GEORGE MASON UNIVERSITY

CS 310 – Fall 2025

Data Structures

L02-Review

Archange G. Destiné
adestine@gmu.edu

Review

01. Review – Java Basics (Data Types, OOP, Inheritance, Debugging)

02. Generics

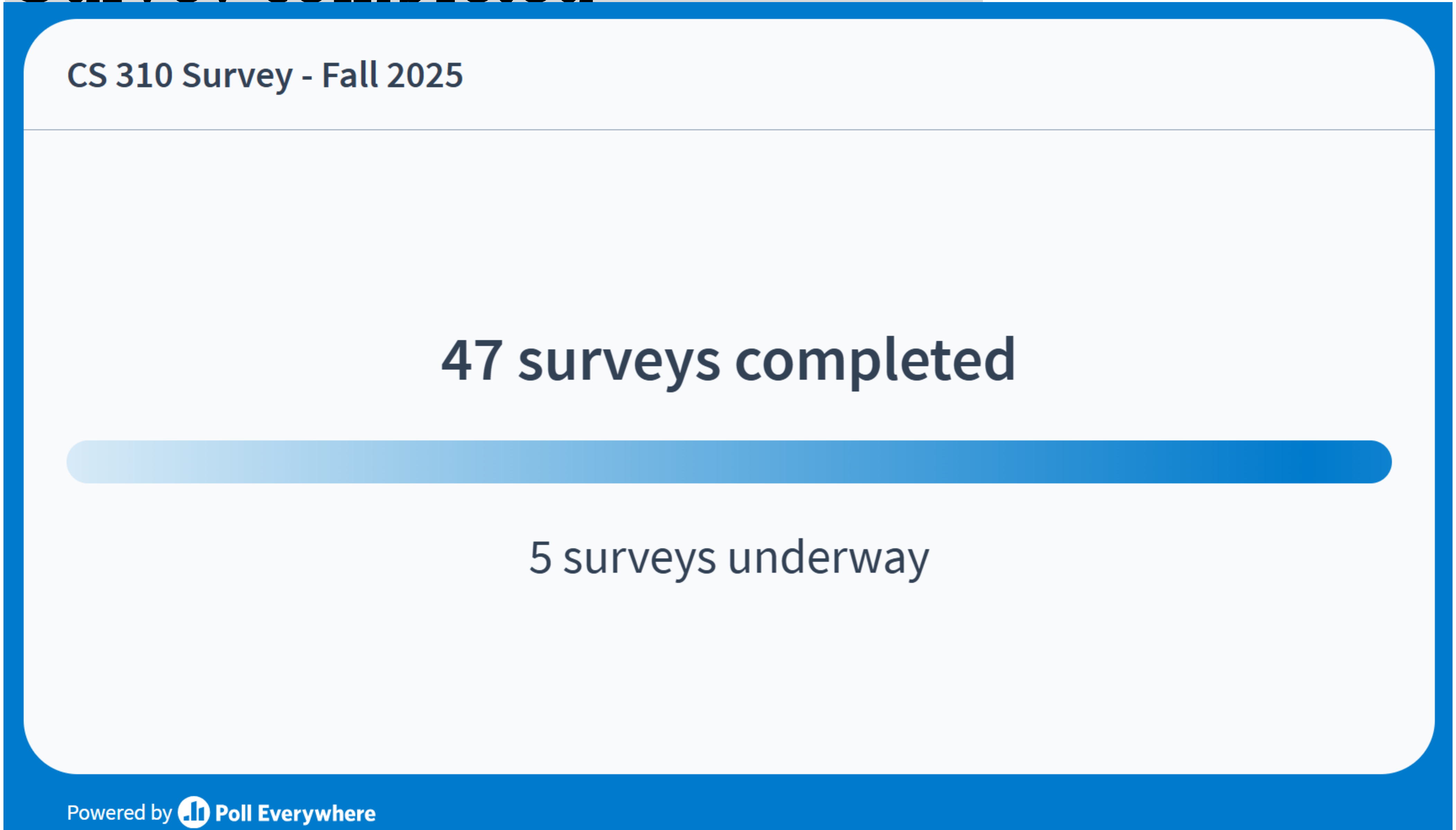
03. Command Line Interface

04. Computational Complexity

Recap from last lecture

- **Where to find** course material?
- How to access to **Piazza**, **Canvas**, **Gradescope**?
- **If you need *any* help**: Piazza, Office Hours, Instructors
- **Overview** of Data Structures: List: Static Array, Dynamic Array, Stack, Queue… also: Tree, Graph, …

Survey completed



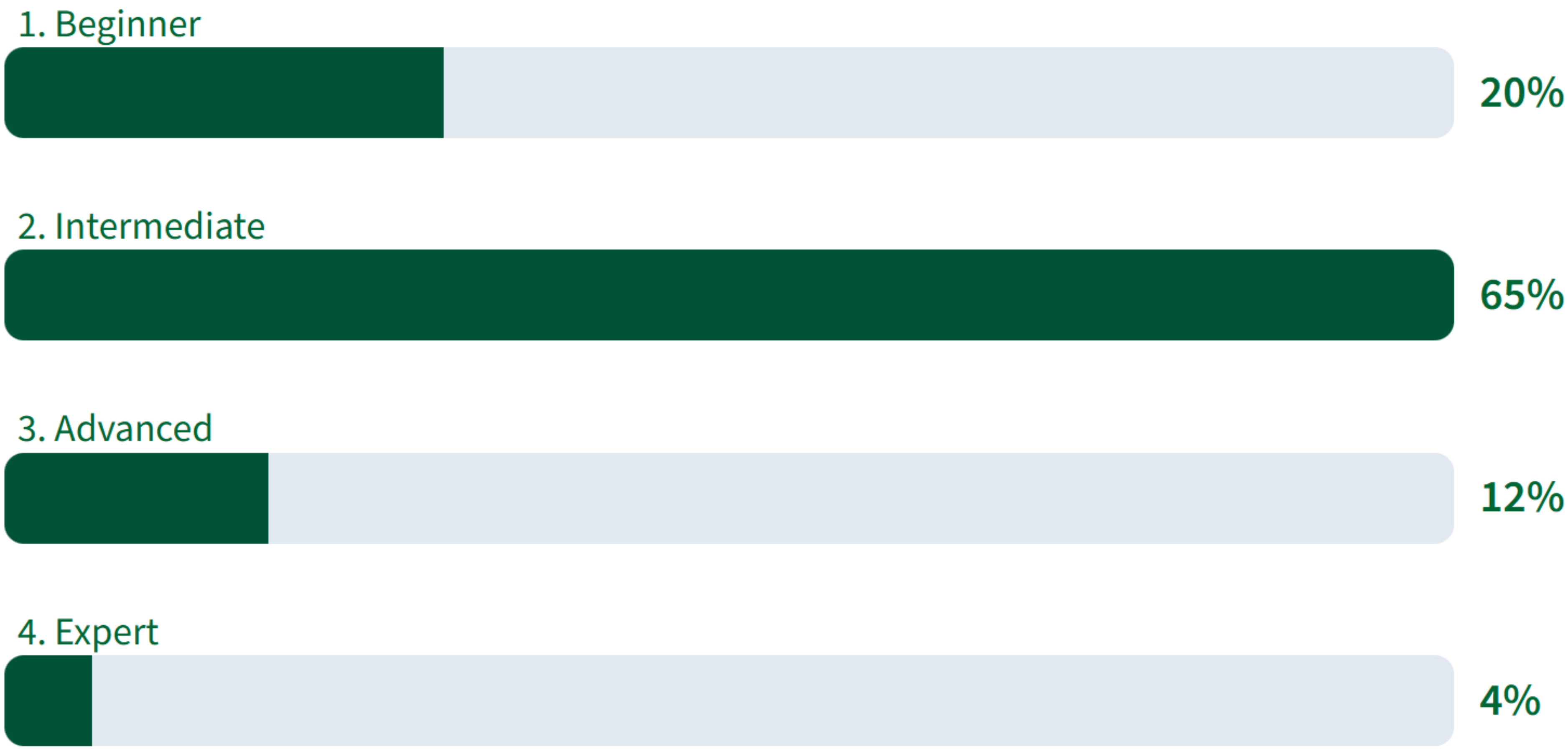
CS 310 Survey - Fall 2025

47 surveys completed

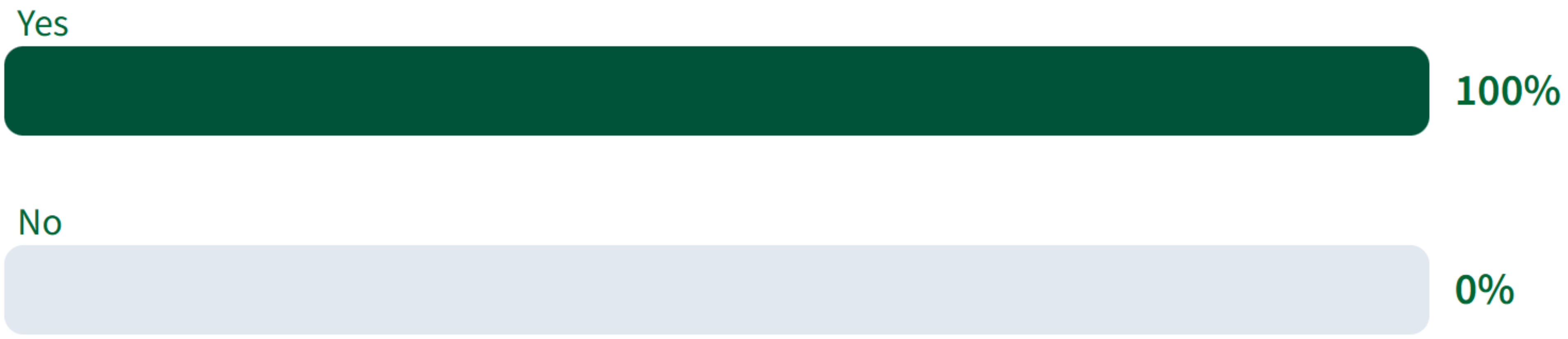
5 surveys underway

Powered by Poll Everywhere

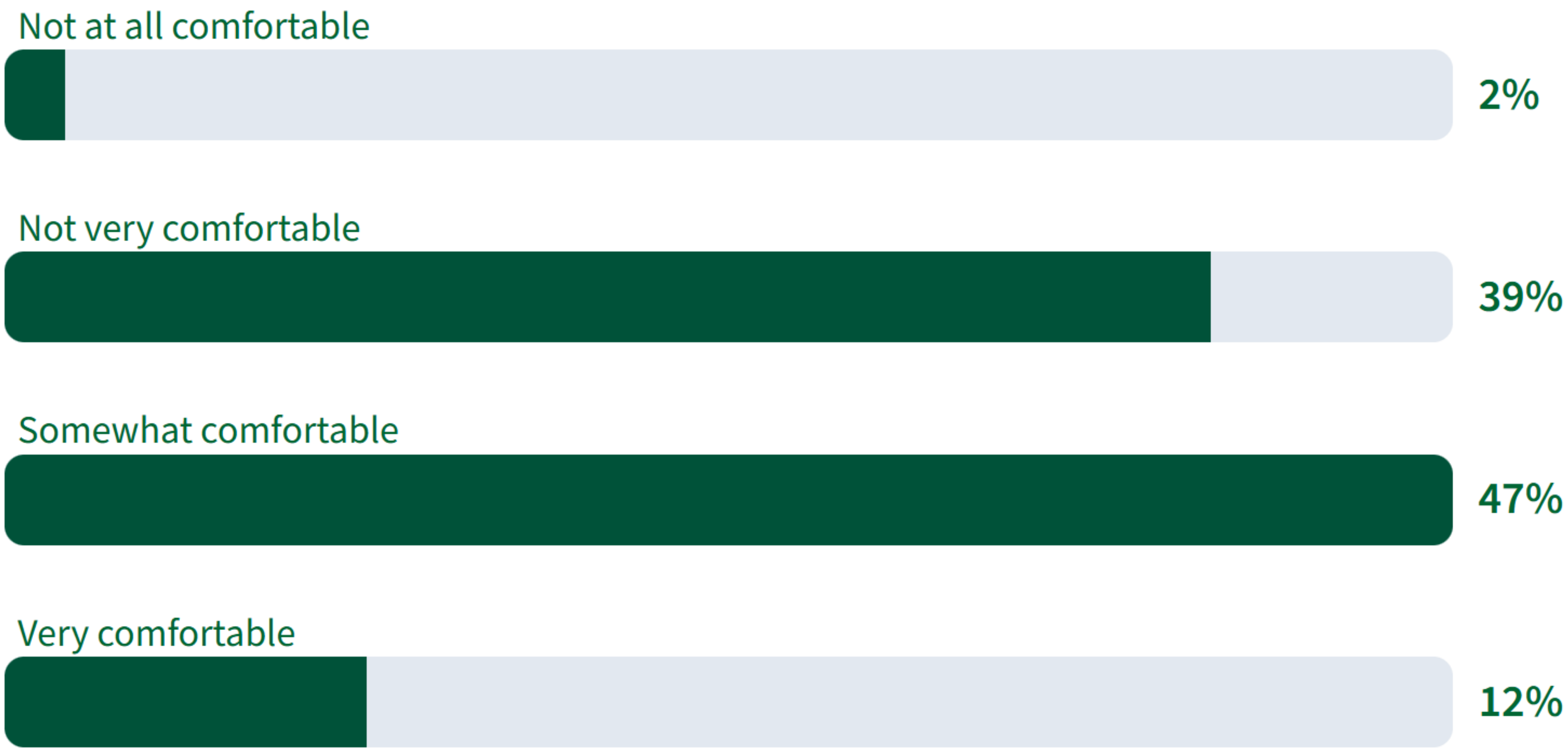
On a scale of 1 to 4, how would you rate your proficiency in Java programming?



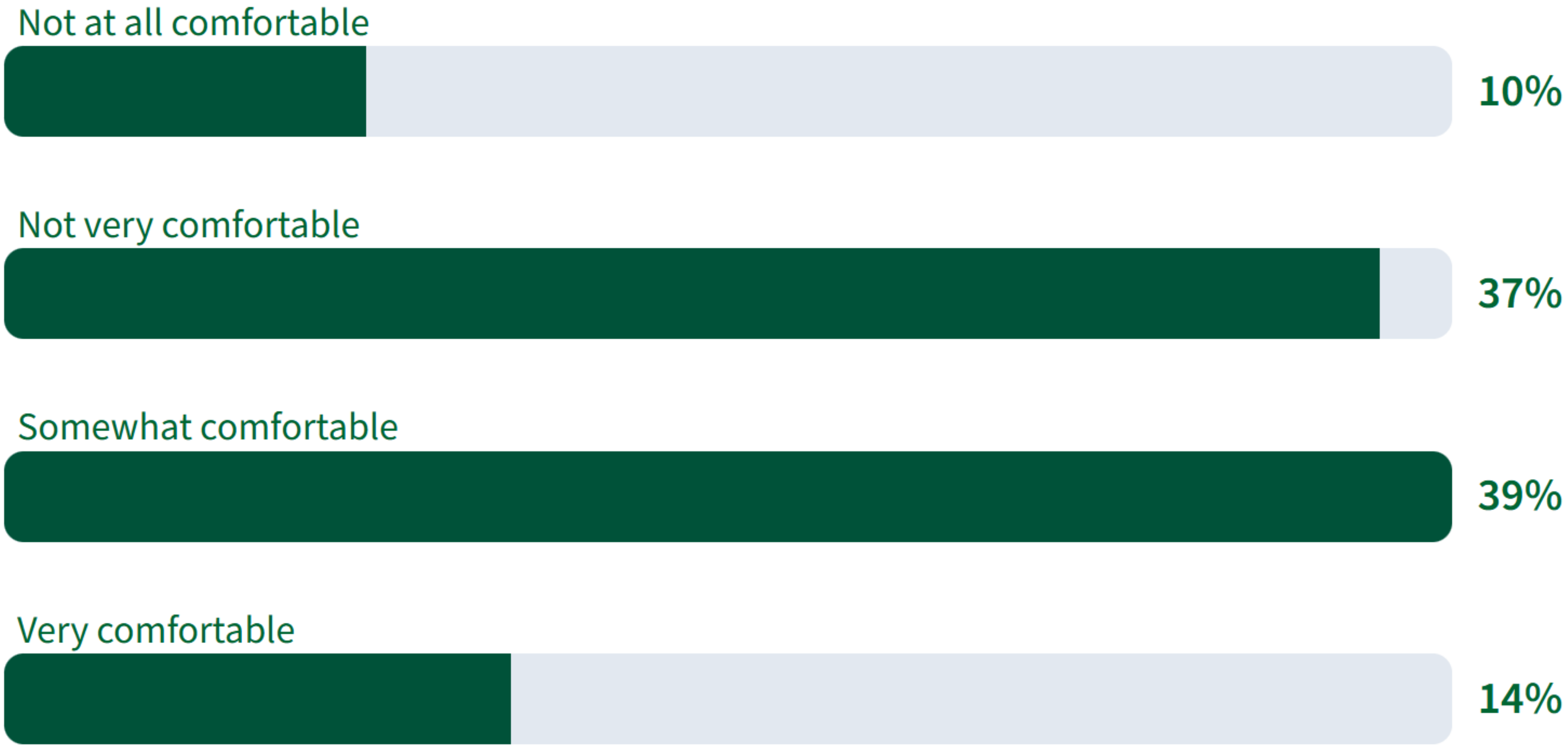
Have you previously taken a course in Object-Oriented Programming using Java?



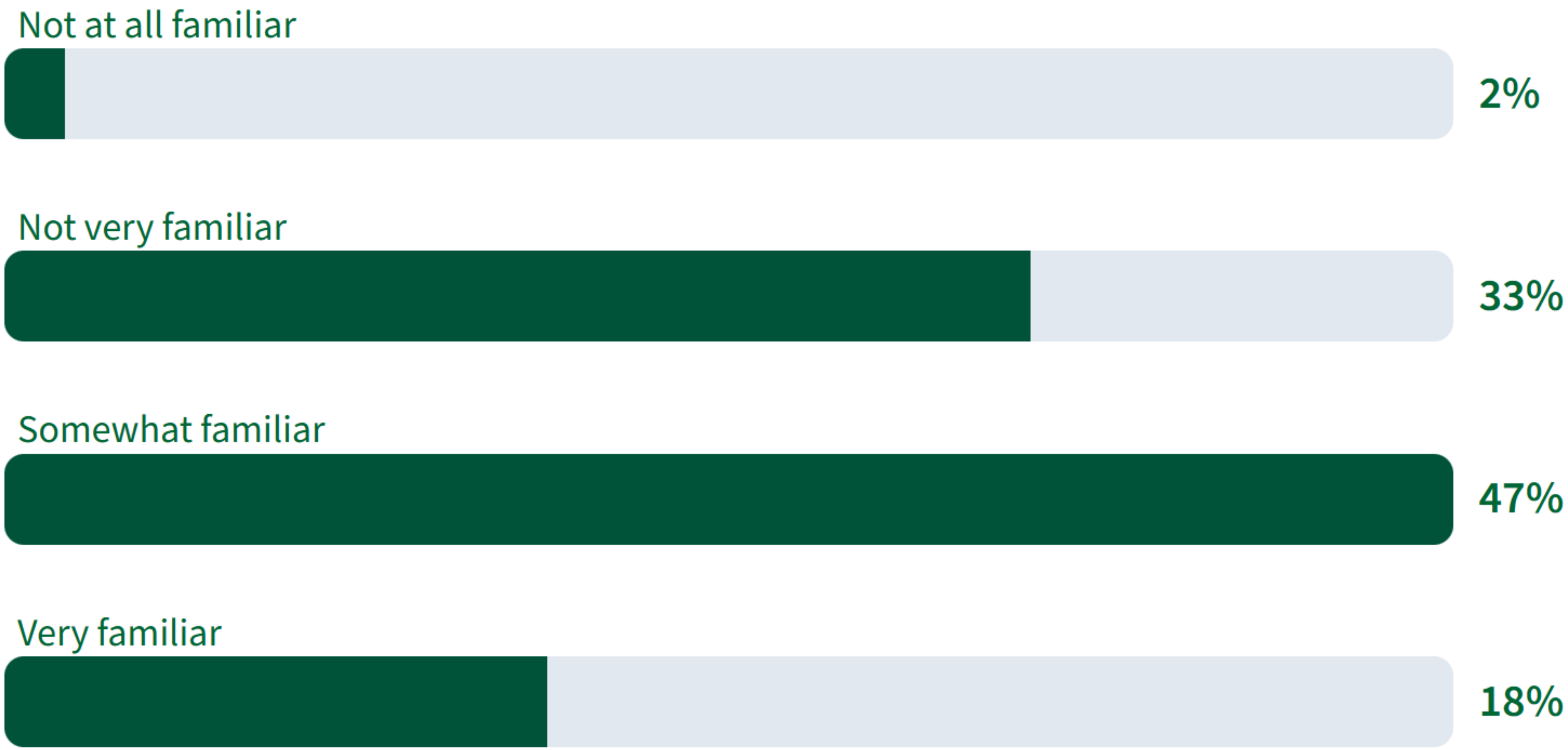
How comfortable are you with debugging Java code?



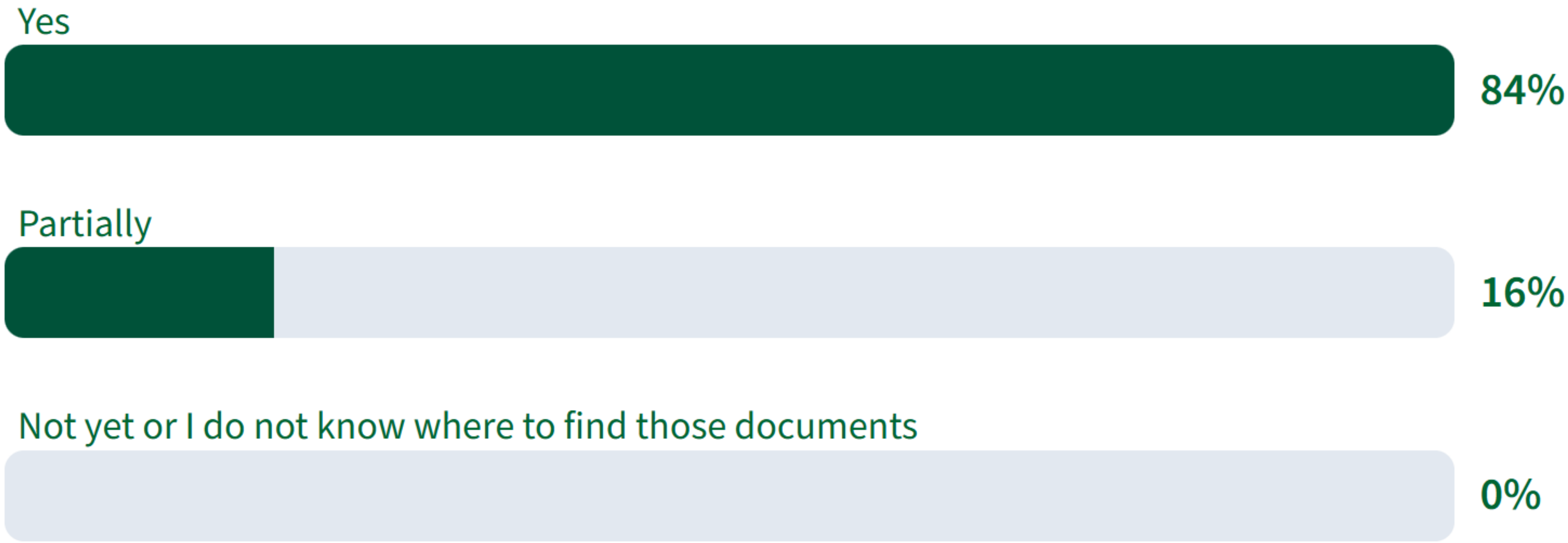
How comfortable are you with writing recursive functions?

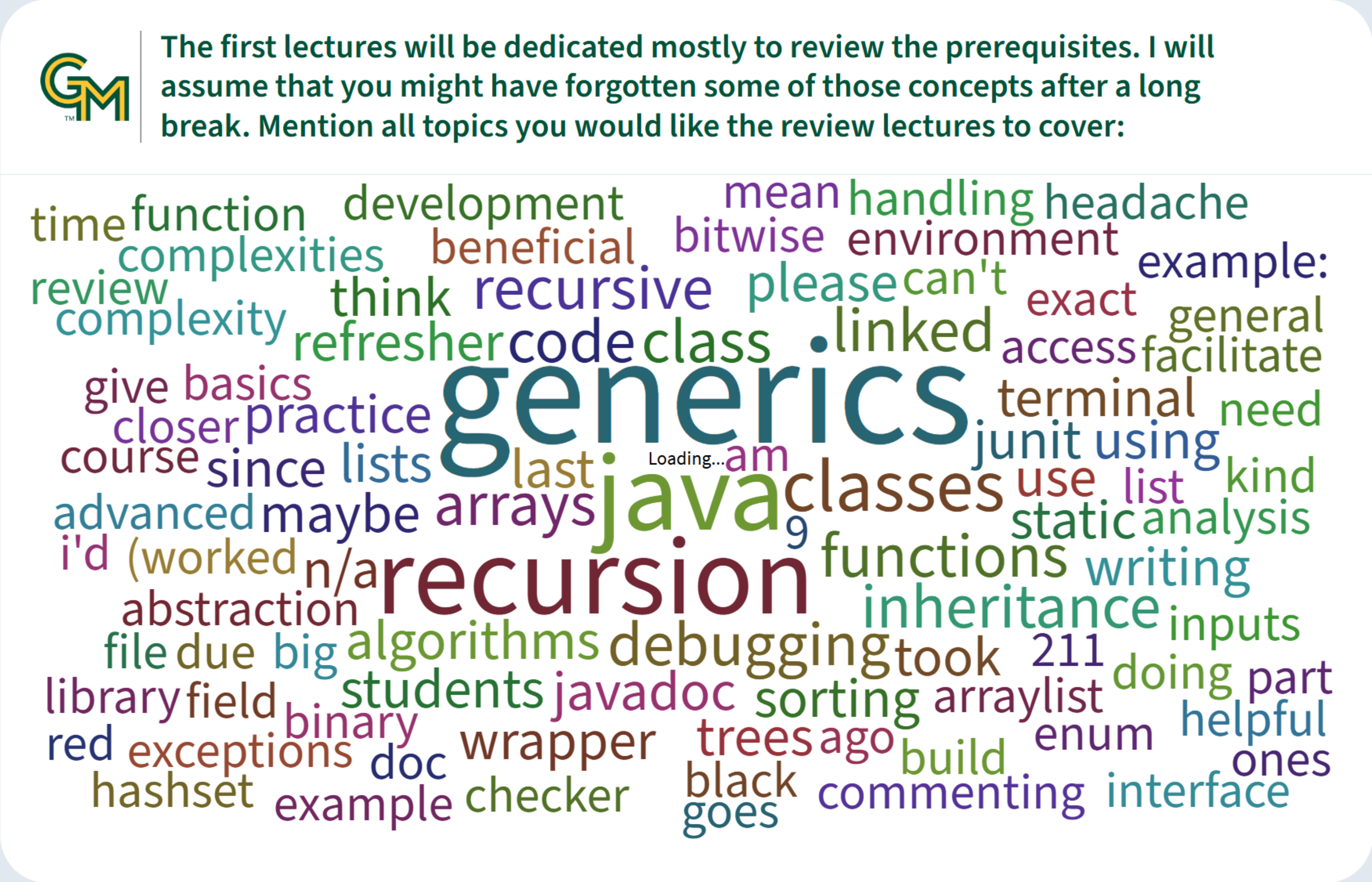


Rate your familiarity with generics in Java programming:



Did you read the syllabus and tentative schedule?

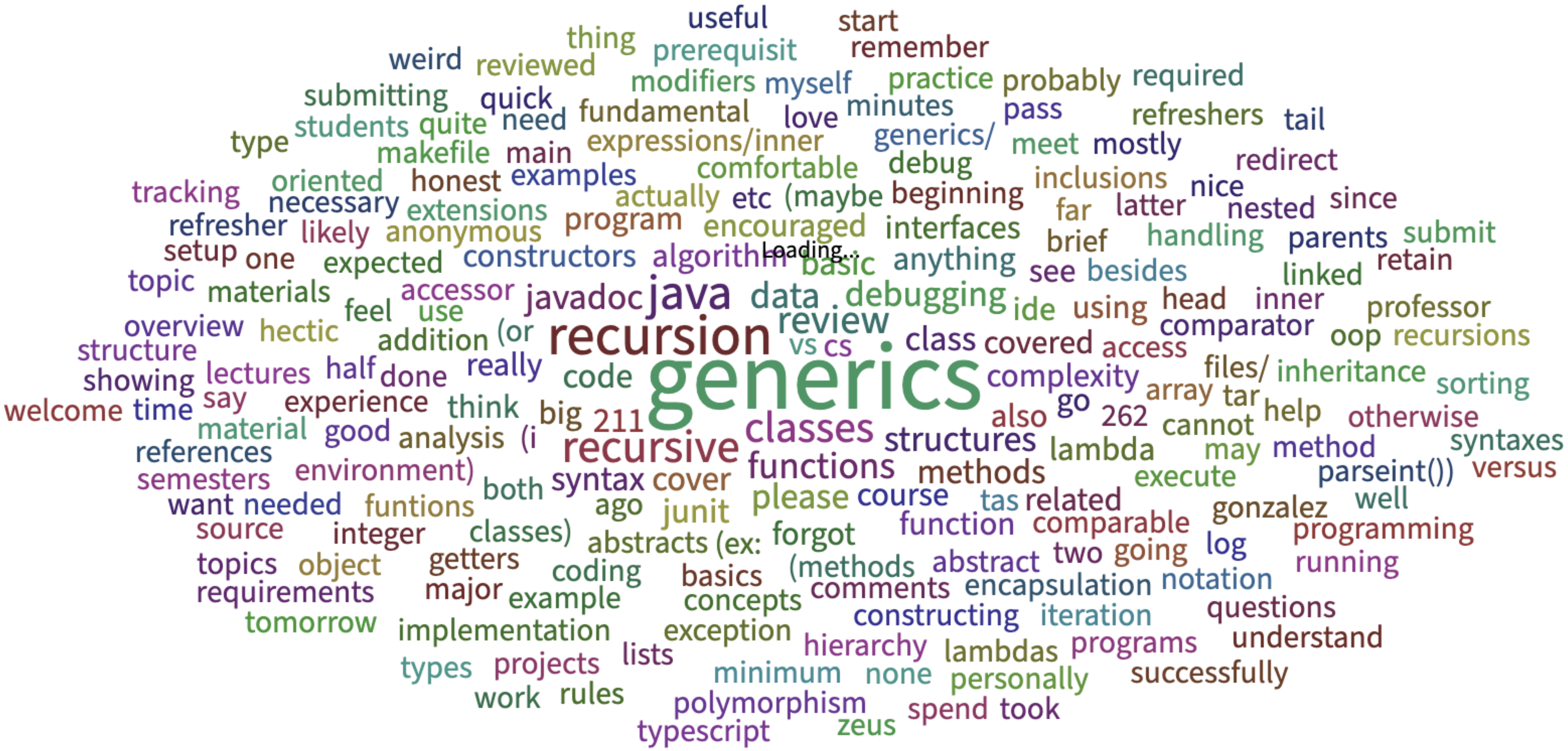




The first lectures will be dedicated mostly to review the prerequisites. I will assume that you might have forgotten some of those concepts after a long break. Mention all topics you would like the review lectures to cover:

those concepts after a long break. Mention all topics you would like the review lectures to cover:

Similar to Spring 2025



01- Review – Java Basics (Data Types, OOP, Inheritance)

Java Basics - Quick Review

Debugging Demo

We will use the files:
- Operation.java
- Program2.java

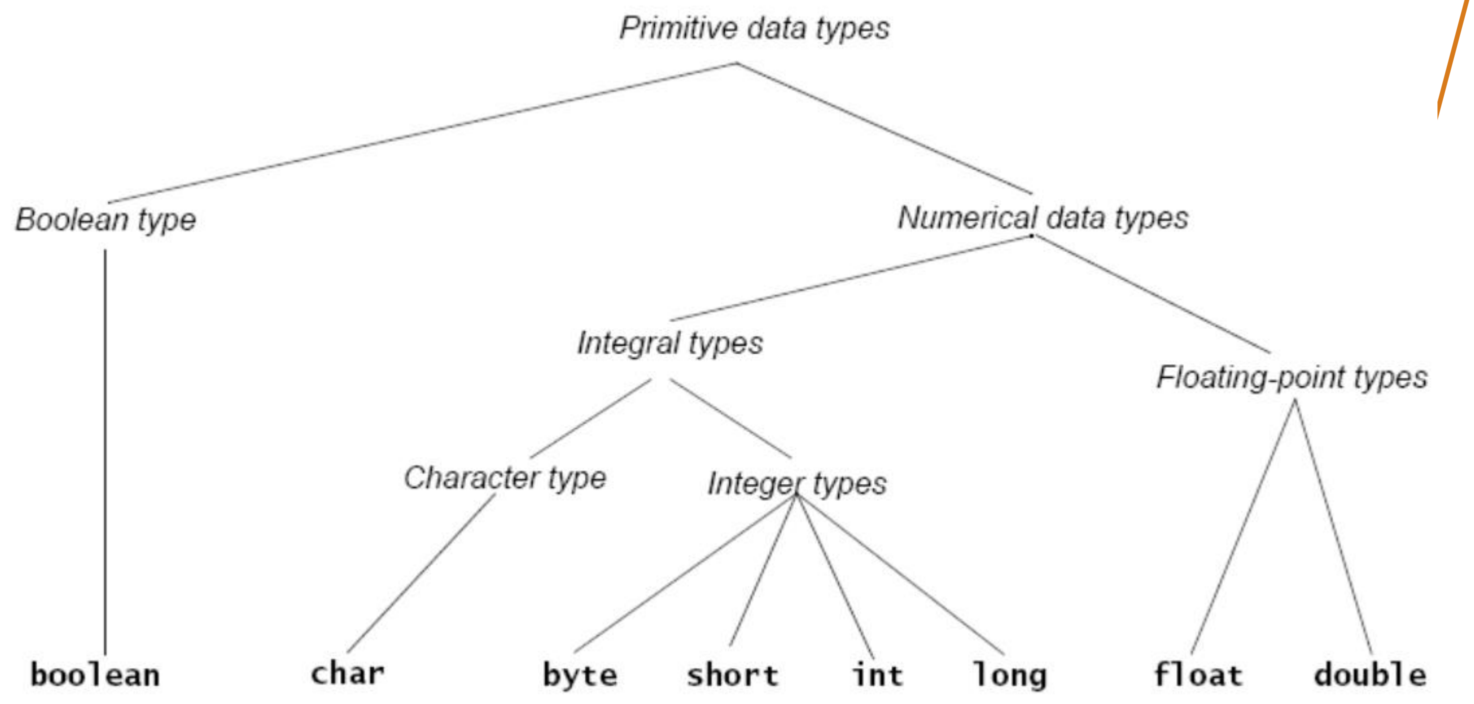
Debugging using System.out.println…
Using the IDE (Breakpoint, Step Over, Step Into, Step out)

Blank Slide

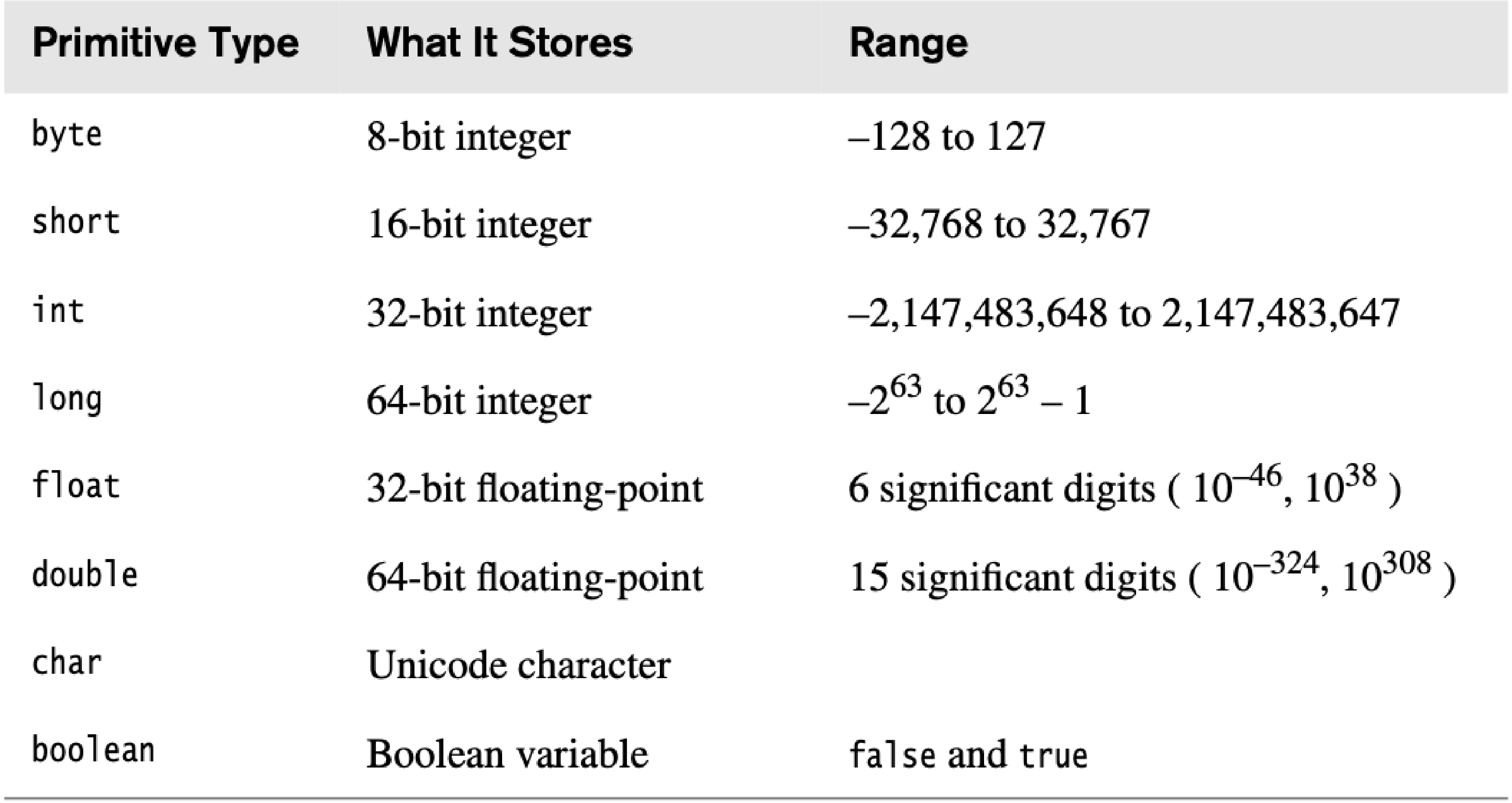
Java Basics - Quick Review

Java has 2 basic data types: Primitive types / Reference type

You should know what reference type correspond to each primitive type.
This will be helpful when using generics. Generic type can be replaced by Reference type, not primitives.



Java Basics - Quick Review



Primitive Type	What It Stores	Range
`byte`	8-bit integer	–128 to 127
`short`	16-bit integer	–32,768 to 32,767
`int`	32-bit integer	–2,147,483,648 to 2,147,483,647
`long`	64-bit integer	-2^{63} to $2^{63} - 1$
`float`	32-bit floating-point	6 significant digits (10^{-46}, 10^{38})
`double`	64-bit floating-point	15 significant digits (10^{-324}, 10^{308})
`char`	Unicode character	
`boolean`	Boolean variable	`false` and `true`

Java Basics - Quick Review

How to better understand **Generics**
--
Java uses OOP Paradigm

Object-Oriented Design Goals:

1) Robustness… In addition to have the correct outputs for anticipated inputs, we want the program to handle unexpected inputs.
2) Adaptability (can evolve, is portable, etc.)
3) **Reusability** (using same code in different systems)

The 3rd goal is the motivation behind: Inheritance, Abstract, Interface, and **Generic**.

Java Basics - Quick Review

Object and References…
Illustration using whiteboard.

```
Car c = new Car();
```

What is **`c`**?
What happen in the memory when **`new Car()`** is executed?

What if we do **`Car d = c;`** ?

Blank Slide

Java Basics - Quick Review

How to better understand **Generics**

--

Keyword: Reusability
via **Inheritance / IS-A relationship** (see Liskov Substitution Principle)
WhiteBoard

Java Basics - Quick Review

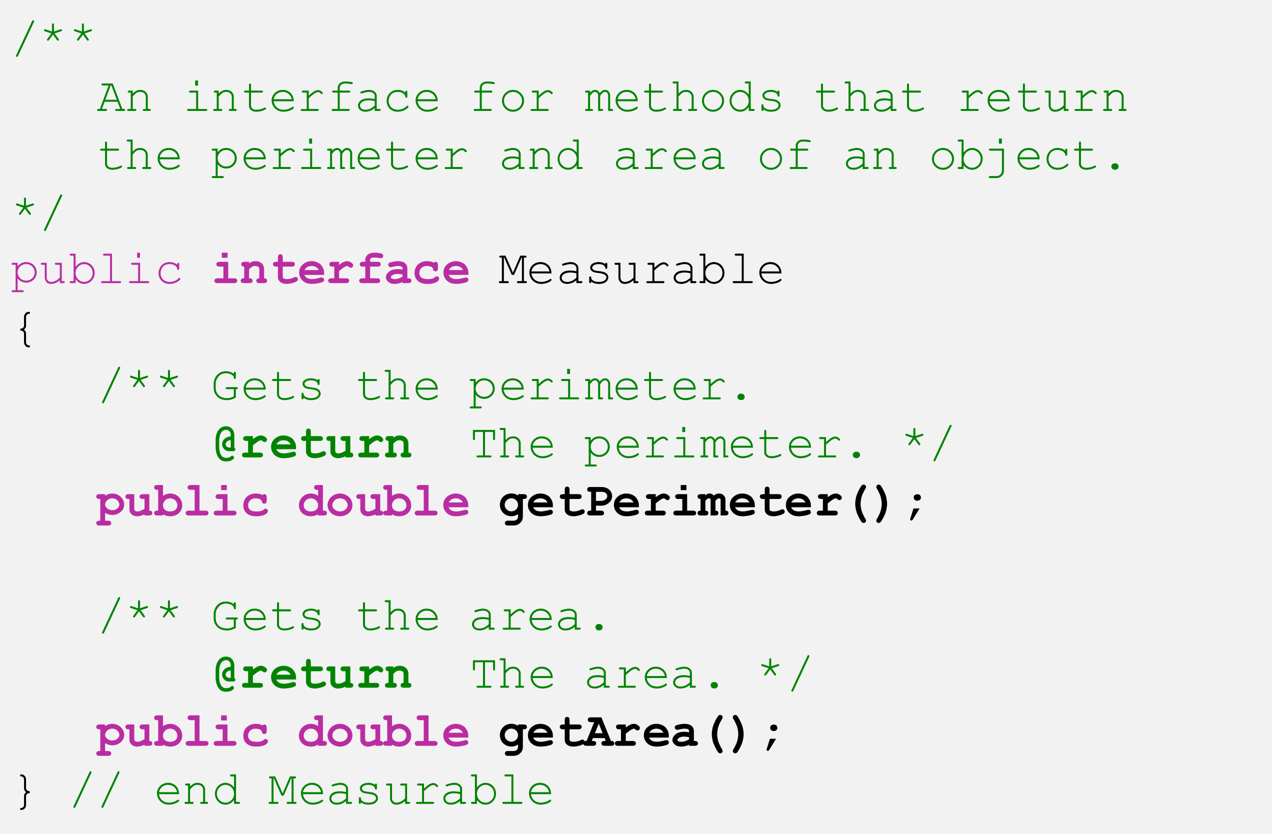
How to better understand **Generics**

--

Keyword: Reusability

via **Abstract/Interface (focus on What instead of How)**

WhiteBoard



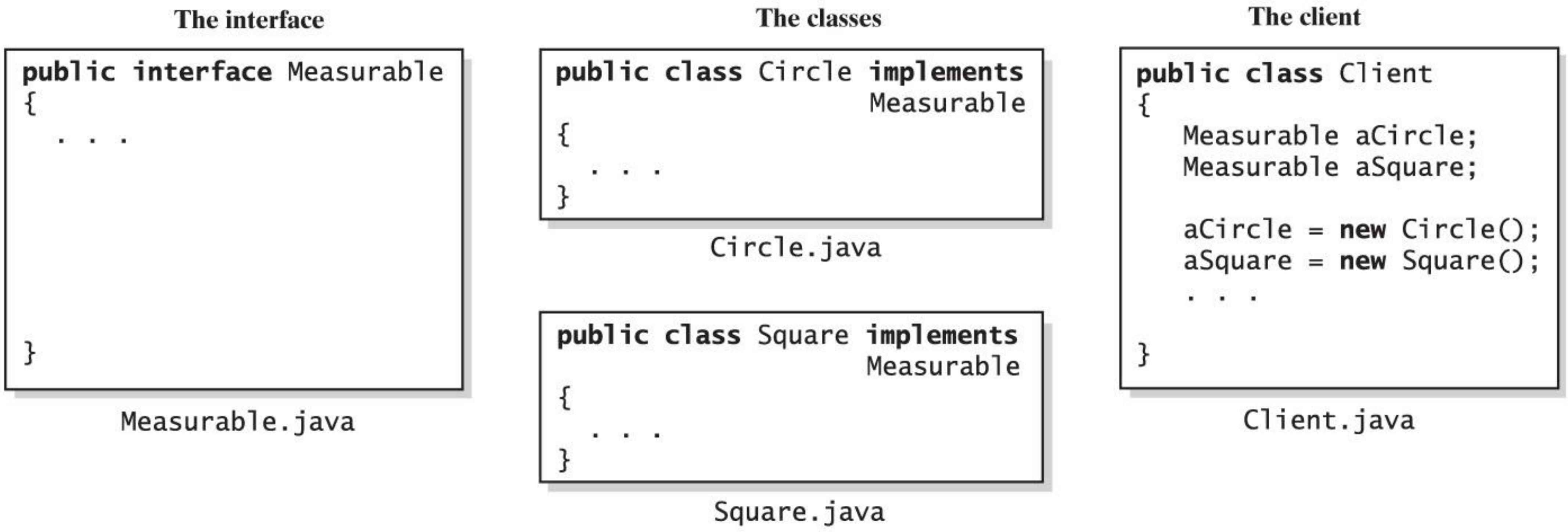
```
/**
   An interface for methods that return
   the perimeter and area of an object.
*/
public interface Measurable
{
   /** Gets the perimeter.
       @return  The perimeter. */
   public double getPerimeter();

   /** Gets the area.
       @return  The area. */
   public double getArea();
} // end Measurable
```

Java Basics - Quick Review

How to better understand **Generics**

--

Keyword: Reusability
via **Abstract/Interface (focus on What instead of How)**



The interface

```
public interface Measurable
{
   . . .



}
```

Measurable.java

The classes

```
public class Circle implements
                         Measurable
{
   . . .
}
```

Circle.java

```
public class Square implements
                         Measurable
{
   . . .
}
```

Square.java

The client

```
public class Client
{
   Measurable aCircle;
   Measurable aSquare;

   aCircle = new Circle();
   aSquare = new Square();
   . . .

}
```

Client.java

OOP review: Interface vs Abstract Class

- **Purpose** of interface similar to that of abstract class
- Use an **abstract class** …
  - If you want to provide a **method definition**
  - Or declare a **private data field** that your classes will have in common
  - Subclasses are **similar in nature**.
- A class can implement several interfaces but **can extend only one abstract class**.

Blank Slide

Java Basics - Quick Review

How to better understand **Generics**

--

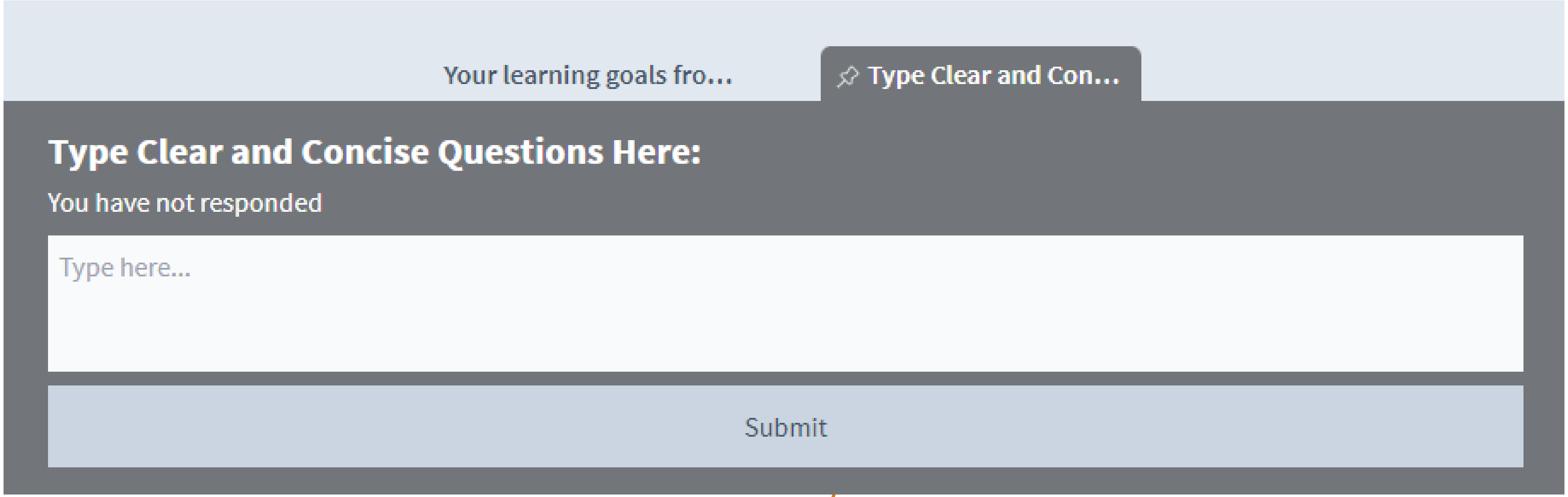
Keyword: Reusability
via **Generics**
Coding examples:
MyGeneric.java (Hint- Liskov Substitution Principle)
See: UseMyGeneric.java
BestGeneric.java
See: UseBestGeneric.java

- Setting Upper Bound / Update with an Interface (Sellable for instance)

Questions…

Use **PollEv.com/adestine** for in-class Q&A

or just **Raise Hand**



Type Clear and Concise Questions Here:

You have not responded

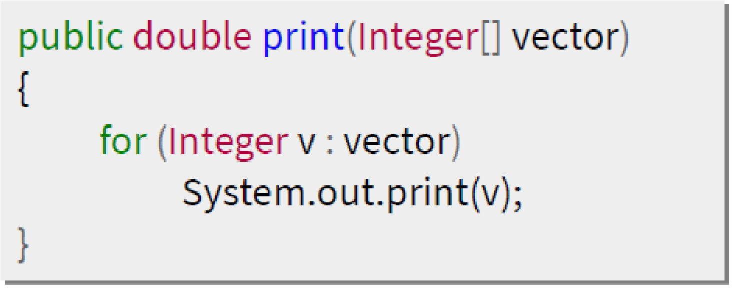
02- Generics

Let's review Generics (learnt from previous course)

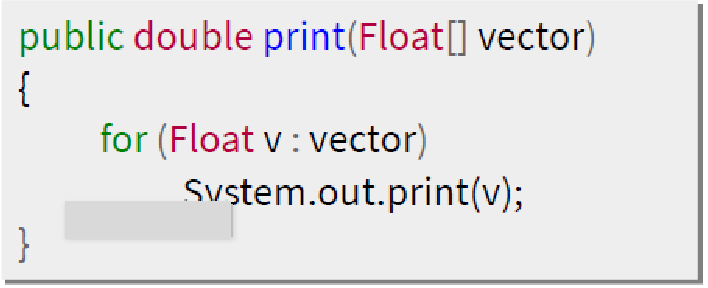
Generics (Quick review)

Problem 1: inefficient overloading

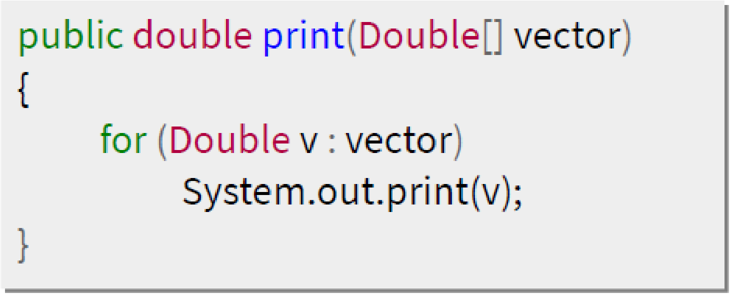
We need a function that can calculate the average of a vector of any type



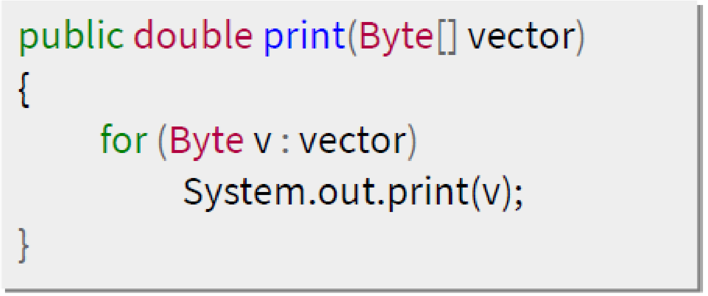
```
public double print(Integer[] vector)
{
        for (Integer v : vector)
                System.out.print(v);
}
```



```
public double print(Float[] vector)
{
        for (Float v : vector)
                System.out.print(v);
}
```



```
public double print(Double[] vector)
{
        for (Double v : vector)
                System.out.print(v);
}
```

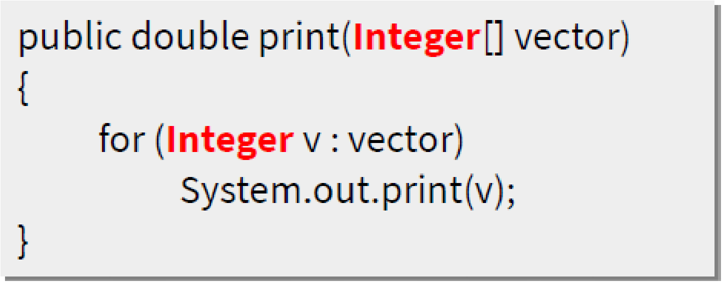


```
public double print(Byte[] vector)
{
        for (Byte v : vector)
                System.out.print(v);
}
```

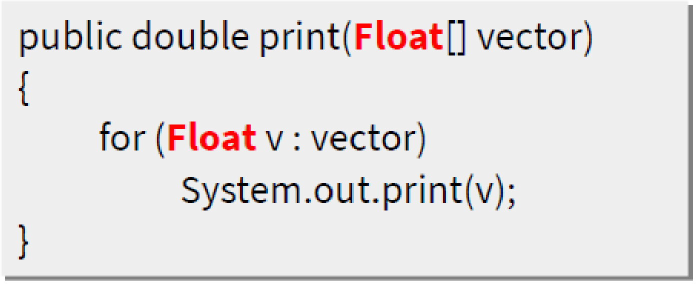
Generics (Quick review)

Problem 1: inefficient overloading

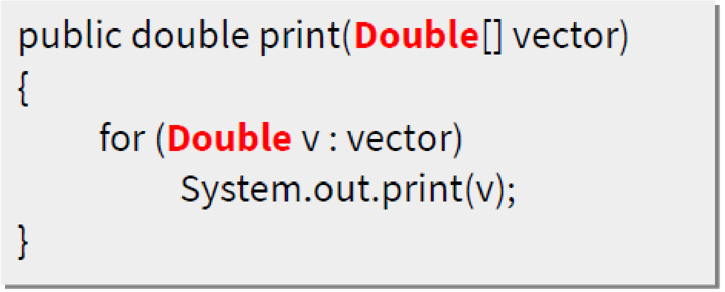
The only variation among these overloaded methods is the **type** of the vector



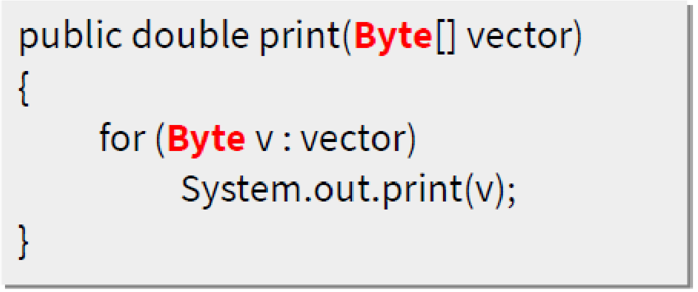
```
public double print(Integer[] vector)
{
        for (Integer v : vector)
                System.out.print(v);
}
```



```
public double print(Float[] vector)
{
        for (Float v : vector)
                System.out.print(v);
}
```



```
public double print(Double[] vector)
{
        for (Double v : vector)
                System.out.print(v);
}
```



```
public double print(Byte[] vector)
{
        for (Byte v : vector)
                System.out.print(v);
}
```

Generics (Quick review)

Problem 1: inefficient overloading

Ideally, we would like to create just **one** method that can dynamically accept any type **T**

```
public double print(T[] vector)
{
        for (T v : vector)
                System.out.print(v);
}
```

```
public double print(Float[] vector)
{
        for (Float v : vector)
                System.out.print(v);
}
```

don’t try this code, it won’t compile as is

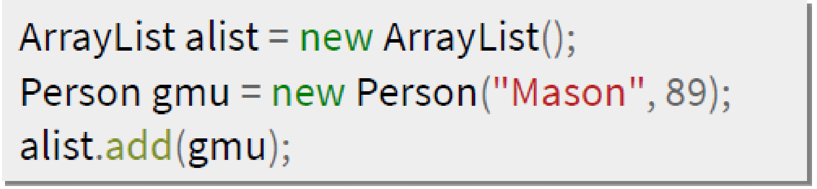
```
public double print(Double[] vector)
{
        for (Double v : vector)
                System.out.print(v);
}
```

```
public double print(Byte[] vector)
{
        for (Byte v : vector)
                System.out.print(v);
}
```

Generics (Quick review)

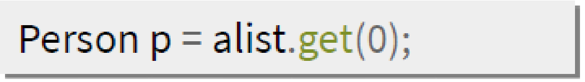
Problem 2: "lost" types

Consider the following code:



```
ArrayList alist = new ArrayList();
Person gmu = new Person("Mason", 89);
alist.add(gmu);
```

And then we can't directly get a Person back out like this:
→ compile-time error: found Object, needed Person



```
Person p = alist.get(0);
```

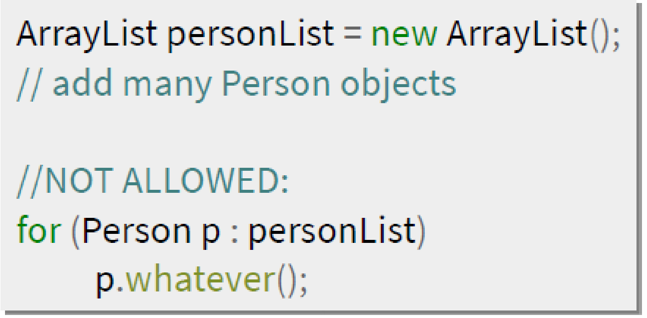
Instead, we must **downcast** everything that comes out of the ArrayList:

Generics (Quick review)

Problem 2: "lost" types

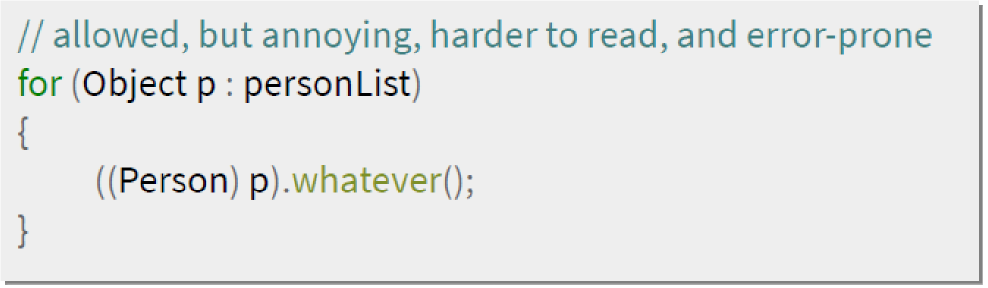
This issue would always arise when we retrieve things from the ArrayList

It's even more annoying with the for-each loop:



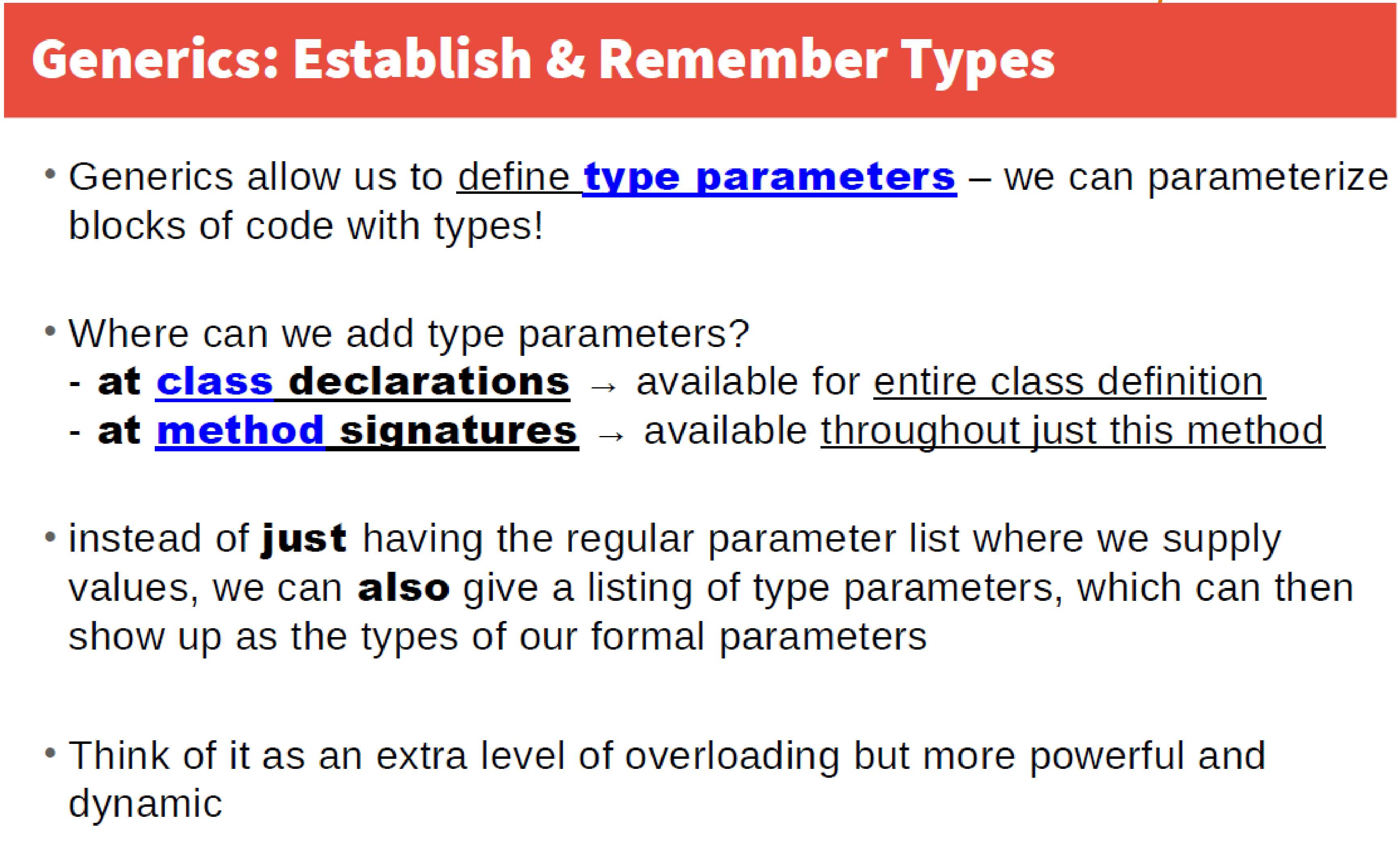
```
ArrayList personList = new ArrayList();
// add many Person objects

//NOT ALLOWED:
for (Person p : personList)
        p.whatever();
```

Instead, we must use **downcasting** after we retrieve with **Object type**, like this:



```
// allowed, but annoying, harder to read, and error-prone
for (Object p : personList)
{
        ((Person) p).whatever();
}
```

Generics (Quick review)



Generics: Establish & Remember Types

- Generics allow us to define **type parameters** – we can parameterize blocks of code with types!

- Where can we add type parameters?
  - **at class declarations** → available for entire class definition
  - **at method signatures** → available throughout just this method

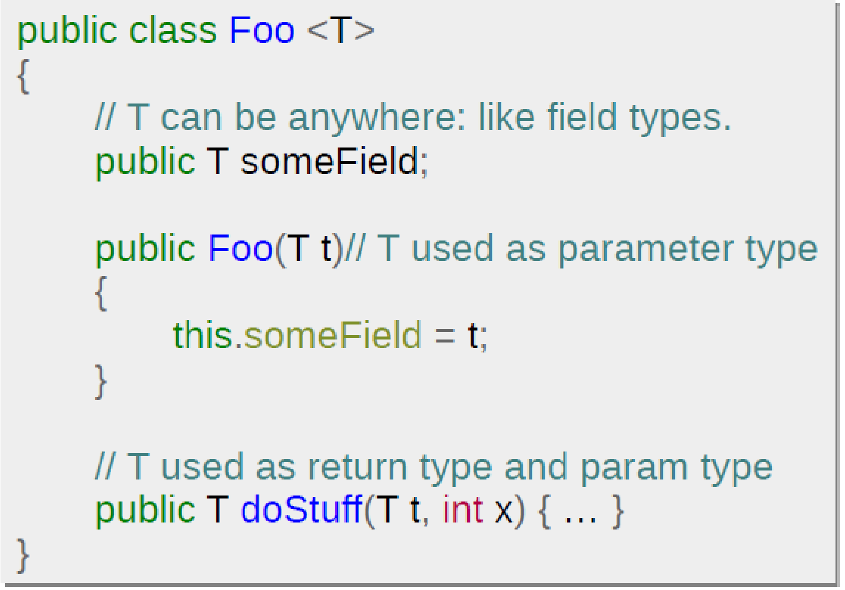
- instead of **just** having the regular parameter list where we supply values, we can **also** give a listing of type parameters, which can then show up as the types of our formal parameters

- Think of it as an extra level of overloading but more powerful and dynamic

Generics (Quick review)

Declaring Generic Classes

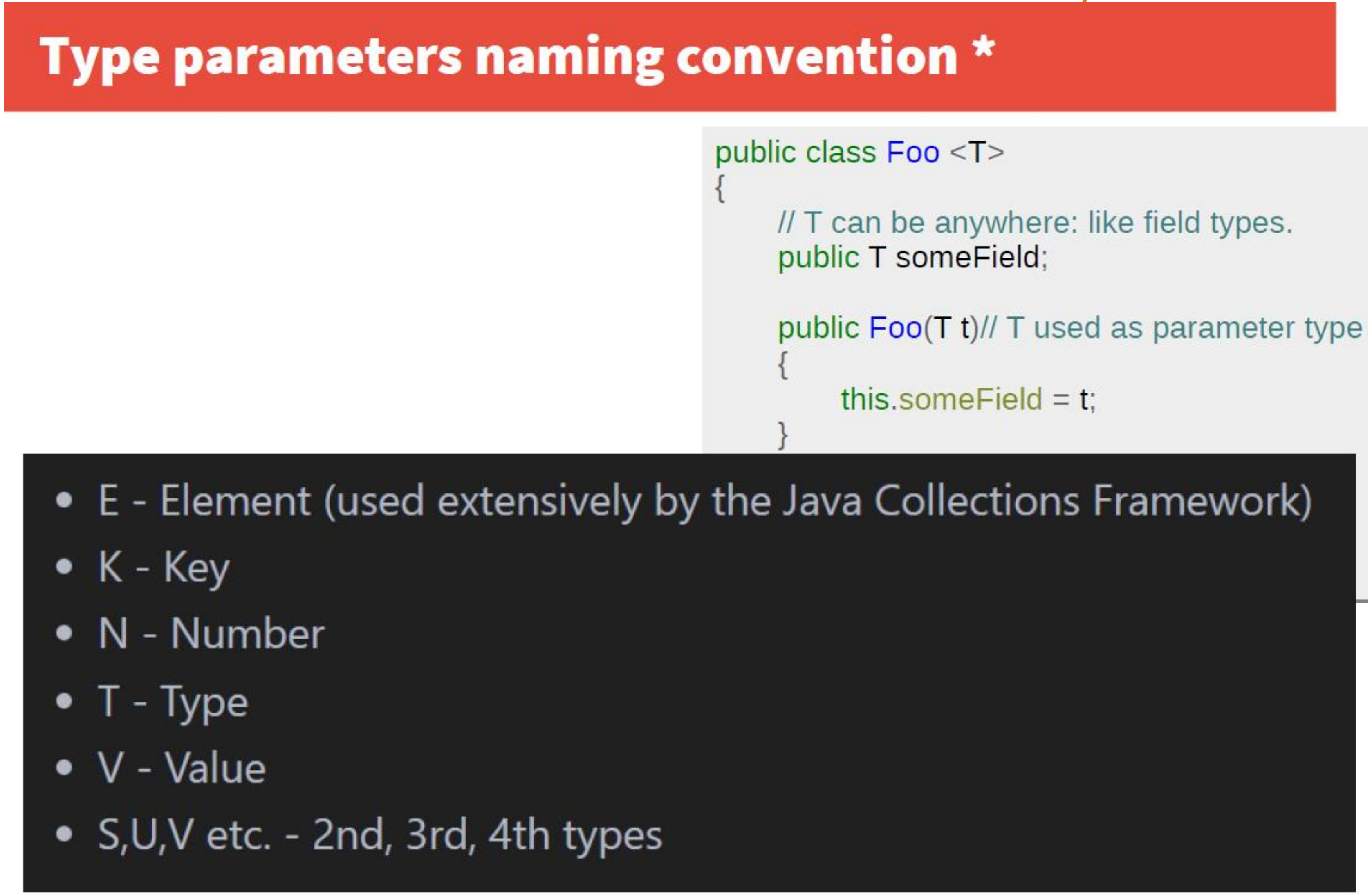
We can add a generic type to a class definition:



```
public class Foo <T>
{
      // T can be anywhere: like field types.
      public T someField;

      public Foo(T t)// T used as parameter type
      {
            this.someField = t;
      }

      // T used as return type and param type
      public T doStuff(T t, int x) { … }
}
```

Simply add the ***<T>*** or ***<E>*** or whatever name you like right after the class name, and then use ***T*** instead of a specific type in your code

Generics (Quick review)



Type parameters naming convention *

```
public class Foo <T>
{
      // T can be anywhere: like field types.
      public T someField;

      public Foo(T t)// T used as parameter type
      {
            this.someField = t;
      }
```

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

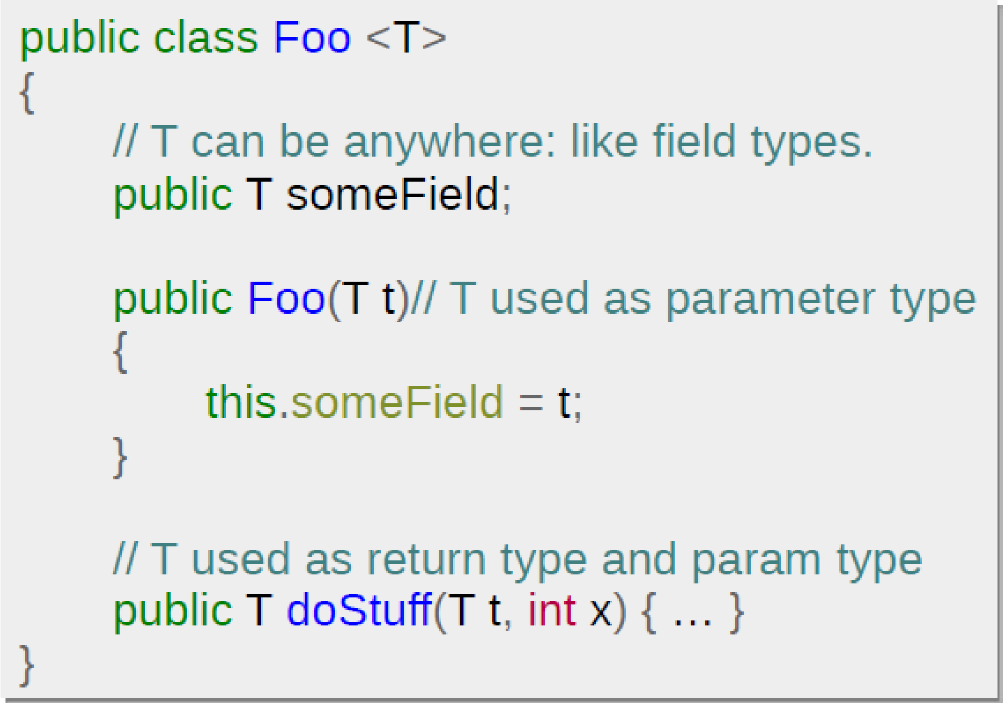
Generics (Quick review)

Generic Type invocation

Replace T with a concrete type:

Ex:

Foo<Car> myVar;



```
public class Foo <T>
{
    // T can be anywhere: like field types.
    public T someField;

    public Foo(T t)// T used as parameter type
    {
        this.someField = t;
    }

    // T used as return type and param type
    public T doStuff(T t, int x) { … }
}
```

Generics (Quick review)

Syntax to declare generic class / generic method

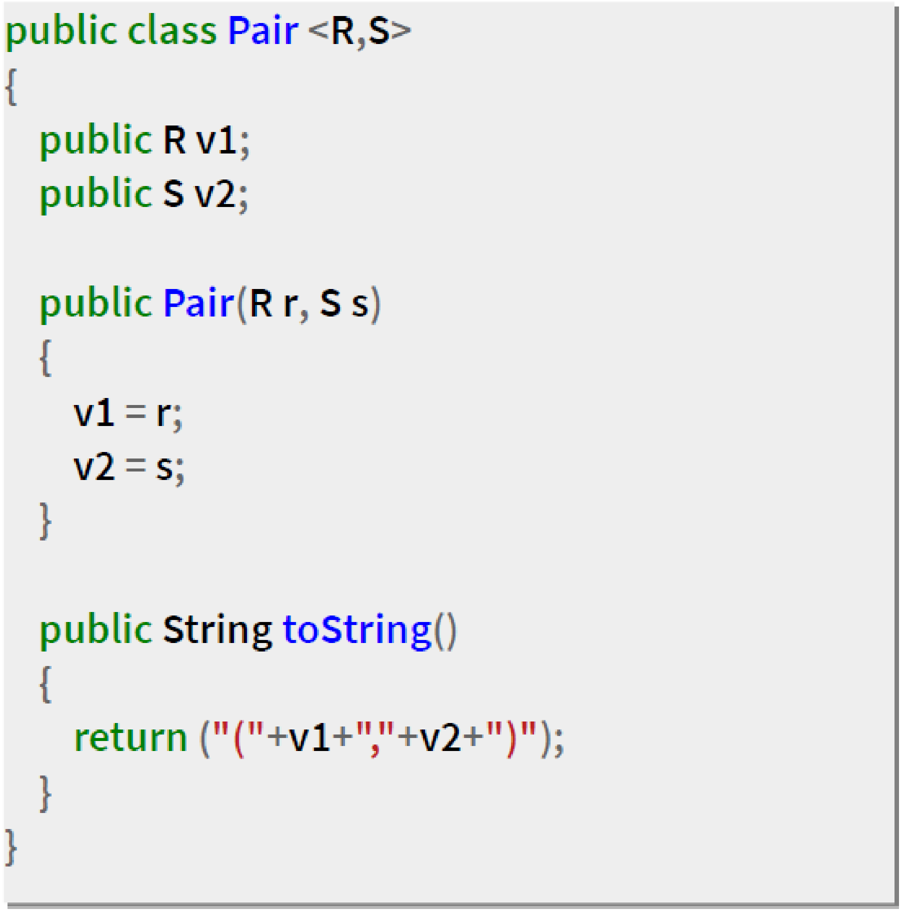
Syntax:

- Declaring generic at Class level

- Declaring generic at Method level

Generics (Quick review)

Declaring Generic Classes with more types

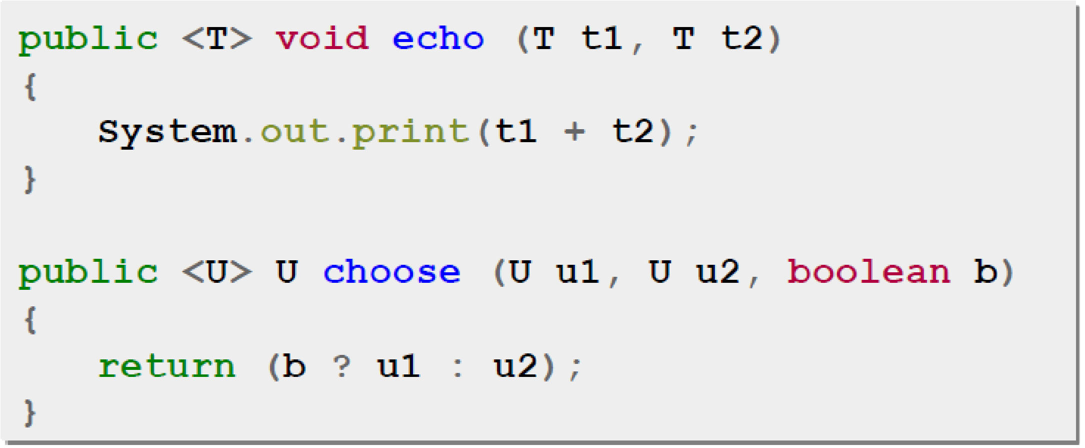


```
public class Pair <R,S>
{
   public R v1;
   public S v2;

   public Pair(R r, S s)
   {
      v1 = r;
      v2 = s;
   }

   public String toString()
   {
      return ("("+v1+","+v2+")");
   }
}
```

Generics (Quick review)

Declaring Generic Methods

- In a generic method the **<T>** notation must go **before the return type**
- It may then be used as a type anywhere in the method: parameter types, local definitions' types… even the return type!
- All we know about **u1** or **u2** is that it is a value of the **U** type. That's not much info! But we can still write useful, highly re-usable code this way

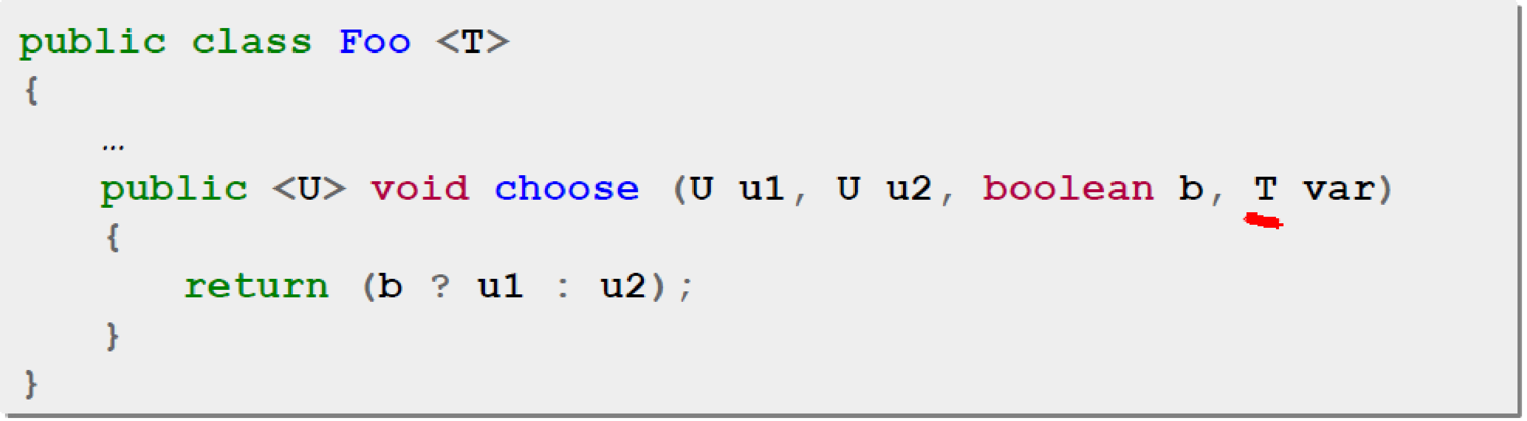


```java
public <T> void echo (T t1, T t2)
{
    System.out.print(t1 + t2);
}

public <U> U choose (U u1, U u2, boolean b)
{
    return (b ? u1 : u2);
}
```

Generics (Quick review)

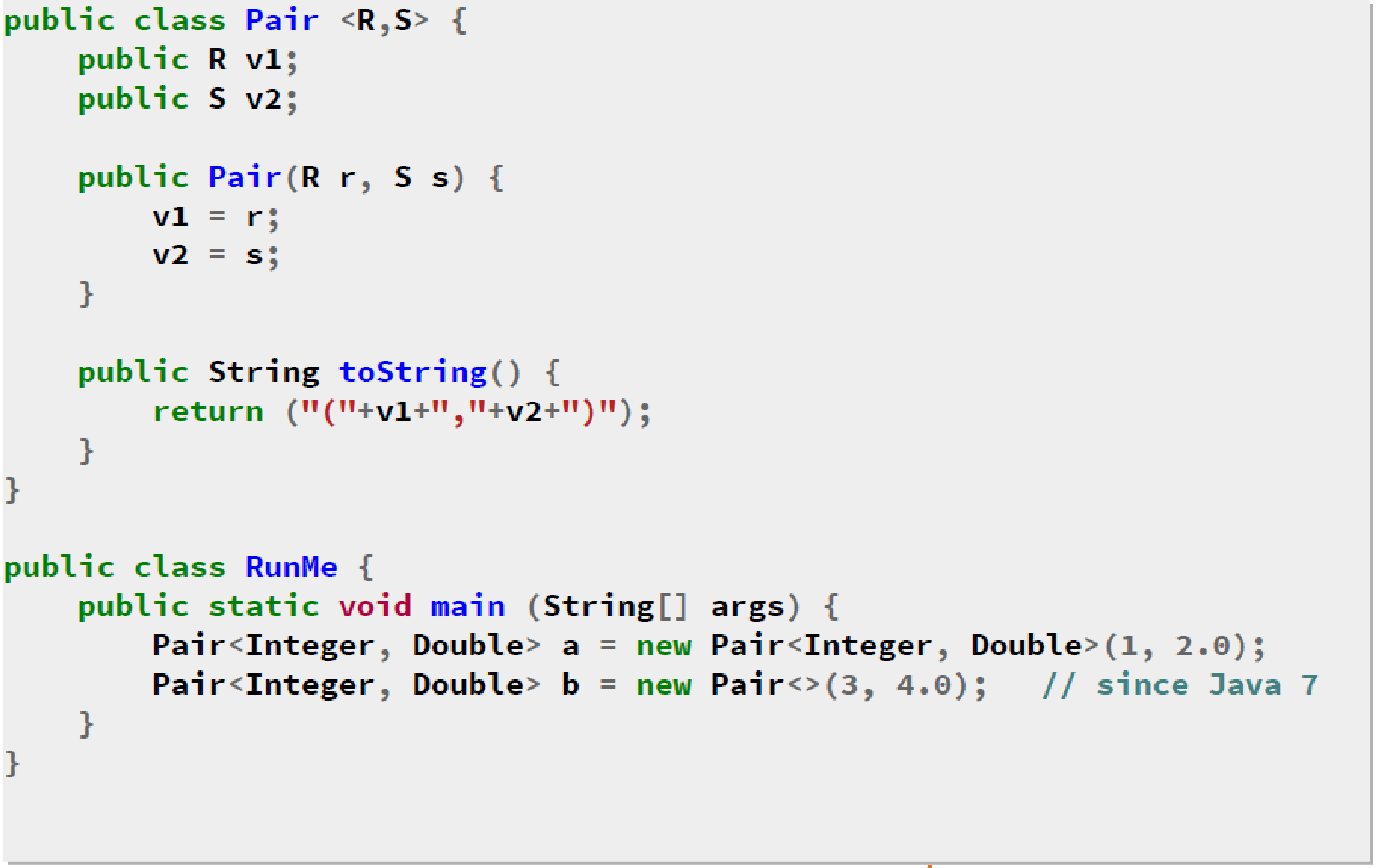
Declaring both Generic Class and Generic Method

We can declare new generic types that are only visible with one method, like **<U>**, and have a different generic type for the class, like **<T>**



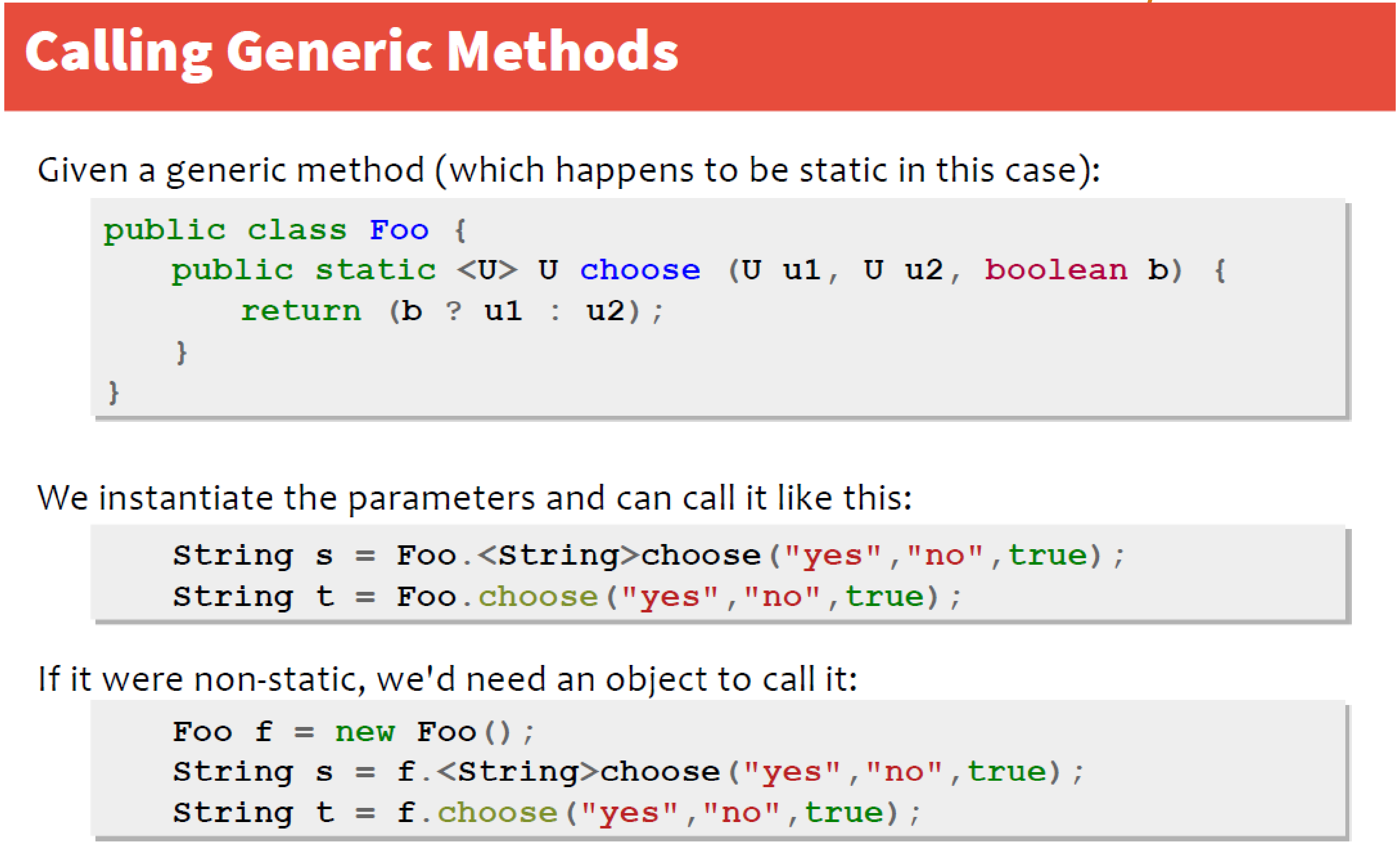
```
public class Foo <T>
{
    …
    public <U> void choose (U u1, U u2, boolean b, T var)
    {
        return (b ? u1 : u2);
    }
}
```

Generics (Quick review)



```java
public class Pair <R,S> {
    public R v1;
    public S v2;

    public Pair(R r, S s) {
        v1 = r;
        v2 = s;
    }

    public String toString() {
        return ("("+v1+","+v2+")");
    }
}

public class RunMe {
    public static void main (String[] args) {
        Pair<Integer, Double> a = new Pair<Integer, Double>(1, 2.0);
        Pair<Integer, Double> b = new Pair<>(3, 4.0);   // since Java 7
    }
}
```

Generics (Quick review)



Calling Generic Methods

Given a generic method (which happens to be static in this case):

```
public class Foo {
    public static <U> U choose (U u1, U u2, boolean b) {
        return (b ? u1 : u2);
    }
}
```

We instantiate the parameters and can call it like this:

```
    String s = Foo.<String>choose("yes","no",true);
    String t = Foo.choose("yes","no",true);
```

If it were non-static, we'd need an object to call it:

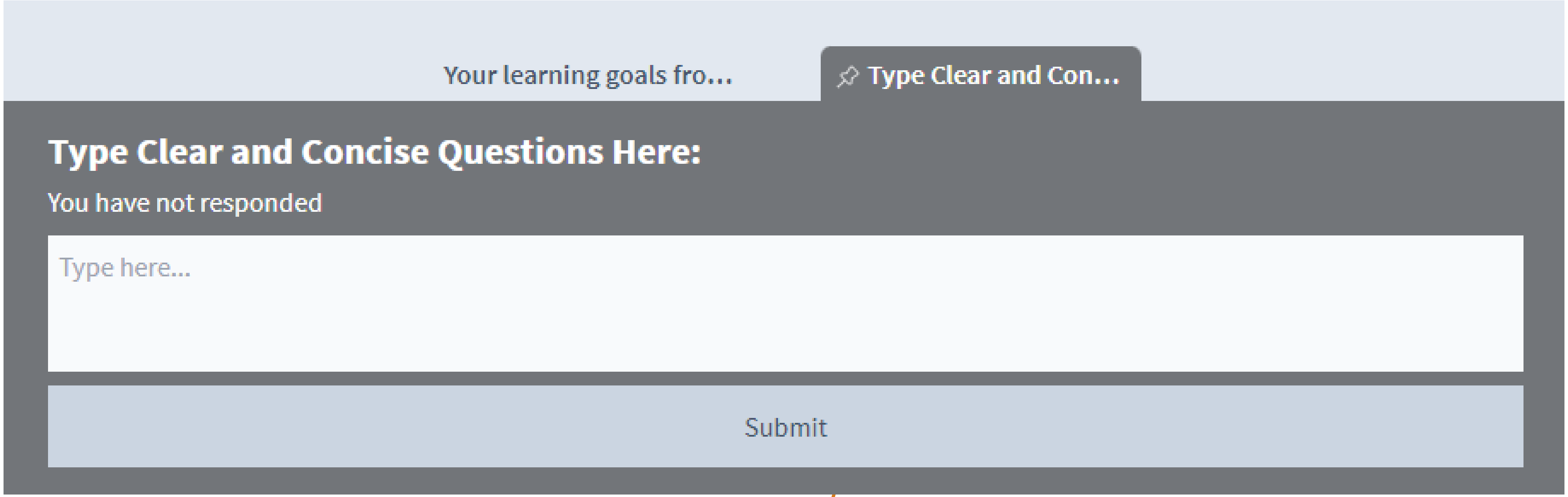
```
    Foo f = new Foo();
    String s = f.<String>choose("yes","no",true);
    String t = f.choose("yes","no",true);
```

Since Java 7 we only need to specify the generic type when it's not clear from the parameters

Questions…

Use **PollEv.com/adestine** for in-class Q&A

or just **Raise Hand**



Type Clear and Concise Questions Here:

You have not responded

More on Generics
Consider those 2 problems…

Write a non-generic class that:

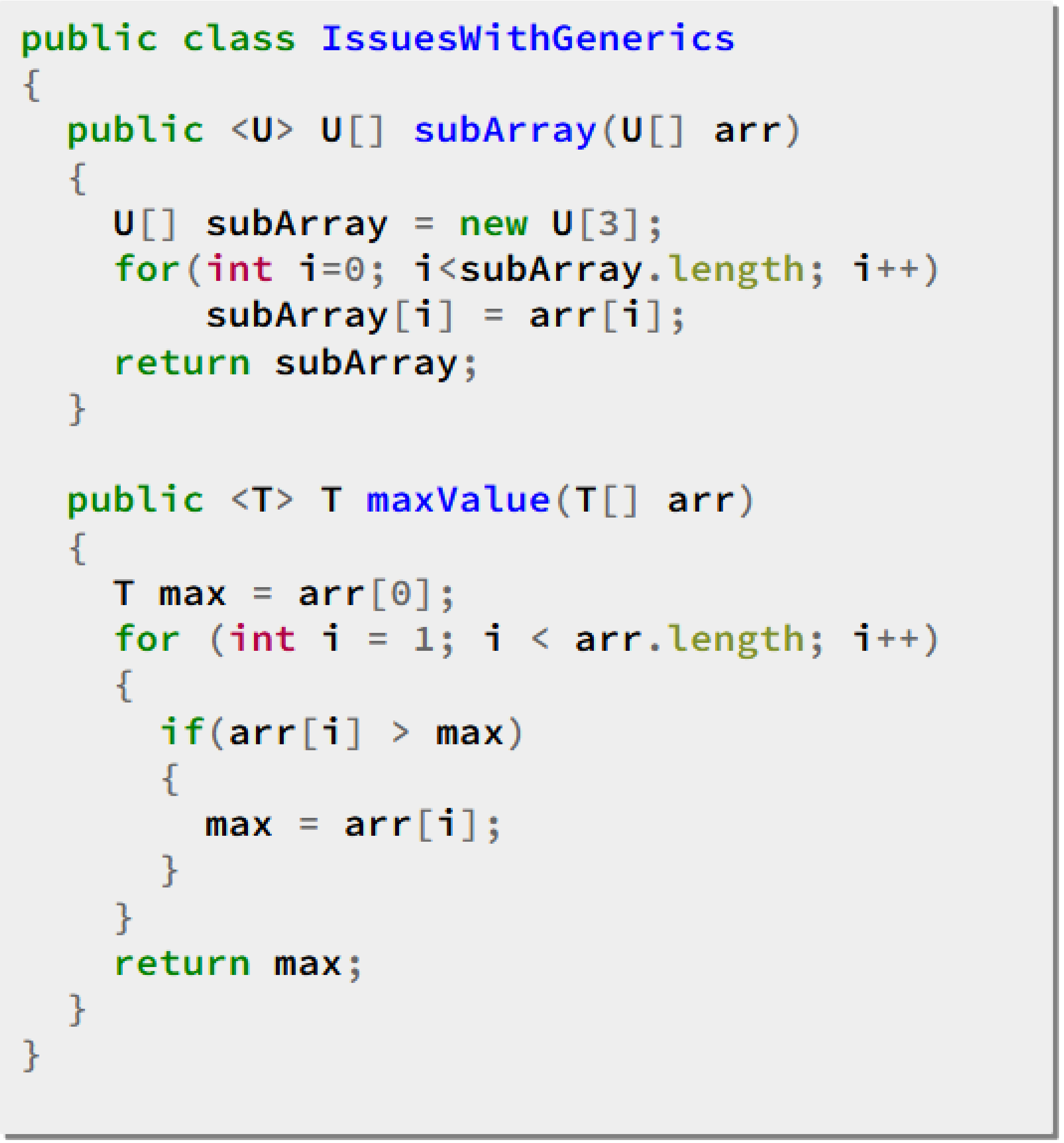
1. Has a generic method that returns a subarray of the first three items of a generic array

```
public <U> U[] subArray(U[] arr)
```

2. Has a generic method that returns the max value of a generic array

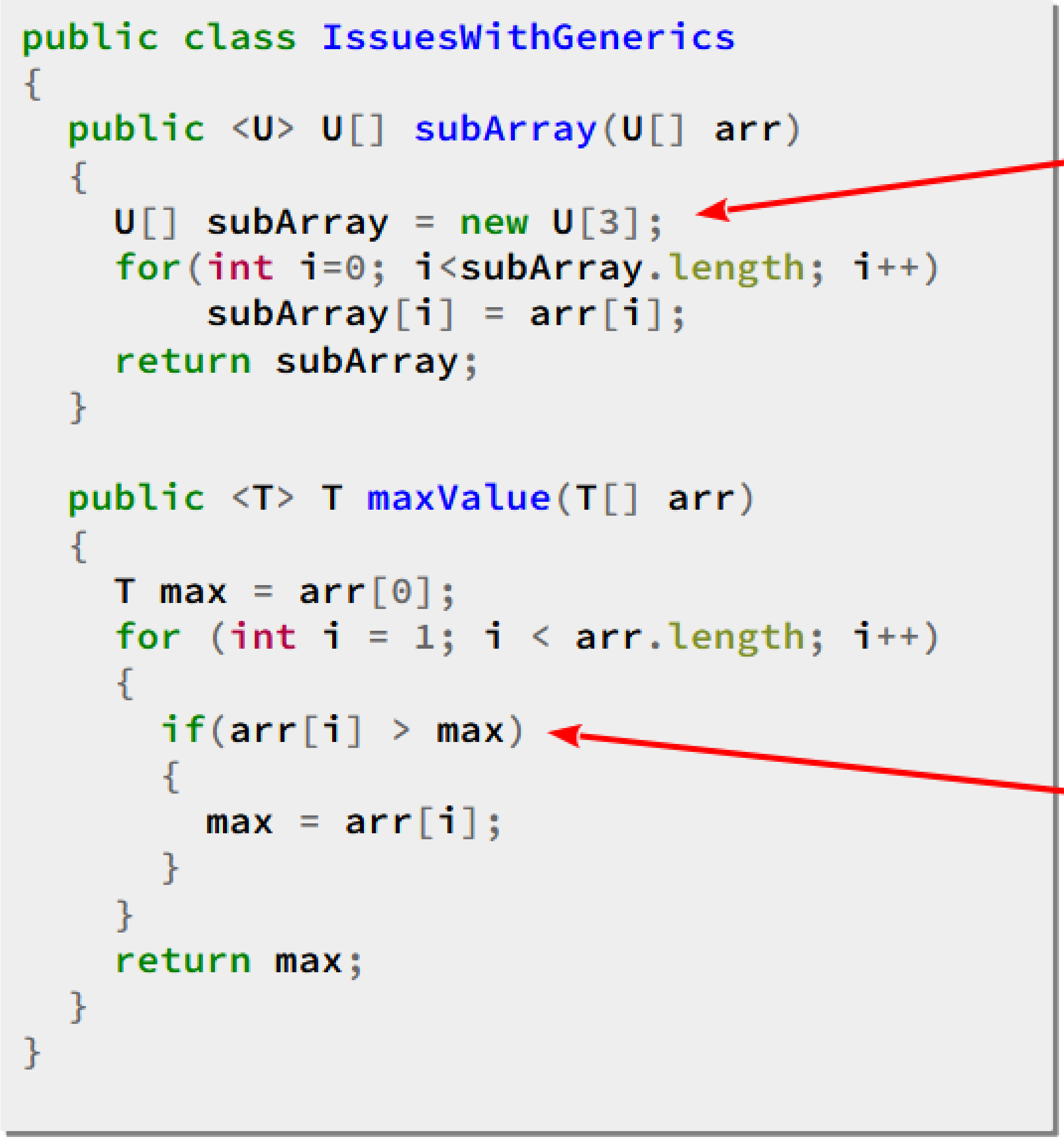
```
public <T> T maxValue(T[] arr)
```

Are those solutions correct ?



```
public class IssuesWithGenerics
{
  public <U> U[] subArray(U[] arr)
  {
    U[] subArray = new U[3];
    for(int i=0; i<subArray.length; i++)
        subArray[i] = arr[i];
    return subArray;
  }

  public <T> T maxValue(T[] arr)
  {
    T max = arr[0];
    for (int i = 1; i < arr.length; i++)
    {
      if(arr[i] > max)
      {
        max = arr[i];
      }
    }
    return max;
  }
}
```

Are those solutions correct ?

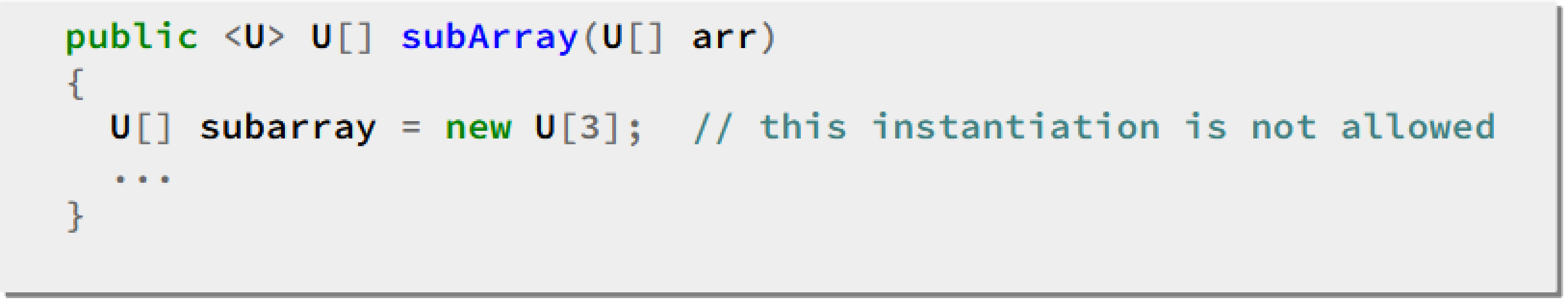


```java
public class IssuesWithGenerics
{
  public <U> U[] subArray(U[] arr)
  {
    U[] subArray = new U[3];
    for(int i=0; i<subArray.length; i++)
        subArray[i] = arr[i];
    return subArray;
  }

  public <T> T maxValue(T[] arr)
  {
    T max = arr[0];
    for (int i = 1; i < arr.length; i++)
    {
      if(arr[i] > max)
      {
        max = arr[i];
      }
    }
    return max;
  }
}
```

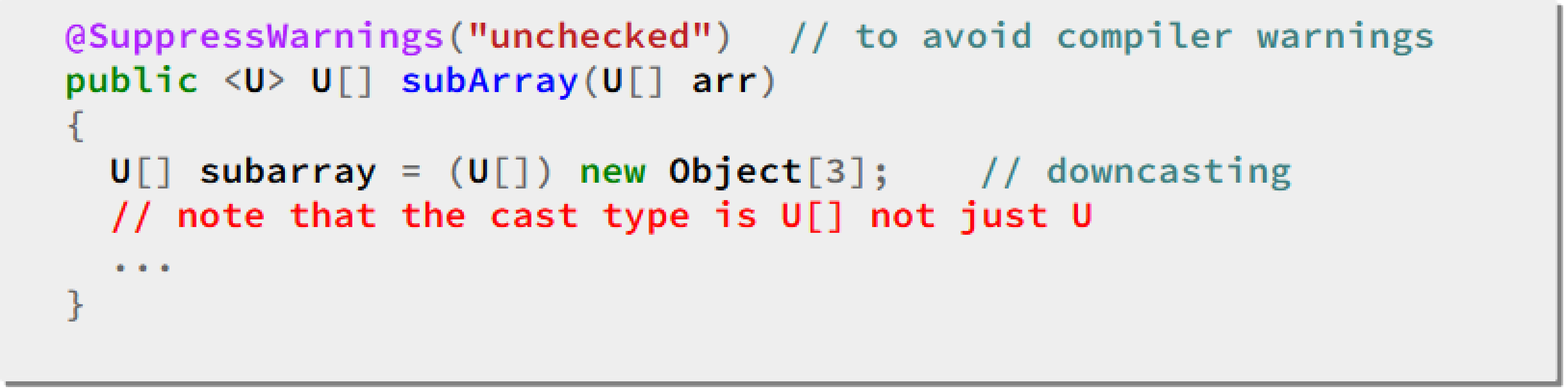
ERROR

ERROR

Are those solutions correct ?

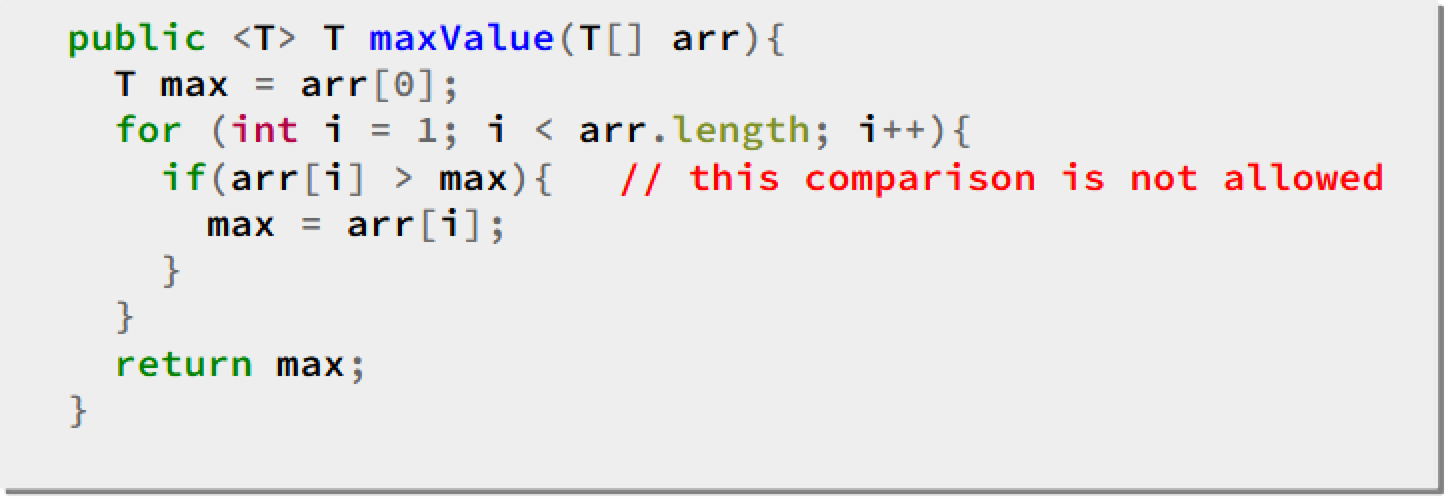


```java
public <U> U[] subArray(U[] arr)
{
  U[] subarray = new U[3];  // this instantiation is not allowed
  ...
}
```

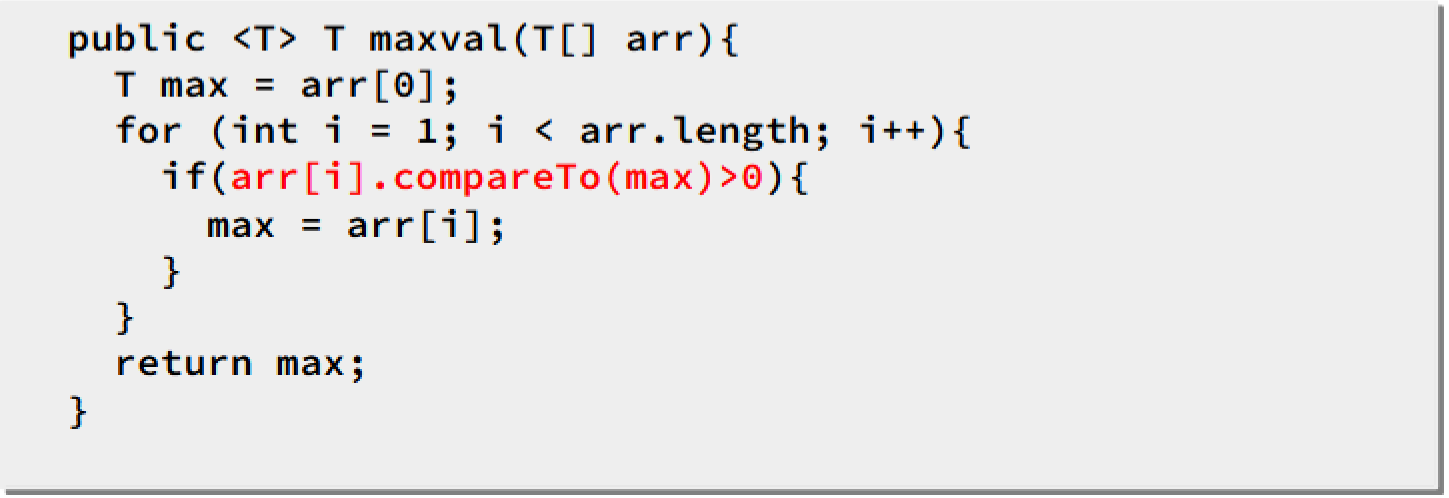


```java
@SuppressWarnings("unchecked")  // to avoid compiler warnings
public <U> U[] subArray(U[] arr)
{
  U[] subarray = (U[]) new Object[3];    // downcasting
  // note that the cast type is U[] not just U
  ...
}
```

Are those solutions correct ?

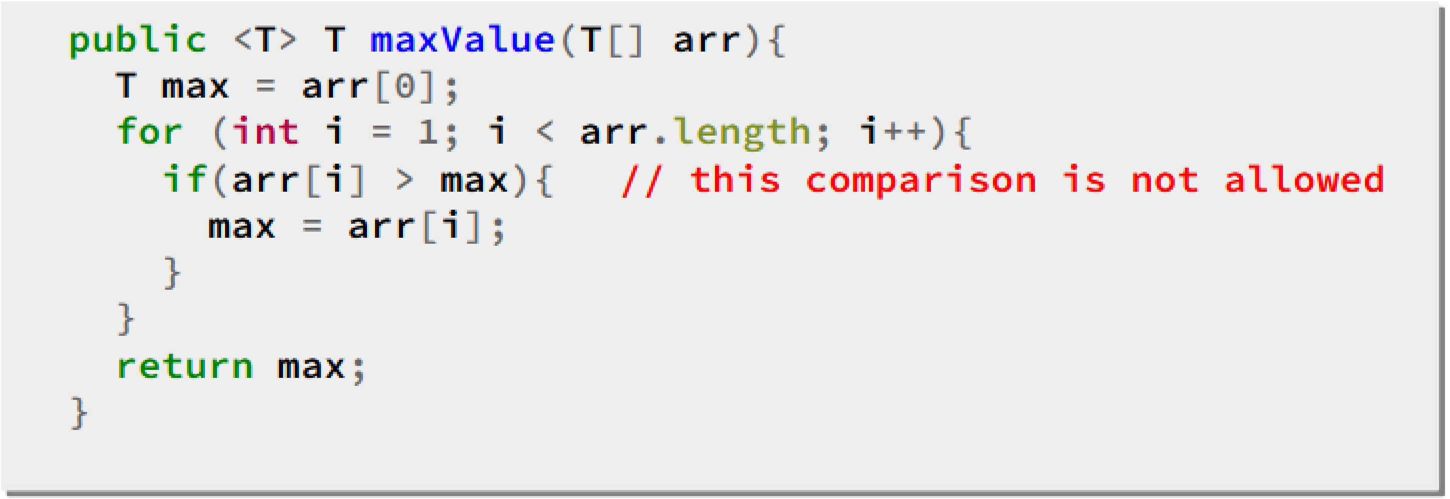


```
public <T> T maxValue(T[] arr){
  T max = arr[0];
  for (int i = 1; i < arr.length; i++){
    if(arr[i] > max){   // this comparison is not allowed
      max = arr[i];
    }
  }
  return max;
}
```

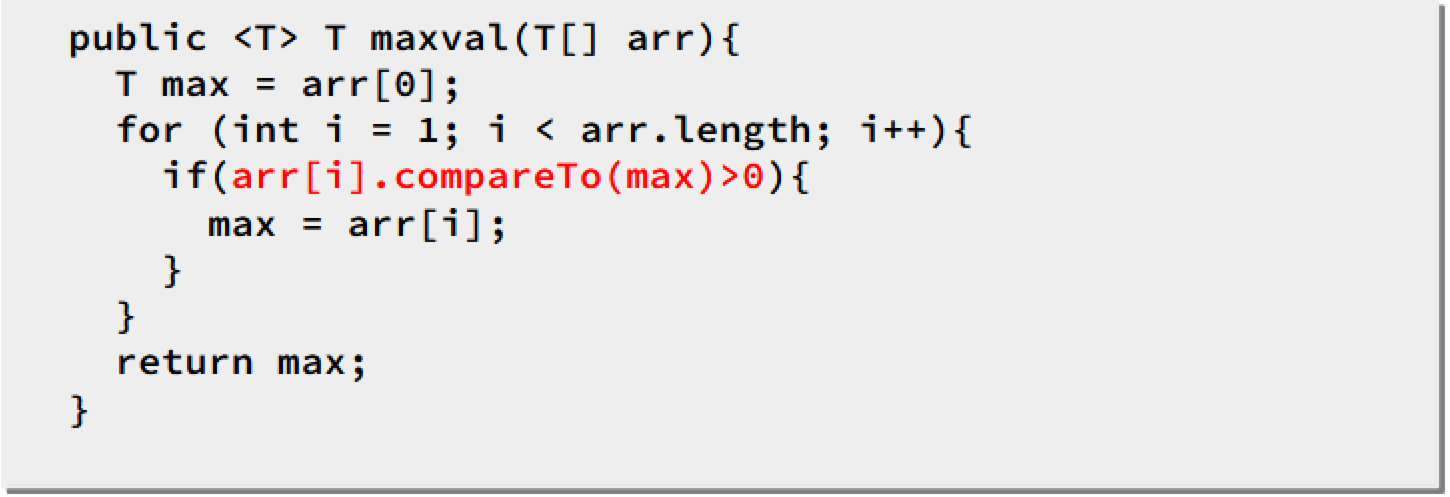


```
public <T> T maxval(T[] arr){
  T max = arr[0];
  for (int i = 1; i < arr.length; i++){
    if(arr[i].compareTo(max)>0){
      max = arr[i];
    }
  }
  return max;
}
```

Are those solutions correct ?



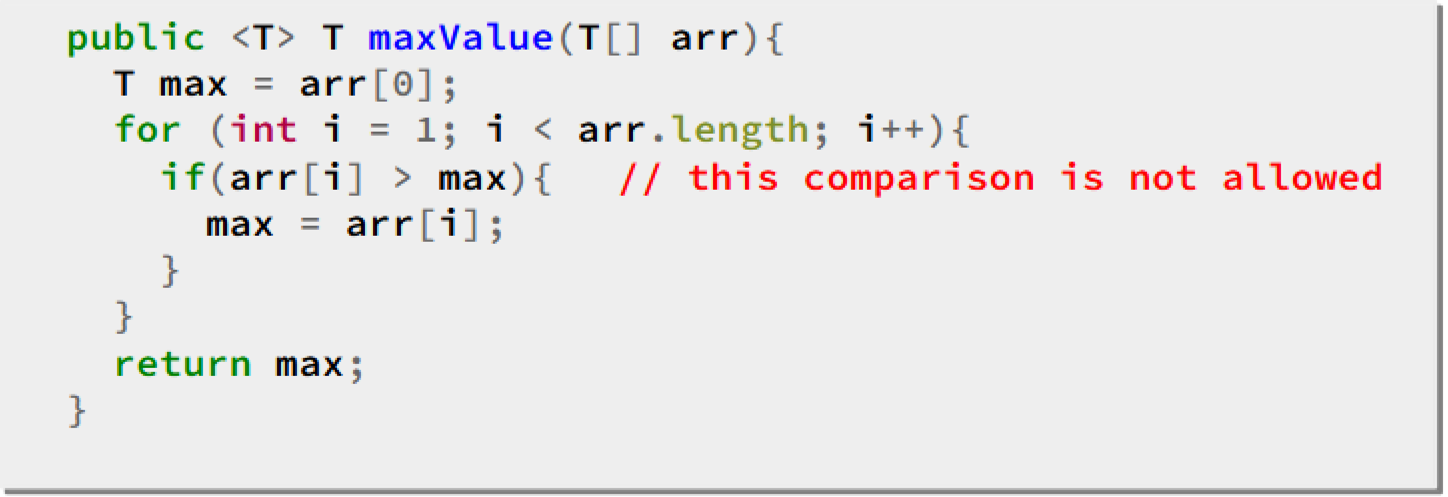
```
public <T> T maxValue(T[] arr){
  T max = arr[0];
  for (int i = 1; i < arr.length; i++){
    if(arr[i] > max){   // this comparison is not allowed
      max = arr[i];
    }
  }
  return max;
}
```



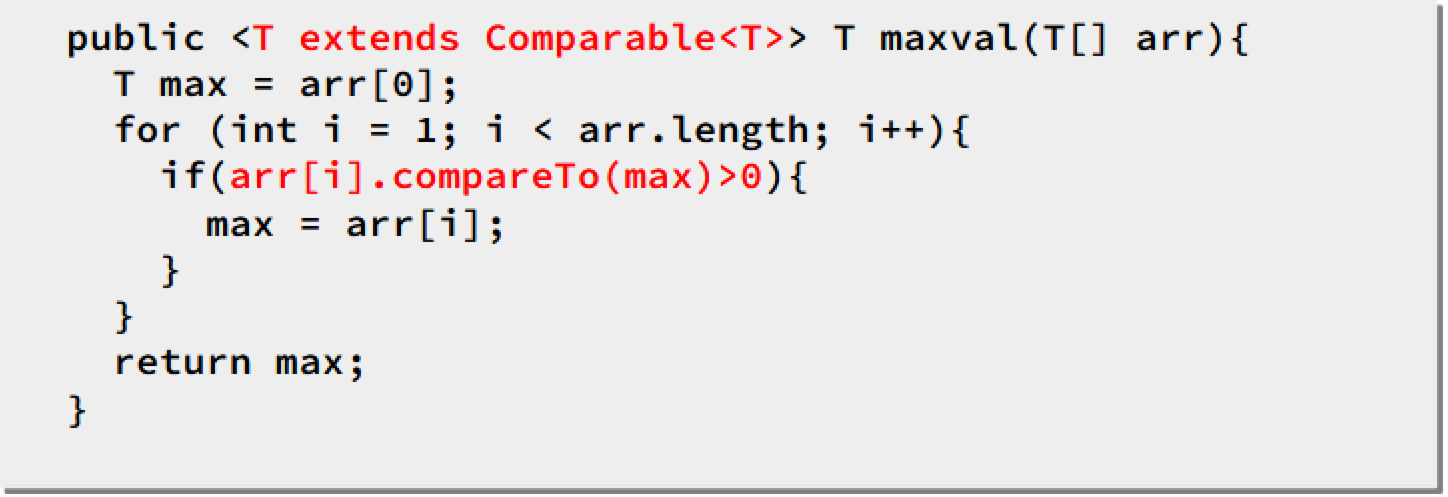
```
public <T> T maxval(T[] arr){
  T max = arr[0];
  for (int i = 1; i < arr.length; i++){
    if(arr[i].compareTo(max)>0){
      max = arr[i];
    }
  }
  return max;
}
```

Unfortunately this doesn’t work because compiler can’t tell if T has a **compareTo()** method

Are those solutions correct ?



```java
public <T> T maxValue(T[] arr){
  T max = arr[0];
  for (int i = 1; i < arr.length; i++){
    if(arr[i] > max){   // this comparison is not allowed
      max = arr[i];
    }
  }
  return max;
}
```



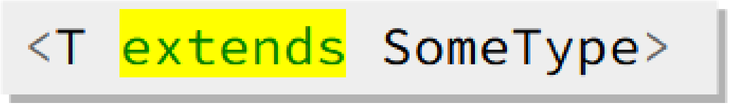
```java
public <T extends Comparable<T>> T maxval(T[] arr){
  T max = arr[0];
  for (int i = 1; i < arr.length; i++){
    if(arr[i].compareTo(max)>0){
      max = arr[i];
    }
  }
  return max;
}
```

The compiler now does know that T comes with a compareTo() method because T implements the Comparable interface

Bounded Type Parameters – Upper Bound

- Sometimes you want to restrict the types that can be used as type arguments

- For example, a method that operates on numbers might only want to accept instances of Number or its subclasses

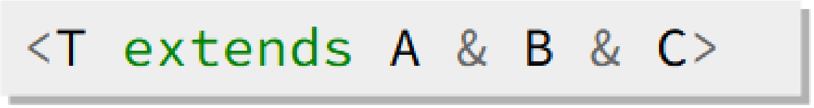
- To declare an **upper bound** we use the **extends** keyword: 

```
<T extends SomeType>
```

- It means that T is a sub-class of SomeType or implements the SomeType interface

Bounded Type Parameters – Upper Bound

- We can even add **multiple extensions**:



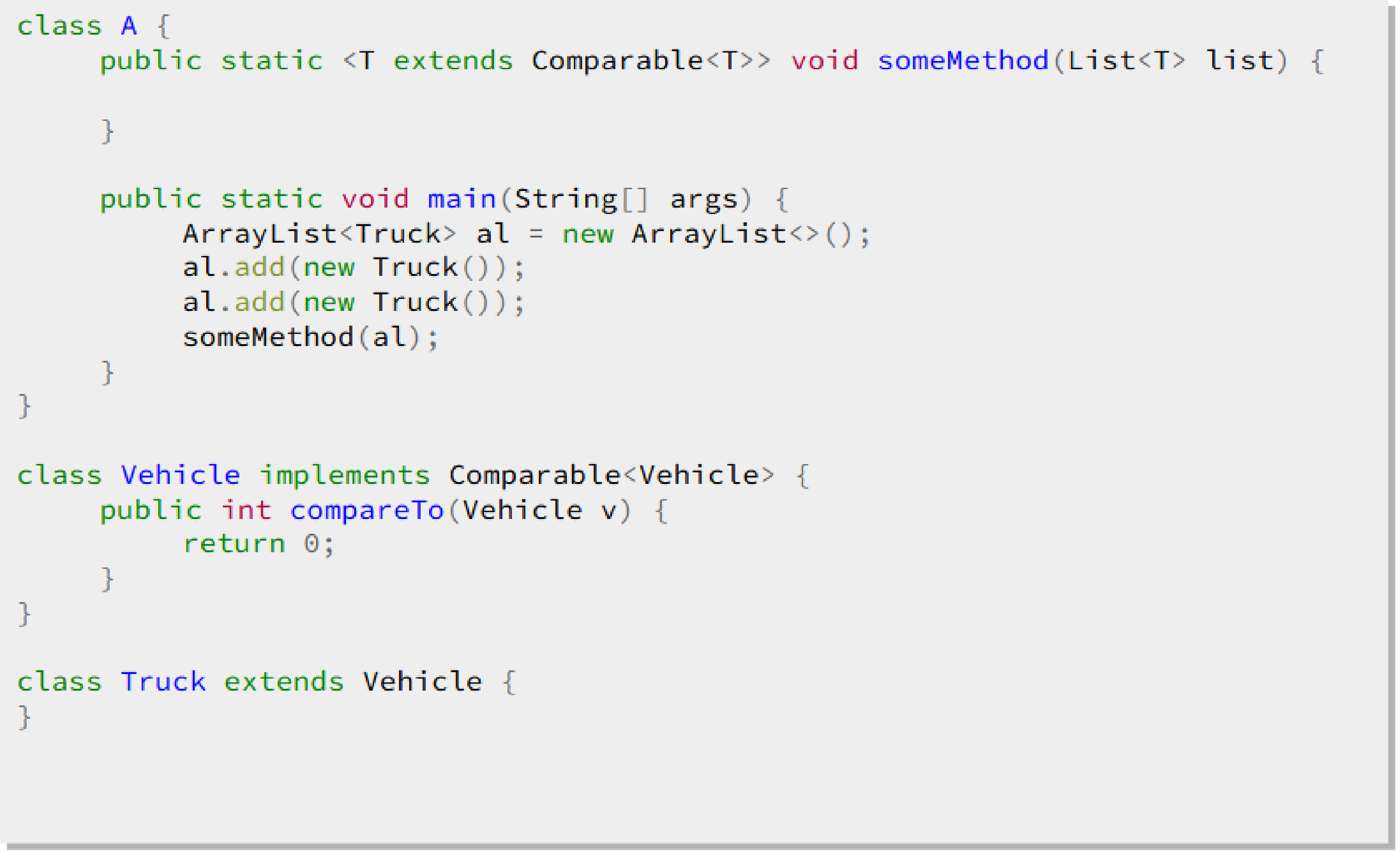
```
<T extends A & B & C>
```

- The same way that classes/interfaces gave us subtypes, we're now saying "any class that's a subtype of SomeType". But we can make multiple claims at once this way

- In multiple extensions, **if one of the bounds is a class**, it must be specified first (i.e. before the interfaces)

Sometimes Upper bound is not enough

The following doesn't compile. Can you see why?

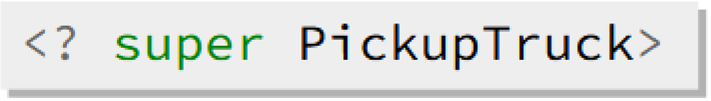


```java
class A {
     public static <T extends Comparable<T>> void someMethod(List<T> list) {

     }

     public static void main(String[] args) {
          ArrayList<Truck> al = new ArrayList<>();
          al.add(new Truck());
          al.add(new Truck());
          someMethod(al);
     }
}

class Vehicle implements Comparable<Vehicle> {
     public int compareTo(Vehicle v) {
          return 0;
     }
}

class Truck extends Vehicle {
}
```

We must replace `Comparable<T>` with `Comparable<? super T>`

Bounded Type Parameters – Lower Bound

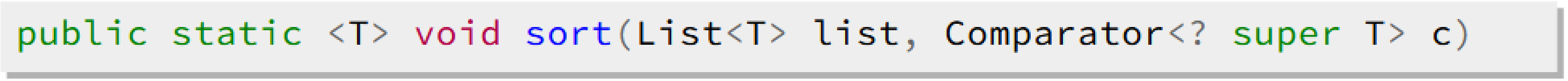
- We can use generics with a lower bound, indicating that it's acceptable for a type parameter to be any type that is a supertype of something particular. Example:



```
<? super PickupTruck>
```

In this case, we can use any type that can accept PickupTruck values like PickupTruck, Truck, and Vehicle

- Look for instance at Collections.sort



```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

We could have limited ourselves to Comparator<T> that would only allow Comparator<PickupTruck> values to sort a list of PickupTruck objects. But if we also had a Comparator<Truck>, or a Comparator<Vehicle> it would make perfect sense to use them as another way to sort trucks or vehicles, which certainly includes PickupTrucks.

By using the **super** keyword we can set a **lower bound** and accept all these different comparators when sorting a list of PickupTrucks.

Combining Lower Bounds and Upper Bounds

- We can also mix upper and lower bounds in the same declaration. Example:

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

It means that the items in the list must be descendants of **`Comparable`** (otherwise it's impossible to compare them and then sort them), but the implementation of the Comparable interface itself (i.e. the implementation of the **`compareTo`** method) doesn't necessarily have to come from **`T`** directly, it can be inherited from its ancestor(s).

- How did we come up with the above declaration. Three attempts:

```
public static <T> void sort(List<T> list) // error
```

```
public static <T extends Comparable<T>> void sort(List<T> list) // ok but limited
```

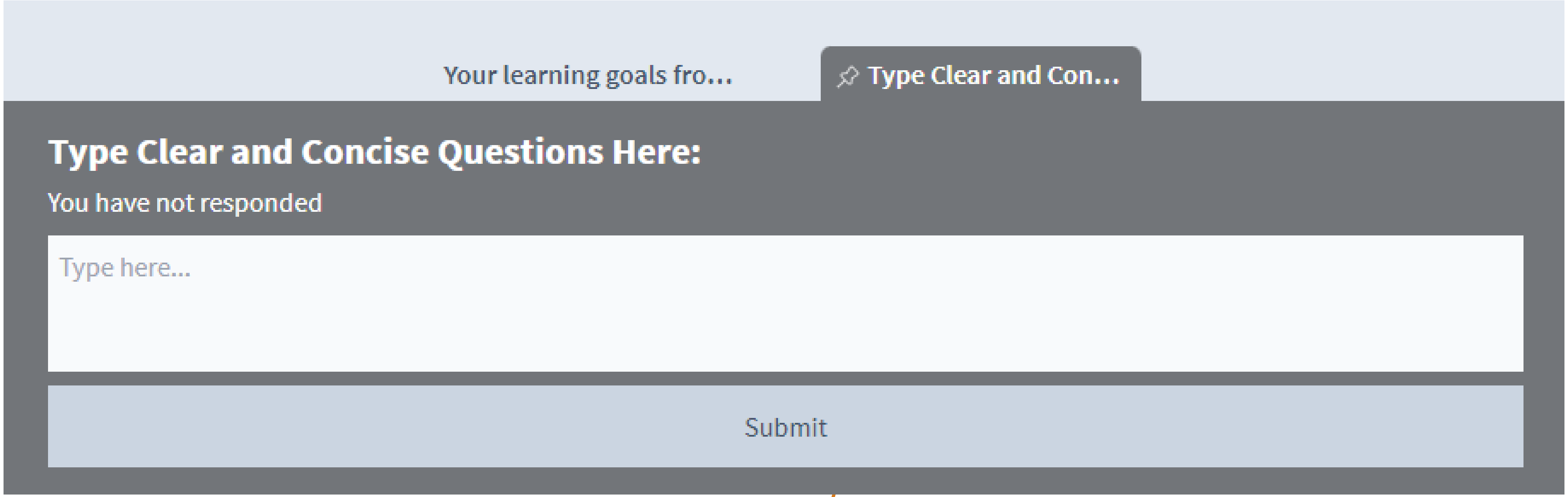
```
public static <T extends Comparable<? super T>> void sort(List<T> list) // best
```

Questions?

Questions…

Use **PollEv.com/adestine** for in-class Q&A

or just **Raise Hand**



Type Clear and Concise Questions Here:

You have not responded

03- Command Line Interface

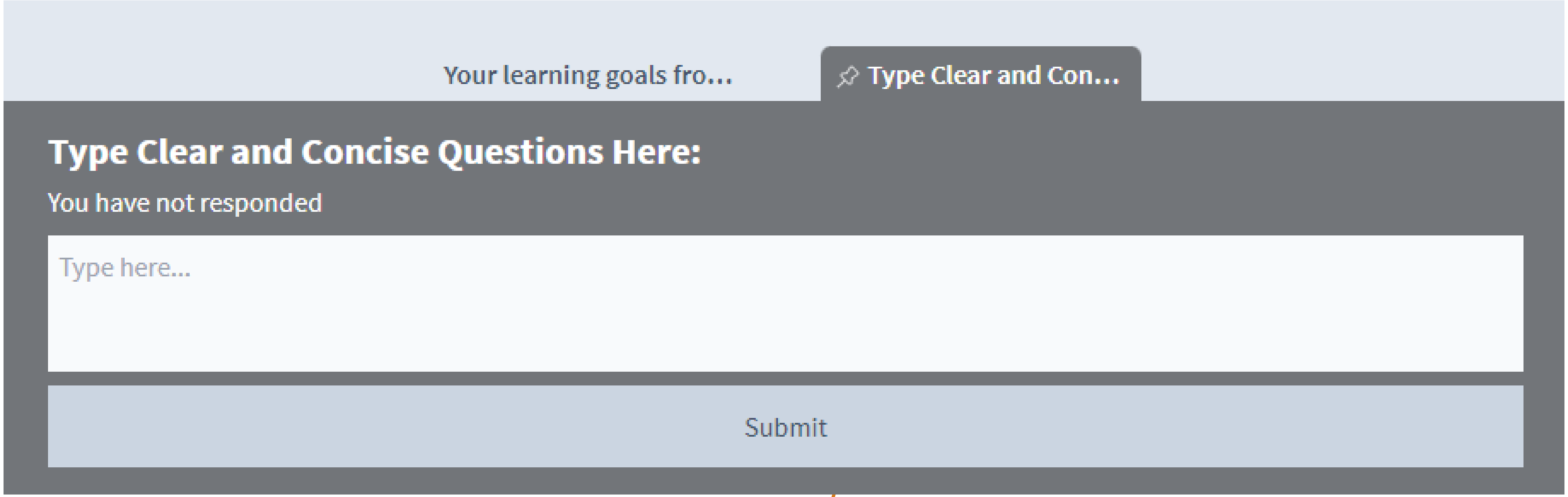
Whiteboard

Coding Illustration

Questions…

Use **PollEv.com/adestine** for in-class Q&A

or just **Raise Hand**



Type Clear and Concise Questions Here:

You have not responded

04- Computational Complexity

Algorithm Analysis

Algorithm (steps to solve a problem) VS

Algorithm Analysis (how good is the solution in terms of time, memory used, …)

Why are we discussing Algorithm in a Data Structures course?

Hint: DS → Objects **and Operations**

Two complexities

Time-complexity

- How much **time**

Space-complexity

- How much **memory**

More about these concepts (next lecture)

Next Lecture (Lecture 03)

1. **Efficient programming / Computational Analysis (more on this topic)**
2. **Iterators, Inner Classes**
3. **Dynamic Arrays**

Reminders:

Keep Working on Project_0

Do the readings (Ch. 6.1-6.3 and Ch. 15)