CSCI 232:
Data Structures and Algorithms

Java Review

Reese Pearsall

Spring 2025

Announcements

- **No lab tomorrow**

- **Quizzes-** You **must** attend the lab that you are registered for
  → If you can’t make it to your lab section, talk to reese beforehand

- Fill out the course questionnaire

We are going to write a program where a user can keep track of their online shopping cart.

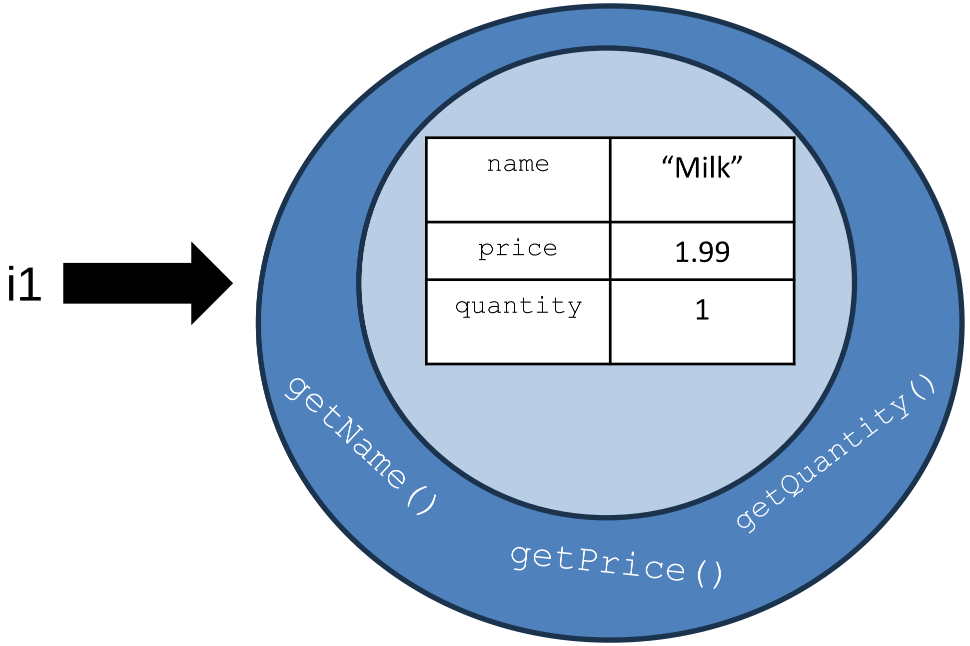
Users can add items, remove items, search for items, get the total price of cart, and apply coupons to items

```java
public class Item {


    private String name;
    private double price;
    private int quantity;

    public Item(String n, double p, int q) {
        this.name = n;
        this.price = p;
        this.quantity = q;
    }

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

    public double getPrice() {
        return this.price;
    }

    public int getQuantity() {
        return this.quantity;
    }
}
```

Java Class: Blueprint for an object (i.e. a “thing”)

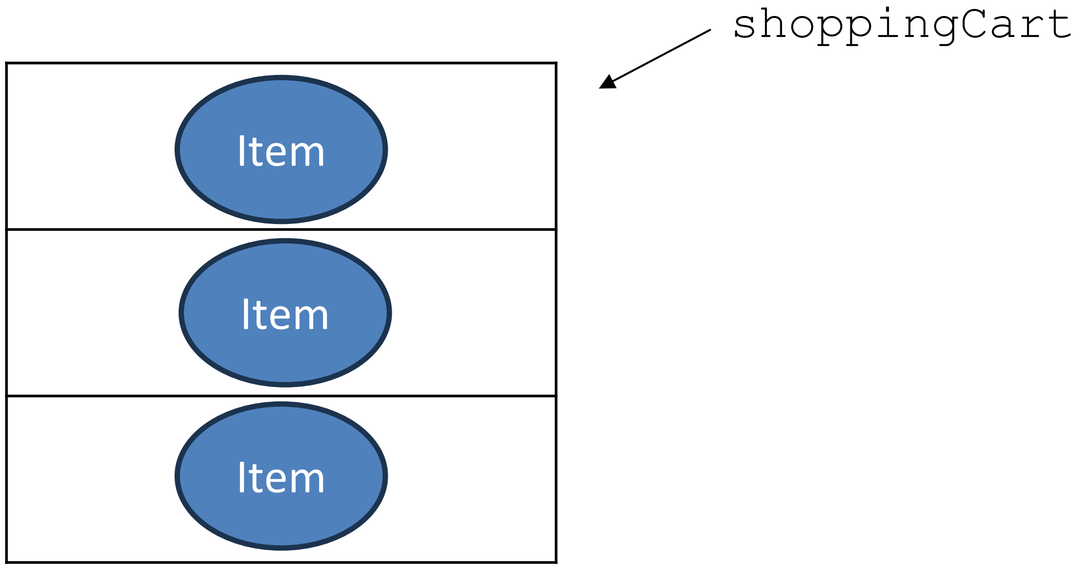
- Instance Field/Attributes
- Methods

```java
Item i1 = new Item("Milk", 1.99, 1);
Item i2 = new Item("Eggs", 3.99, 2);

System.out.println(i1.getName());
System.out.println(i2.getQuantity());
```

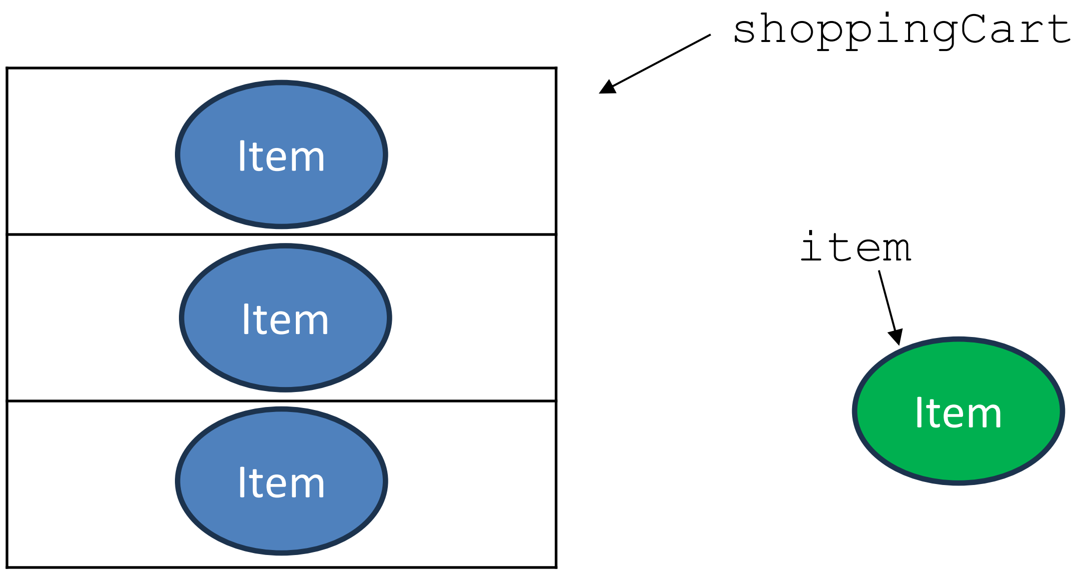


Java Objects: **Instances** of classes.
Program entities

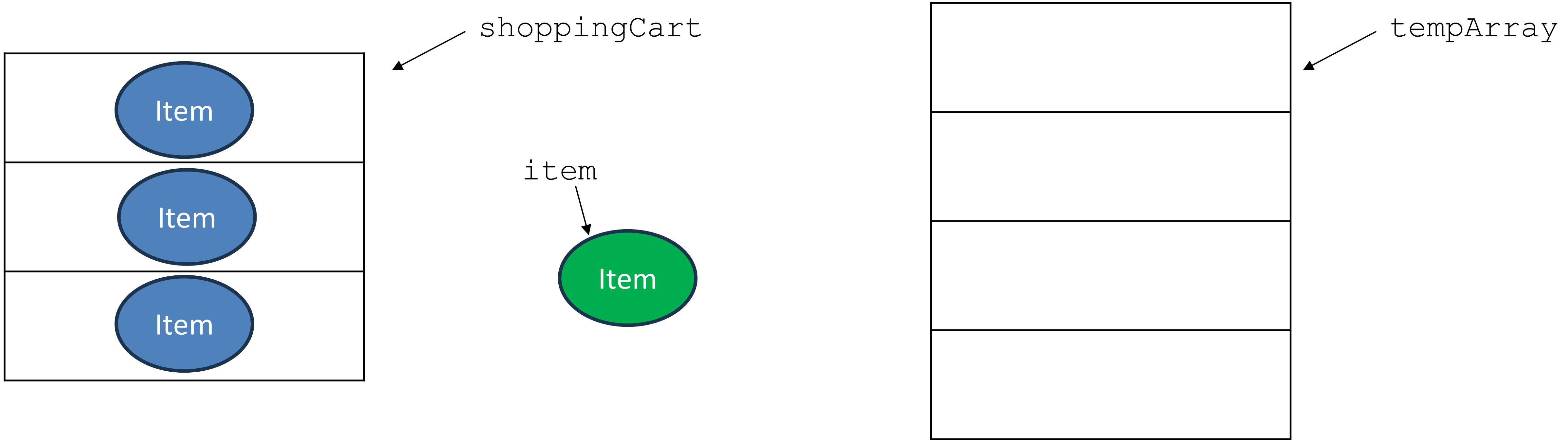
```
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```



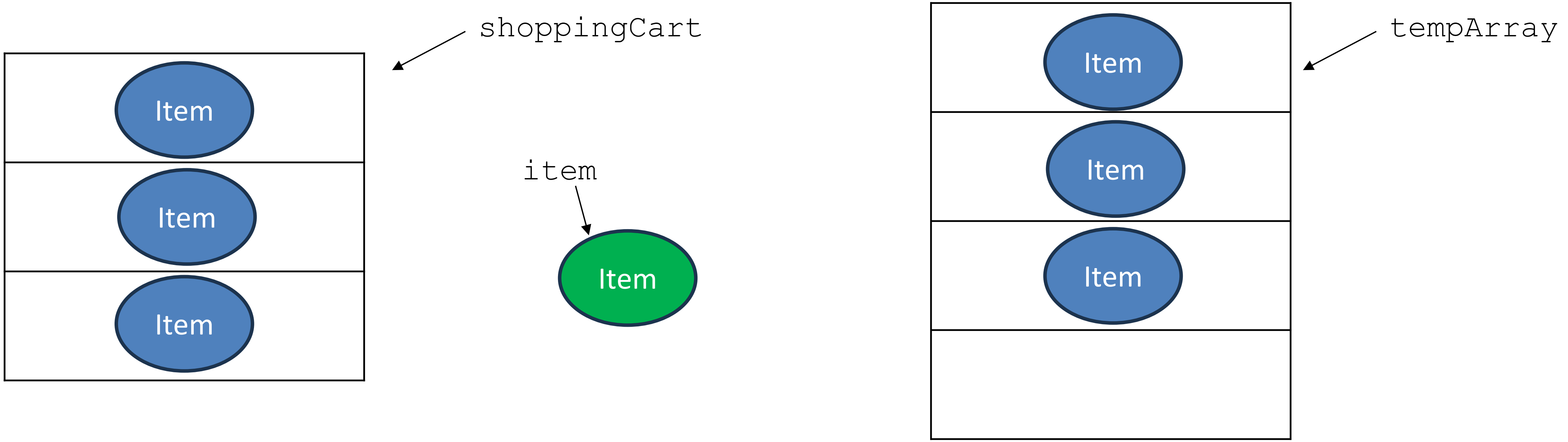
```java
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```



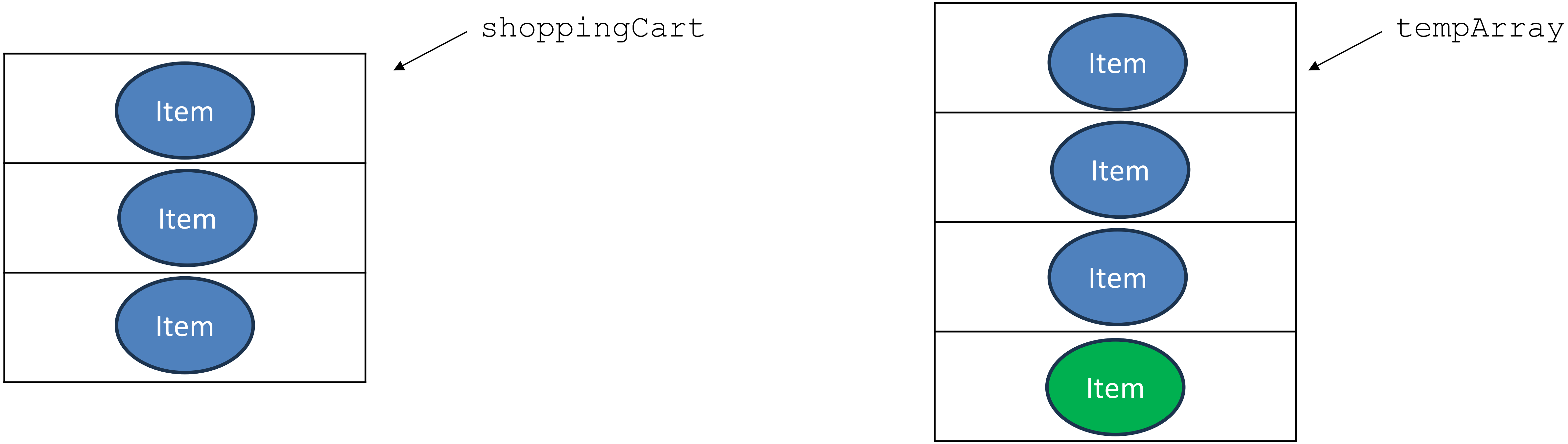
```java
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```



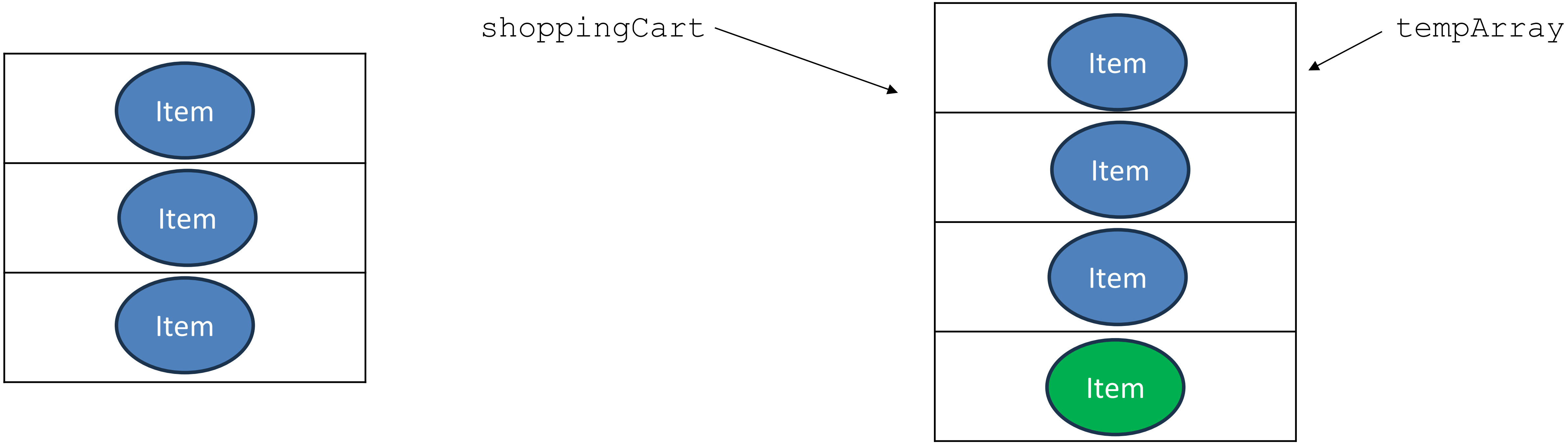
```java
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```



```java
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```



```java
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```



```java
public void addItem(String name, double price, int quantity) {
	Item item = new Item(name, price, quantity);
	Item[] tempArray = new Item[this.shoppingCart.length + 1];
	for(int i = 0; i < this.shoppingCart.length; i++) {
		tempArray[i] = shoppingCart[i];
	}
	tempArray[shoppingCart.length] = item;
	shoppingCart = tempArray;
	this.num_of_items++;
}
```

Running time?

```java
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```

Running time: Number of operations required to complete algorithm

```
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```

Running time: Number of operations required to complete algorithm

Big O Notation: Upper bound on asymptotic growth. I.e. Worst case upper bound of a function

```java
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```

Running time: Number of operations required to complete algorithm

Big O Notation: Upper bound on asymptotic growth. I.e. Worst case upper bound of a function

Big O Notation measures the number of steps needed to complete an algorithm under the worst-case scenario

```java
public int linearSearch(int[] array, int target) {
	for(int i = 0; i < array.length; i++) {
		if(array[i] == target){
			return i;
		}
	}
	return -1;
}
```

To calculate the running time, we add up the running time of each operation

```
public int linearSearch(int[] array, int target) {
 ??? → for(int i = 0; i < array.length; i++) {
            if(array[i] == target){
                return i;
            }
        }
        return -1;
}
```

To calculate the running time, we add up the running time of each operation

Worst case scenario, this for loop will need run **n** times

O(n) **Let n = array.length**

```
public int linearSearch(int[] array, int target) {
  O(n) →for(int i = 0; i < array.length; i++) {
        O(???) → if(array[i] == target){
                    return i;
                }
        }
        return -1;
}
```

To calculate the running time, we add up the running time of each operation

```
public int linearSearch(int[] array, int target) {
  O(n) →for(int i = 0; i < array.length; i++) {
        O(???) → if(array[i] == target){
                        return i;
                }
        }
        return -1;
}
```

To calculate the running time, we add up the running time of each operation

Primitive operation – operation that takes constant time (independent of size of the input)

```
public int linearSearch(int[] array, int target) {
  O(n) →for(int i = 0; i < array.length; i++) {
         O(1) → if(array[i] == target){
                        return i;
                }
        }
        return -1;
}
```

To calculate the running time, we add up the running time of each operation

Primitive operation – operation that takes constant time (independent of size of the input)

```
public int linearSearch(int[] array, int target) {
  O(n) →for(int i = 0; i < array.length; i++) {
          O(1) → if(array[i] == target){
                  O(1) →  return i;
                 }
         }
  O(1) →return -1;
}
```

To calculate the running time, we add up the running time of each operation

Primitive operation – operation that takes constant time (independent of size of the input)

```
public int linearSearch(int[] array, int target) {
  O(n) →for(int i = 0; i < array.length; i++) {
        O(1) → if(array[i] == target){
                O(1) → return i;
               }
        }
  O(1) →return -1;
}
```

To calculate the running time, we add up the running time of each operation

Primitive operation – operation that takes constant time (independent of size of the input)

Total running time: O(n * 1 + 1)

```
public int linearSearch(int[] array, int target) {
  O(n) →for(int i = 0; i < array.length; i++) {
         O(1) → if(array[i] == target){
                O(1) → return i;
                }
         }
  O(1) →return -1;
}
```

To calculate the running time, we add up the running time of each operation

Primitive operation – operation that takes constant time (independent of size of the input)

Total running time: O(n * 1 + 1)

In Big O notation:

- We can drop non dominant factors
- We can drop multiplicative constants (coefficients)

```
public int linearSearch(int[] array, int target) {
  O(n) →for(int i = 0; i < array.length; i++) {
        O(1) → if(array[i] == target){
              O(1) → return i;
              }
        }
  O(1) →return -1;
}
```

To calculate the running time, we add up the running time of each operation

Primitive operation – operation that takes constant time (independent of size of the input)

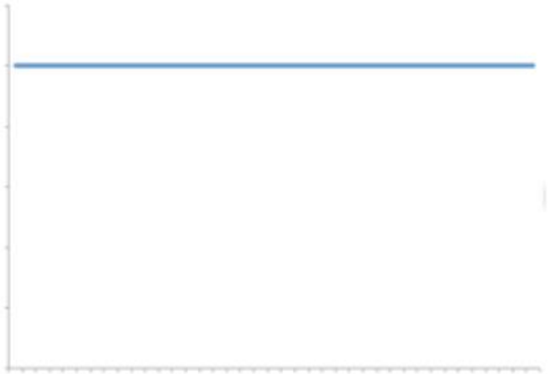
Total running time: O(n) where n = | array |

In Big O notation:

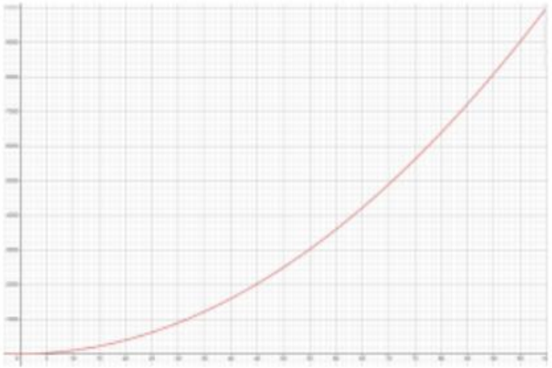
- We can drop non dominant factors
- We can drop multiplicative constants (coefficients)

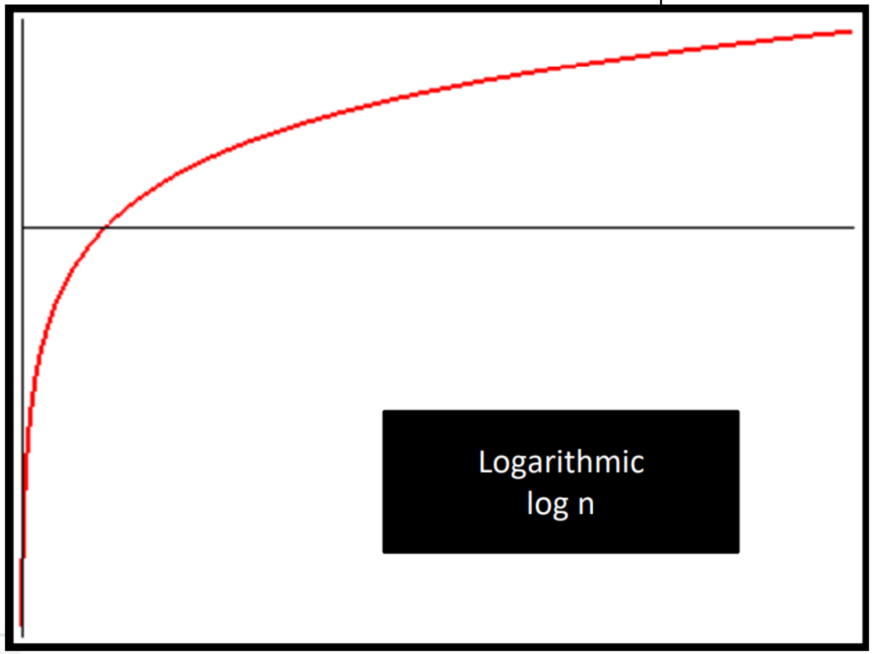
```
{

+ 1];
```

Constant



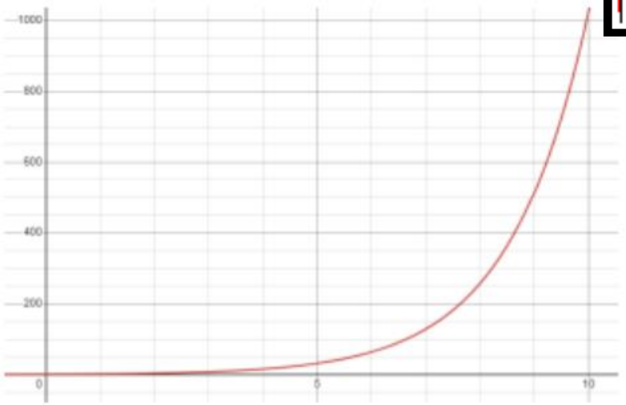
Quadratic





Linear

Exponential



```
function computeDistanceBetweenCaves():

        for each cave in all_caves i;
                for each cave in all_caves j;
                        compute_distance(i, j)
```

	C1	C2	C3	...	C9
C1	/	D(1,2)	D(1,3)	...	D(1,9)
C2	D(2,1)	/	D(2,3)	...	D(2,9)
C3	D(3,1)	D(3,2)	/	...	D(3,9)
...	...	...	...	...	....
C9	D(9,1)	D(9,2)	D(9,3)	...	/

```
function computeDistanceBetweenCaves():

  O(n)  for each cave in all_caves i;
        O(n-1) for each cave in all_caves j;
               O(1) compute_distance(i, j)
```

	C1	C2	C3	…	C9
C1	/	D(1,2)	D(1,3)	…	D(1,9)
C2	D(2,1)	/	D(2,3)	…	D(2,9)
C3	D(3,1)	D(3,2)	/	…	D(3,9)
…	…	…	…	…	….
C9	D(9,1)	D(9,2)	D(9,3)	…	/

```
function computeDistanceBetweenCaves():

O(n) for each cave in all_caves i;
       O(n) for each cave in all_caves j;
              O(1) compute_distance(i, j)
```

	C1	C2	C3	...	C9
C1	/	D(1,2)	D(1,3)	...	D(1,9)
C2	D(2,1)	/	D(2,3)	...	D(2,9)
C3	D(3,1)	D(3,2)	/	...	D(3,9)
...	...	...	...	...	....
C9	D(9,1)	D(9,2)	D(9,3)	...	/

```
function computeDistanceBetweenCaves():

O(n) for each cave in all_caves i;
       O(n) for each cave in all_caves j;
              O(1) compute_distance(i, j)
```

	H1	H2	H3	…	H9
H1	/	D(1,2)	D(1,3)	…	D(1,9)
H2	D(2,1)	/	D(2,3)	…	D(2,9)
H3	D(3,1)	D(3,2)	/	…	D(3,9)
…	…	…	…	…	….
H9	D(9,1)	D(9,2)	D(9,3)	…	/

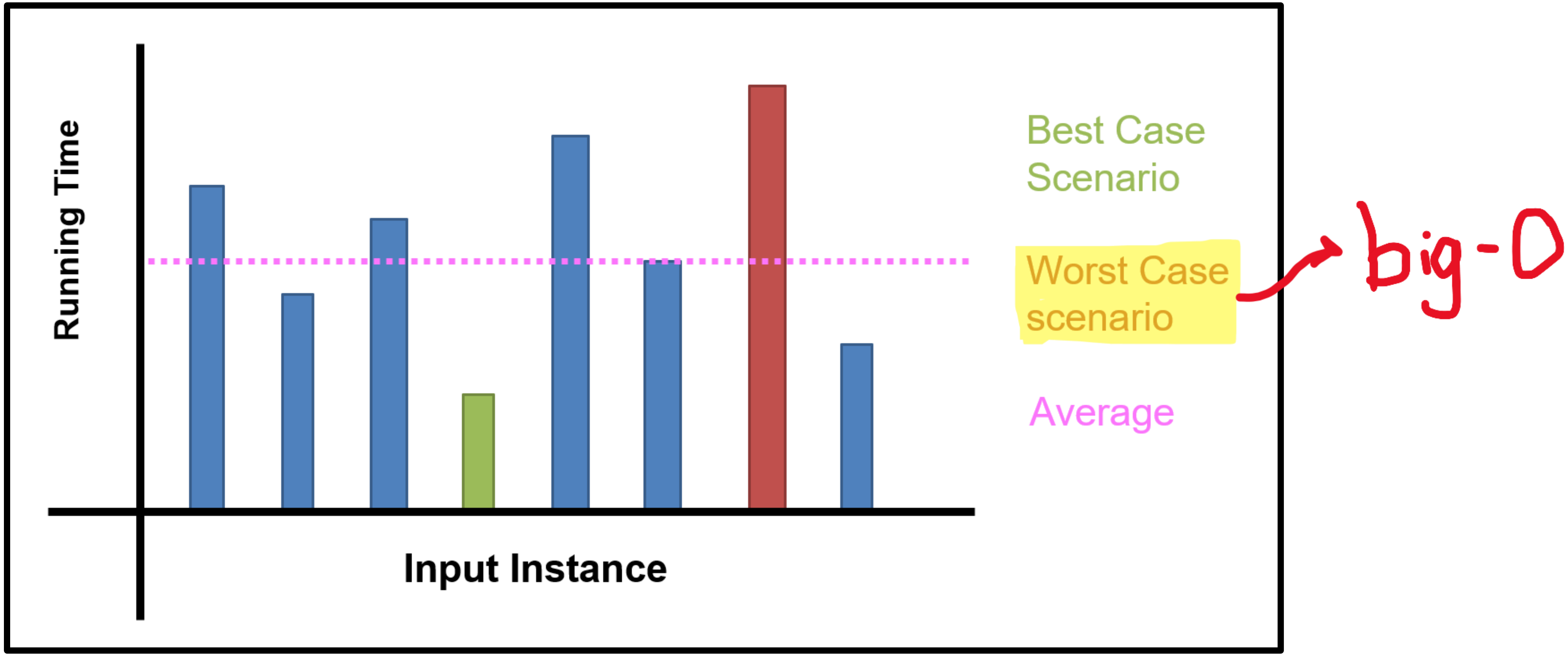
Total running time = O(n) * (O(n) * O(1))

```
function computeDistanceBetweenCaves():

O(n)  for each cave in all_caves i;
          O(n)  for each cave in all_caves j;
                    O(1) compute_distance(i, j)
```

	C1	C2	C3	…	C9
C1	/	D(1,2)	D(1,3)	…	D(1,9)
C2	D(2,1)	/	D(2,3)	…	D(2,9)
C3	D(3,1)	D(3,2)	/	…	D(3,9)
…	…	…	…	…	….
C9	D(9,1)	D(9,2)	D(9,3)	…	/

Total running time = O(n) * (O(n) * O(1))

O(n^2) **Where n = # of caves**



In computer science (and this class in particular), we will be focusing on stating running time in terms of **worst-case scenario**

Big O Formal Definition

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers

$f(n)$ is $\mathbf{O}(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n), \text{ for all } n \geq n_0$$

Big O Formal Definition

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers

$f(n)$ is $\mathbf{O}(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n), \text{ for all } n \geq n_0$$

Past a certain spot, g(n) dominates f(n) within a multiplicative constant

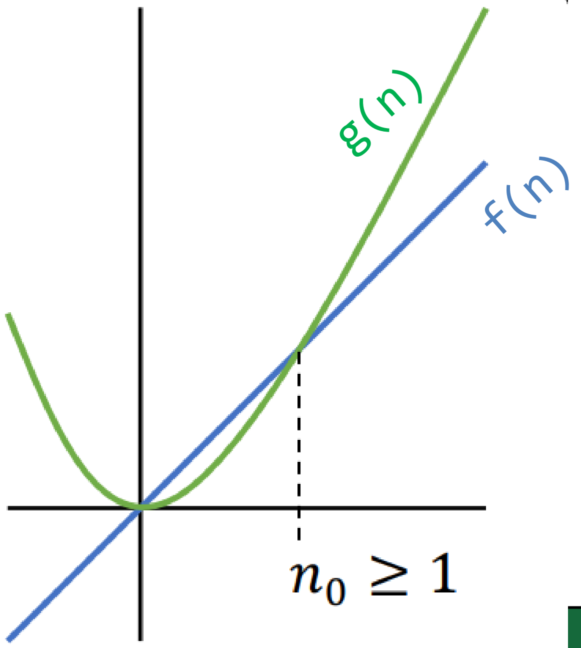
Big O Formal Definition

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers

$f(n)$ is $\mathbf{O}(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n), \text{ for all } n \geq n_0$$

Past a certain spot, g(n) dominates f(n) within a multiplicative constant



$$\forall n \geq 1, n^2 \geq n$$
$$\Rightarrow n \in O(n^2)$$

O -notation provides an upper bound on some function $f(n)$

Which would you rather have?

Given a problem of size *n*

Algorithm **A** runs in $O(n^2)$ time.

Algorithm **B** runs in $O(n)$ time.

Which would you rather have?

Given a problem of size *n*

Algorithm **A** runs in $n^2 \in O(n^2)$ time.

Algorithm **B** runs in $n + 10^{25} \in O(n)$ time.

Which would you rather have?

Given a problem of size *n*

Algorithm **A** runs in $n^2 \in O(n^2)$ time.

Algorithm **B** runs in $n + 10^{25} \in O(n)$ time.

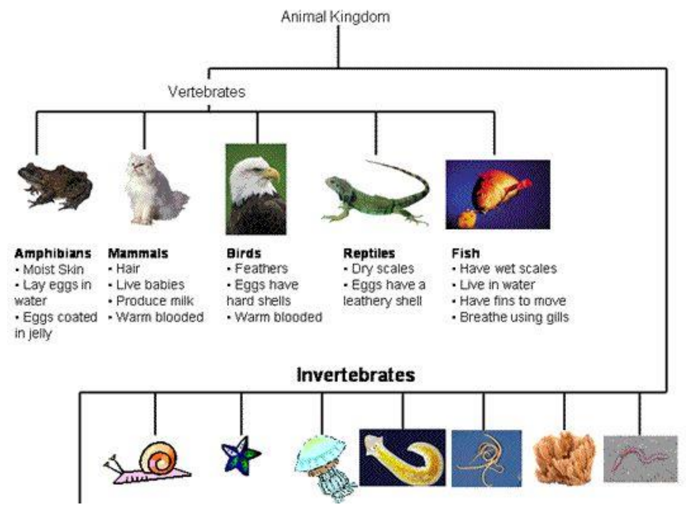
Which would you rather have?

Given a problem of size *n*

Algorithm **A** runs in $n^2 \in O(n^2)$ time.

Algorithm **B** runs in $n + 10^{25} \in O(n)$ time.

Big-O is a helpful way to broadly describe the running time of different programs, but it isn't perfect



```java
public void addItem(String name, double price, int quantity) {
        Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```

```java
public void addItem(String name, double price, int quantity) {
  O(1) → Item item = new Item(name, price, quantity);
        Item[] tempArray = new Item[this.shoppingCart.length + 1];
        for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
        }
        tempArray[shoppingCart.length] = item;
        shoppingCart = tempArray;
        this.num_of_items++;
}
```

```
public void addItem(String name, double price, int quantity) {
  O(1) → Item item = new Item(name, price, quantity);
O(n+1) → Item[] tempArray = new Item[this.shoppingCart.length + 1];
         for(int i = 0; i < this.shoppingCart.length; i++) {
                 tempArray[i] = shoppingCart[i];
         }
         tempArray[shoppingCart.length] = item;
         shoppingCart = tempArray;
         this.num_of_items++;
}
```

```
public void addItem(String name, double price, int quantity) {
  O(1) → Item item = new Item(name, price, quantity);
O(n+1) → Item[] tempArray = new Item[this.shoppingCart.length + 1];
  O(n) → for(int i = 0; i < this.shoppingCart.length; i++) {
                tempArray[i] = shoppingCart[i];
         }
         tempArray[shoppingCart.length] = item;
         shoppingCart = tempArray;
         this.num_of_items++;
}
```

```
public void addItem(String name, double price, int quantity) {
   O(1) → Item item = new Item(name, price, quantity);
 O(n+1) → Item[] tempArray = new Item[this.shoppingCart.length + 1];
   O(n) → for(int i = 0; i < this.shoppingCart.length; i++) {
            O(1) → tempArray[i] = shoppingCart[i];
          }
          tempArray[shoppingCart.length] = item;
          shoppingCart = tempArray;
          this.num_of_items++;
}
```

```
public void addItem(String name, double price, int quantity) {
   O(1) → Item item = new Item(name, price, quantity);
O(n+1) → Item[] tempArray = new Item[this.shoppingCart.length + 1];
  O(n) → for(int i = 0; i < this.shoppingCart.length; i++) {
           O(1) → tempArray[i] = shoppingCart[i];
         }
   O(1) → tempArray[shoppingCart.length] = item;
   O(1) → shoppingCart = tempArray;
   O(1) → this.num_of_items++;
}
```

```
public void addItem(String name, double price, int quantity) {
    O(1) → Item item = new Item(name, price, quantity);
O(n+1) → Item[] tempArray = new Item[this.shoppingCart.length + 1];
  O(n) → for(int i = 0; i < this.shoppingCart.length; i++) {
             O(1) → tempArray[i] = shoppingCart[i];
         }
    O(1) → tempArray[shoppingCart.length] = item;
    O(1) → shoppingCart = tempArray;
    O(1) → this.num_of_items++;
}
```

```
public void addItem(String name, double price, int quantity) {
  O(1) → Item item = new Item(name, price, quantity);
  O(n) → Item[] tempArray = new Item[this.shoppingCart.length + 1];
  O(n) → for(int i = 0; i < this.shoppingCart.length; i++) {
             O(1) → tempArray[i] = shoppingCart[i];
         }
  O(1) → tempArray[shoppingCart.length] = item;
  O(1) → shoppingCart = tempArray;
  O(1) → this.num_of_items++;
}
```

Total running time: O(n) + O(n)

O(2n)

O(n) where n = shoppingCart.length

```
public void addItem(String name, double price, int quantity) {
  O(1) → Item item = new Item(name, price, quantity);
  O(n) → Item[] tempArray = new Item[this.shoppingCart.length + 1];
  O(n) → for(int i = 0; i < this.shoppingCart.length; i++) {
           O(1) → tempArray[i] = shoppingCart[i];
         }
  O(1) → tempArray[shoppingCart.length] = item;
  O(1) → shoppingCart = tempArray;
  O(1) → this.num_of_items++;
}
```

Total running time: O(n) + O(n)

O(2n)

O(n) where n = shoppingCart.length

Takeaway: Adding to a full array takes O(n) time