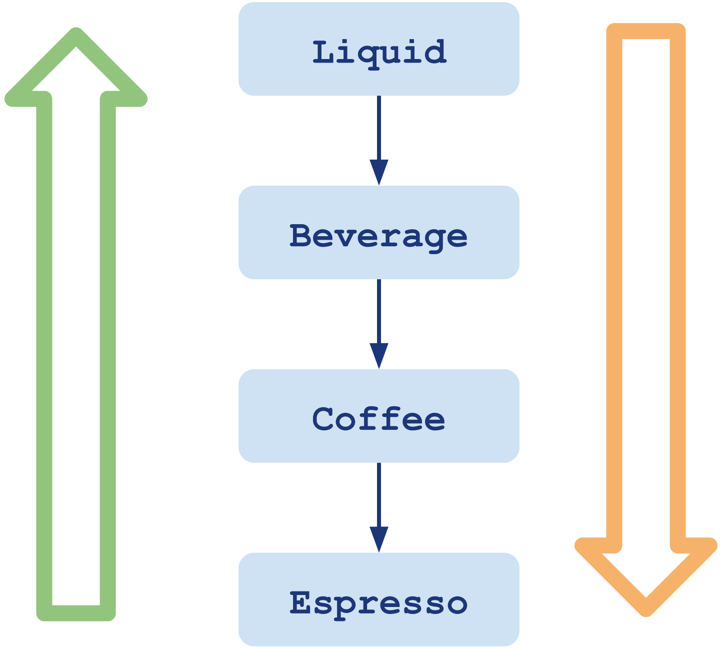
- The super keyword references the superclass of a class
- Super can be used to
  - call constructor of the parent
  - access data members of parent class



```
class ParentClass {
     private int num;
     public ParentClass(int num) {
          this.num = num;
     }
     ...
}

public class ChildClass extends ParentClass {
     private String str;
     public Child Class(int num, String str) {
          super(num);
          this.str = str;
     }
     ...
}
```

Casting Between Types

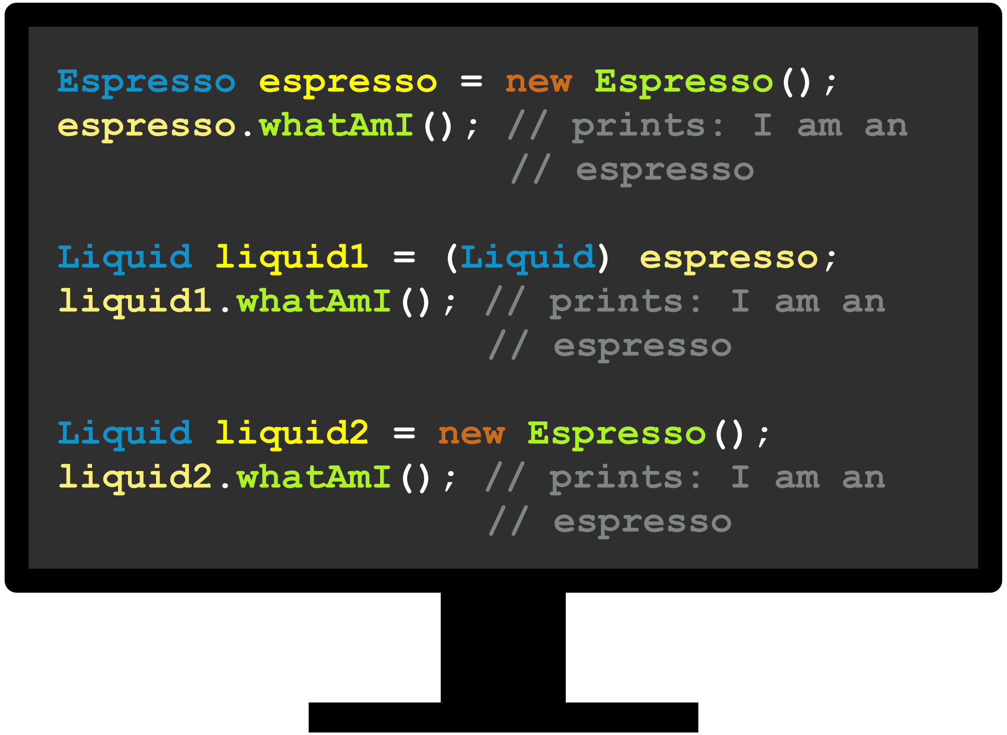
We **Upcast** to convert the type from that of a child class to the type of its parent or any of the classes it inherits from along the chain of inheritance.



We **Downcast** by converting the type from a parent class to a child class or any class that inherits from the original parent.

UpCasting

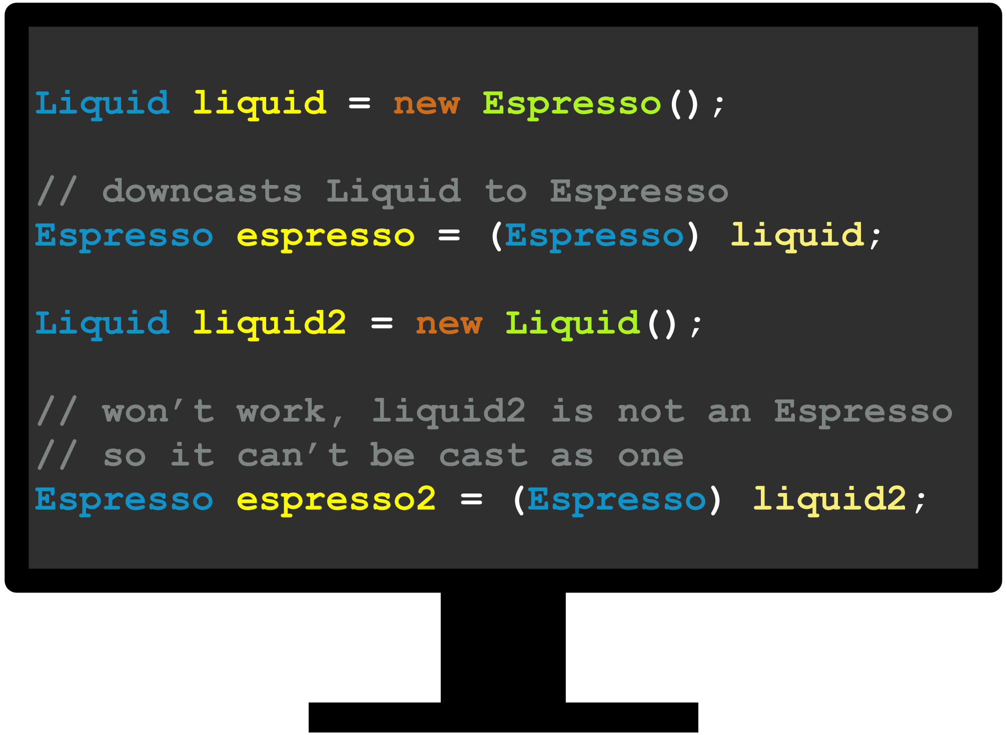
- Can cast by...
  - Explicitly casting with parenthesis and type specified
  - Initializing the object with new keyword
- Why upcast?
  - Want to write code that deals with only supertype



```
Espresso espresso = new Espresso();
espresso.whatAmI(); // prints: I am an
                    // espresso

Liquid liquid1 = (Liquid) espresso;
liquid1.whatAmI(); // prints: I am an
                   // espresso

Liquid liquid2 = new Espresso();
liquid2.whatAmI(); // prints: I am an
                   // espresso
```

DownCasting

- ➔ Can cast by...
  - ◆ Explicitly casting with parenthesis and type specified
- ➔ Why downcast?
  - ◆ Want to access specific behaviors of a subtype

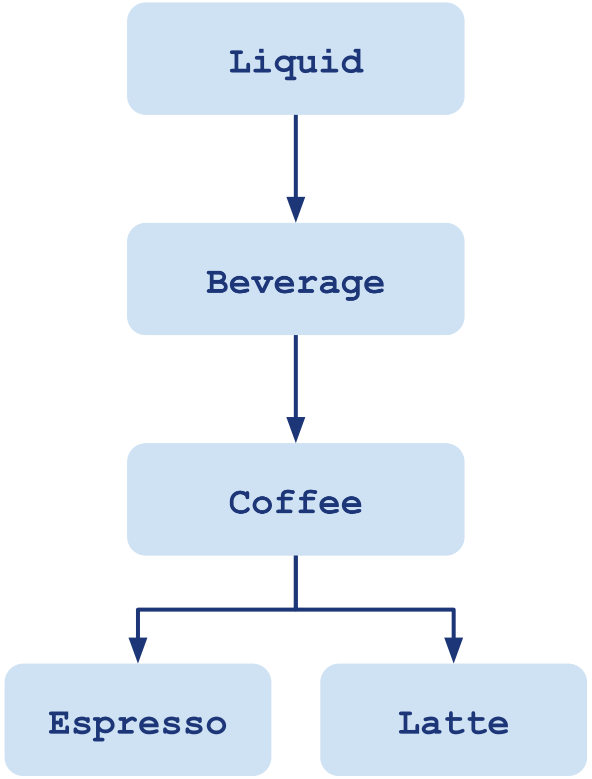


```
Liquid liquid = new Espresso();

// downcasts Liquid to Espresso
Espresso espresso = (Espresso) liquid;

Liquid liquid2 = new Liquid();

// won't work, liquid2 is not an Espresso
// so it can't be cast as one
Espresso espresso2 = (Espresso) liquid2;
```

Instance of an Object

The **instanceof** keyword checks if an object is of a given type and returns back true or false. Checks ***is-a relationship***.

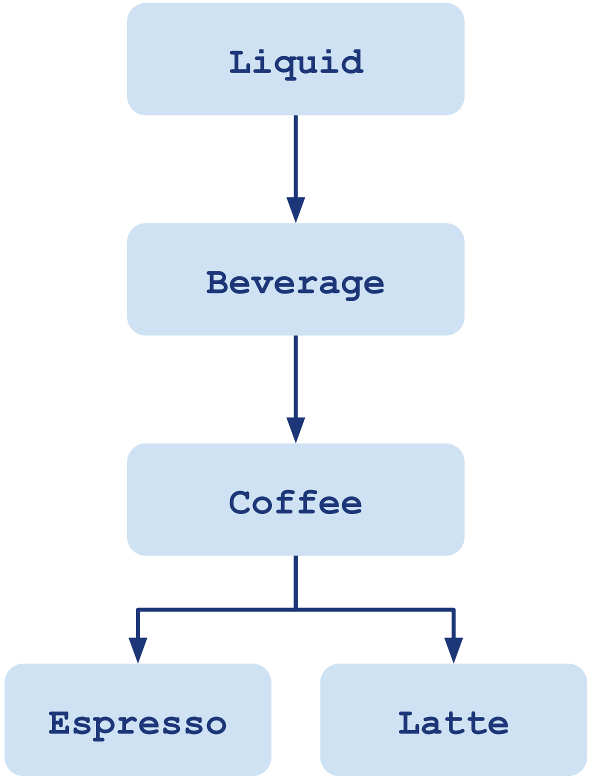
```
Espresso expr = new Espresso();

if ( expr instanceof Espresso ){ // will print
      System.out.println("expr is an Espresso");
}
if ( expr instanceof Liquid ){ // will print
      System.out.println("expr is a Liquid");
}
if ( expr instanceof Latte ){ // compile error
      System.out.println("expr is a String");
}
```



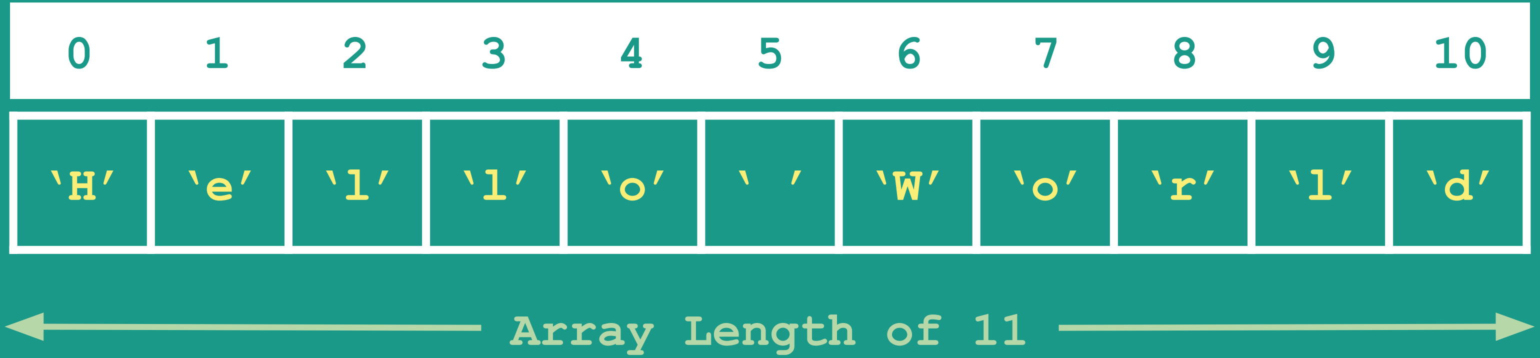
Instance of an Object

The instanceof keyword will return false if object not of the type or if the object is null.

```java
Coffee cof1 = new Coffee();
Coffee cof2 = null;

if ( cof1 instanceof Espresso ){ // won't print
     System.out.println("expr is an Espresso");
}
if ( cof2 instanceof Coffee ){ // won't print
     System.out.println("expr is a Liquid");
}
```


$$\begin{tikzcd} & \text{Liquid} \arrow[d] & \\ & \text{Beverage} \arrow[d] & \\ & \text{Coffee} \arrow[dl] \arrow[dr] & \\ \text{Espresso} & & \text{Latte} \end{tikzcd}$$

Arrays

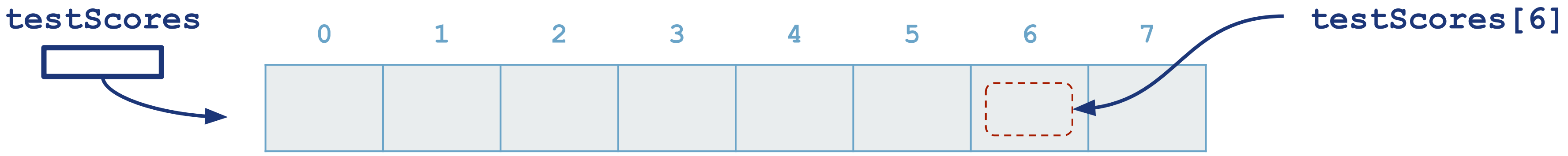


Arrays

```
<data type>[] <variable>;
<variable> = new <data type> [size];
```

```
double[] testScores;
testScores = new double[8];
```

- ➔ An **array** is a collection/group of the same variable types
- ➔ Like an object, must be declared, then have space allocate for it
- ➔ Once the size of an array is set, cannot be changed
- ➔ Each element in array found by its index

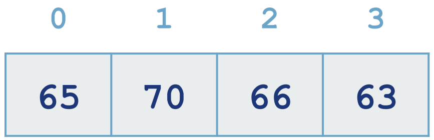


Arrays Cont.

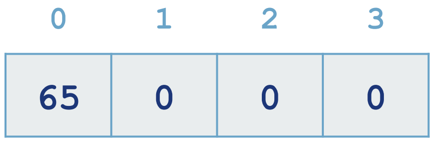
- ➔ Arrays can also be declared and initialized at the same time
- ➔ Newly declared arrays with no values initialized will be set to default values
  - ◆ number types ➝ zero
  - ◆ char ➝ single space
  - ◆ boolean ➝ false
- ➔ The **length** from an array is a final variable set when array initialized

```
temperatures.length  →  4
```

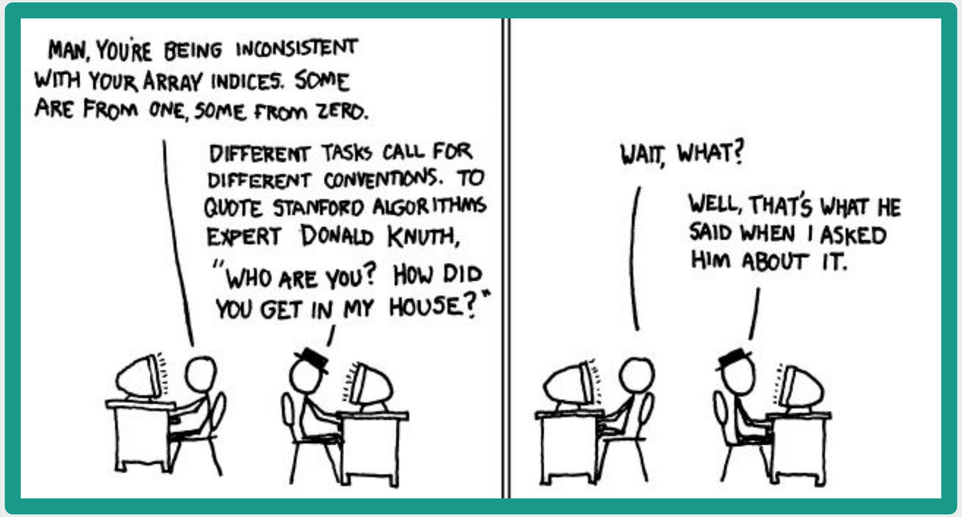
```
int[] temperatures = {65, 70, 66, 63};
```



```
int[] temperatures = new int[4];
temperatures[0] = 65;
```



White board exercise


MAN, YOU'RE BEING INCONSISTENT WITH YOUR ARRAY INDICES. SOME ARE FROM ONE, SOME FROM ZERO.
DIFFERENT TASKS CALL FOR DIFFERENT CONVENTIONS. TO QUOTE STANFORD ALGORITHMS EXPERT DONALD KNUTH, "WHO ARE YOU? HOW DID YOU GET IN MY HOUSE?"
WAIT, WHAT?
WELL, THAT'S WHAT HE SAID WHEN I ASKED HIM ABOUT IT.

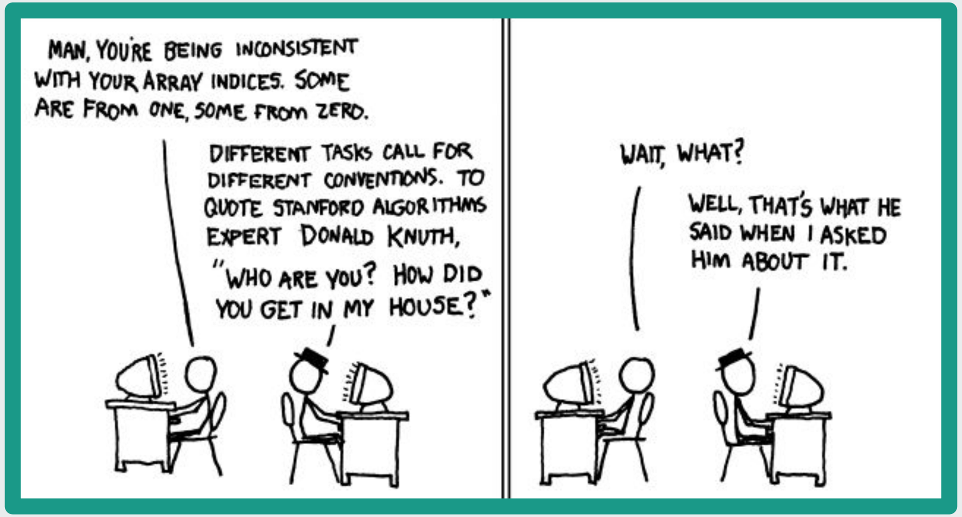
Exercise: Assume you have two arrays of the same data type. You need to check if the two arrays match. The values do not have to be in the same order to match.

✓

```
{1, 3, 5, 0} = {0, 5, 1, 3}
```

✗

```
{ 3, 5, 4, 0} = {0, 5, 1, 3}
```


MAN, YOU'RE BEING INCONSISTENT WITH YOUR ARRAY INDICES. SOME ARE FROM ONE, SOME FROM ZERO.
DIFFERENT TASKS CALL FOR DIFFERENT CONVENTIONS. TO QUOTE STANFORD ALGORITHMS EXPERT DONALD KNUTH, "WHO ARE YOU? HOW DID YOU GET IN MY HOUSE?"
WAIT, WHAT?
WELL, THAT'S WHAT HE SAID WHEN I ASKED HIM ABOUT IT.

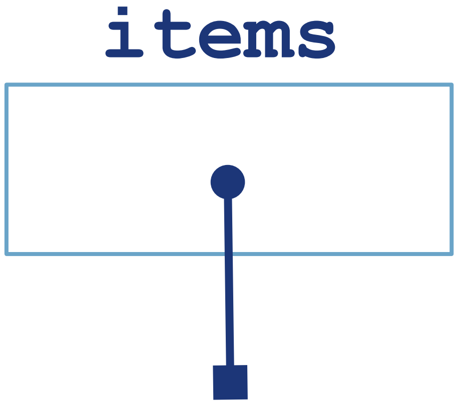
Exercise: There is an integer array with values from 1 to 100, but there is one number missing. Find that missing value.

```
97 is
missing
```

```
{1, 2, 3, 4...96, 98, 99, 100}
```

Arrays of objects

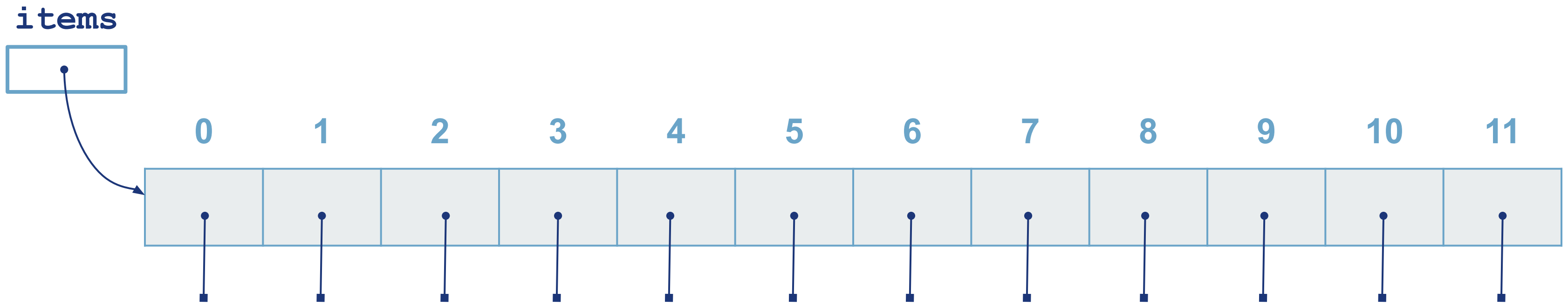
```
Item[] items;

items = new Item[12];

items[0] = new Item();
```



The name items is declared but no allocation has been made for an array.

Arrays of objects

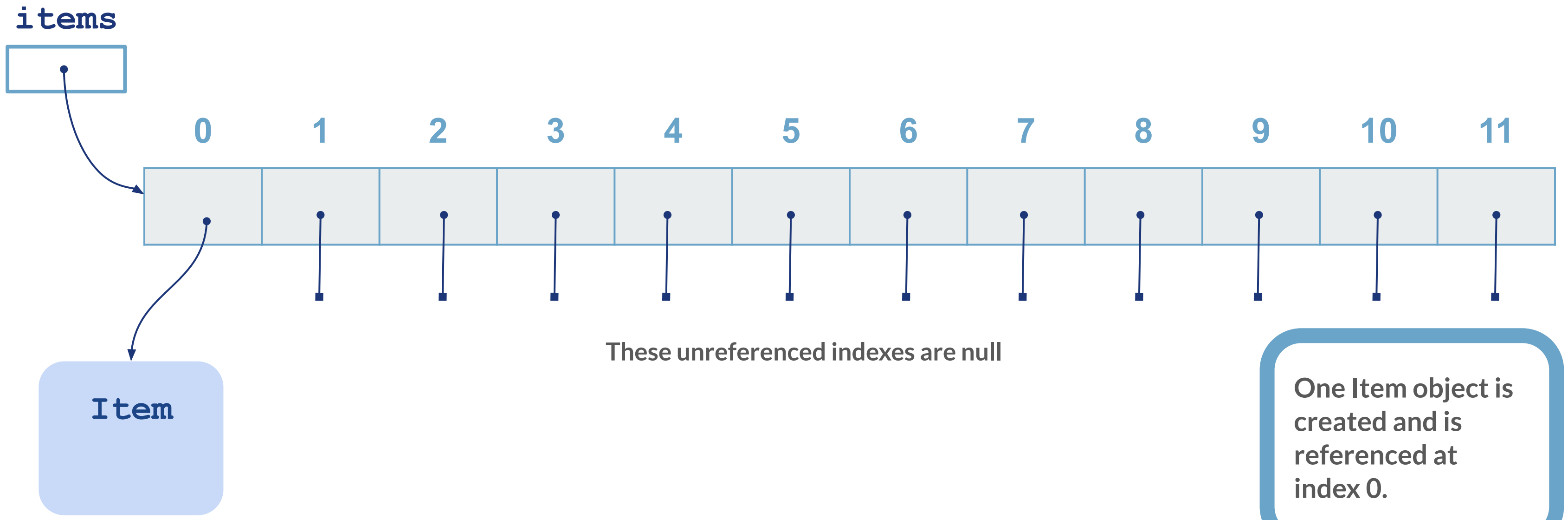
```
Item[] items;

items = new Item[12];

items[0] = new Item();
```



Array allocated for 12 Item objects. No Item objects created yet.

Arrays of objects

```
Item[] items;

items = new Item[12];

items[0] = new Item();
```



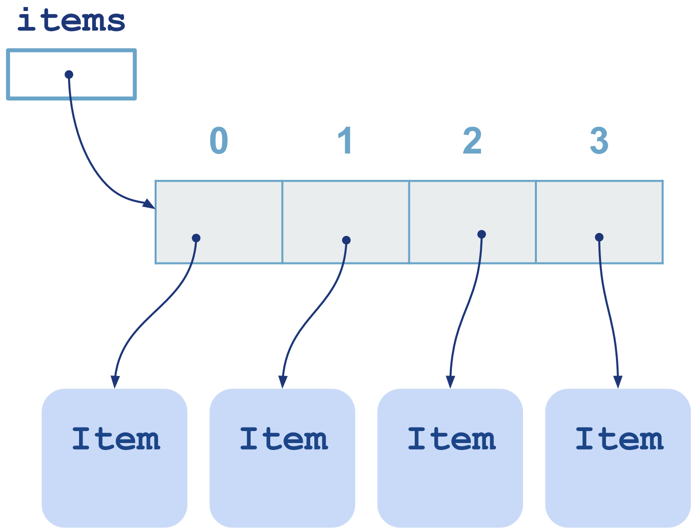
These unreferenced indexes are null

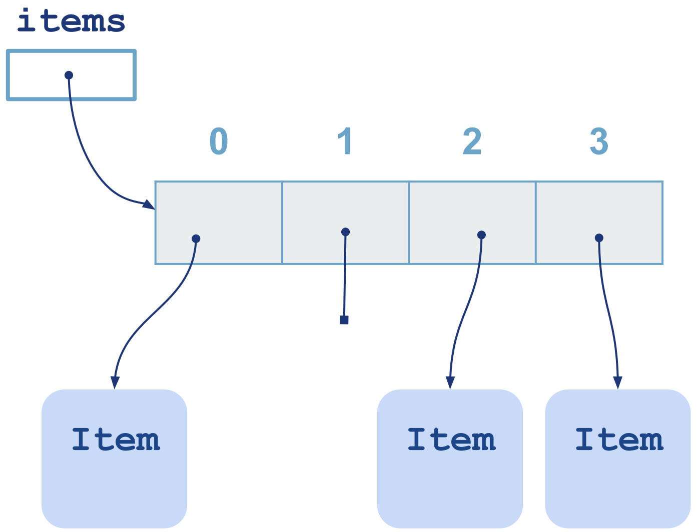
One Item object is created and is referenced at index 0.

Object Deletion

```
Item[] items = new Item[4];
...
items[1] = null;
```

Delete an object by setting it to null.





For Each Loops

```
for(int i = 0; i < items.length; i++) {
      System.out.println(items[i].getName());
}
```

VS

```
for( Item item : items ) {
      System.out.println(item.getName());
}
```

- ➔ Introduced in Java 5
- ➔ Simplifies processing of elements in a collection
- ➔ Constraints:
  - ◆ Read access only (elements can't be changed)
  - ◆ Can only access a single array at a time
  - ◆ Can't skip elements
  - ◆ Can't iterate backwards

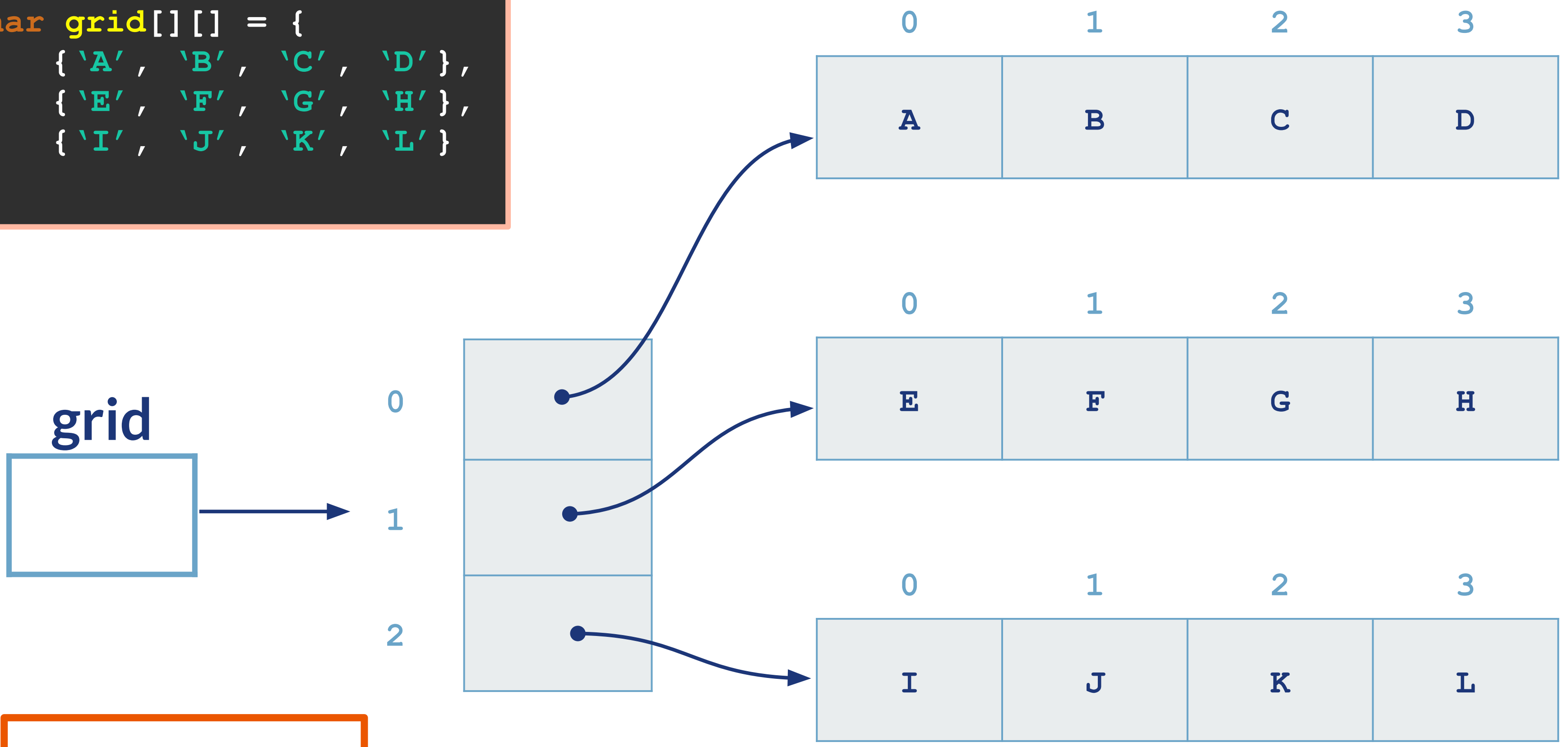
Two-Dimensional Arrays

A **two-dimensional array** is an array that holds a reference to another array in each of its elements.

```
char grid[][] = {
      {‘A’, ‘B’, ‘C’, ‘D’},
      {‘E’, ‘F’, ‘G’, ‘H’},
      {‘I’, ‘J’, ‘K’, ‘L’}
};
```

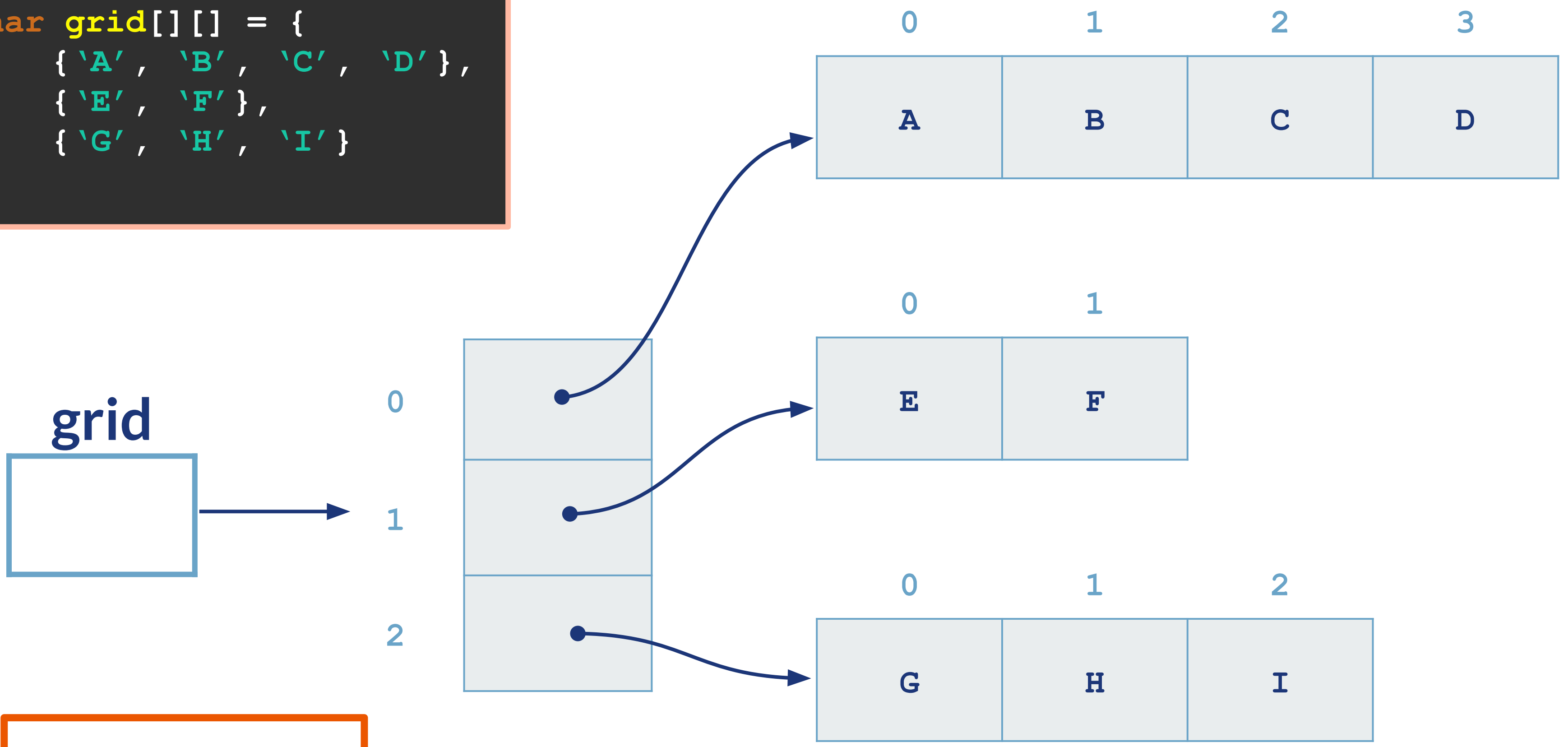
	0	1	2	3
0	A	B	C	D
1	E	F	G	H
2	I	J	K	L

```
char grid[][] = {
     {'A', 'B', 'C', 'D'},
     {'E', 'F', 'G', 'H'},
     {'I', 'J', 'K', 'L'}
};
```



```
arr[1][2] → G
```

```
char grid[][] = {
     {'A', 'B', 'C', 'D'},
     {'E', 'F'},
     {'G', 'H', 'I'}
};
```



```
arr[2][0] → G
```