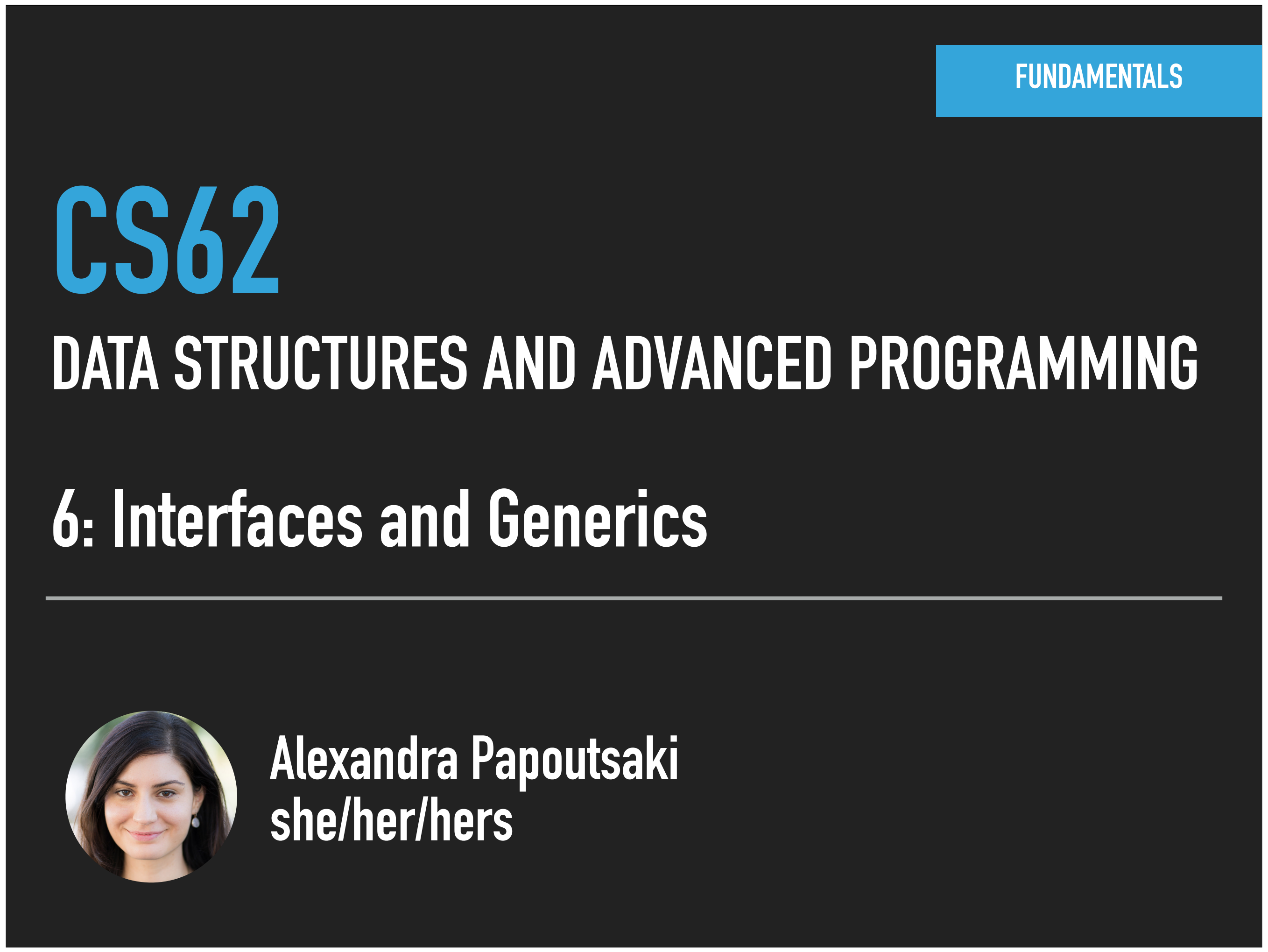


FUNDAMENTALS

CS62

DATA STRUCTURES AND ADVANCED PROGRAMMING

6: Interfaces and Generics

Alexandra Papoutsaki
she/her/hers

Today we will talk about interfaces and generics and we will start working towards understanding linear data structures.

Lecture 6: Interfaces and Generics

- Interfaces
- Background
- Generics

Let's start with interfaces, another core concept in object oriented programming.

Interfaces

- Contracts of what a class must do, not how to do it, abstracting from implementation.
- In Java, an interface is a reference type (like a class), that contains abstract methods and default methods.
- A class that implements an interface is obliged to implement its abstract methods.
- Interfaces cannot be instantiated (no `new` keyword). They can only be *implemented* by classes or *extended* by other interfaces.

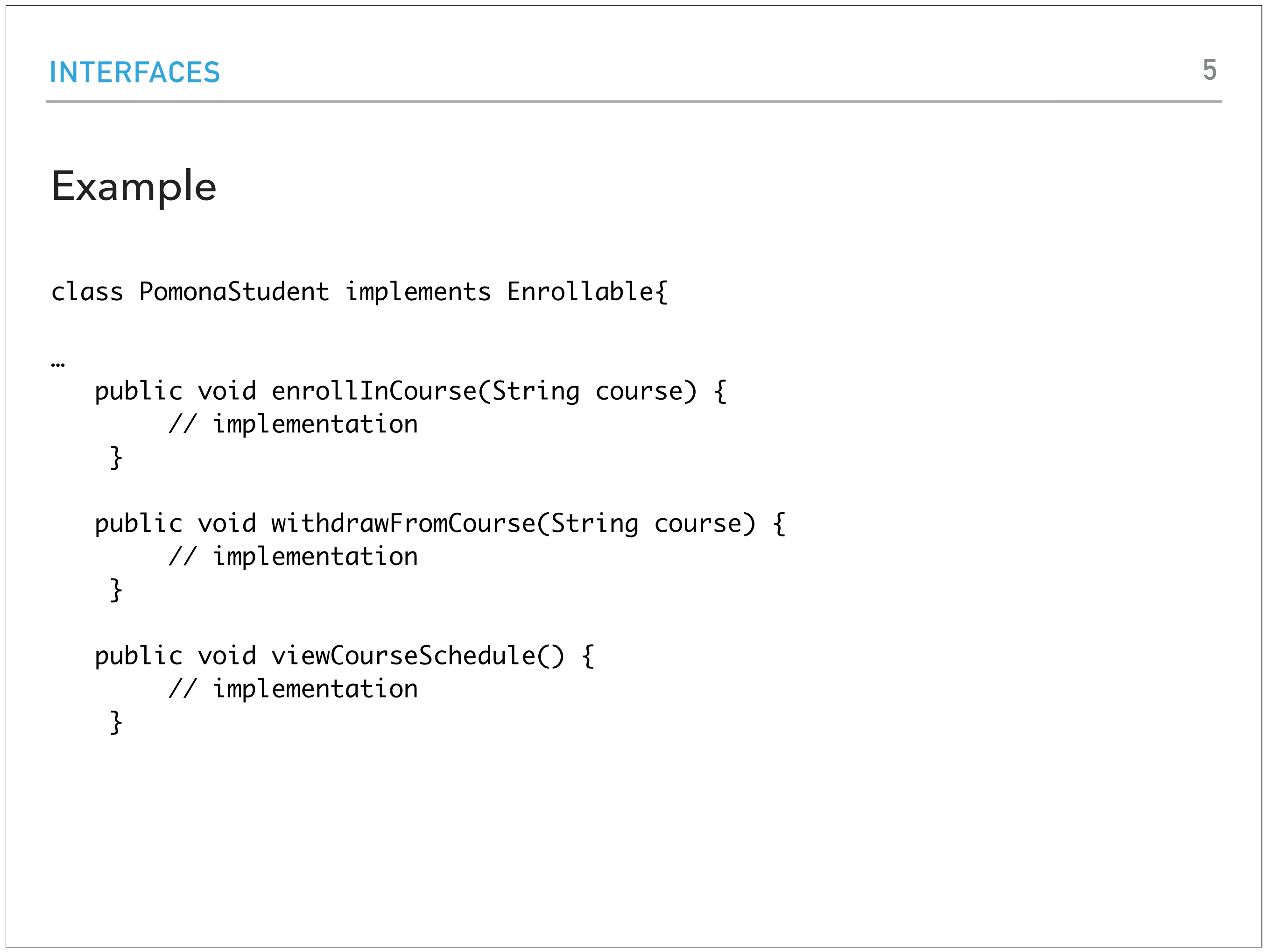
Interfaces are contracts that determine what a class must do without specifying how to do it. In Java, interfaces are reference types (just like classes). An interface contains abstract methods and default methods (and constants and static methods but we will ignore those here). A class that implements an interface is obliged to implement its abstract methods. Interfaces cannot be instantiated (no new keyword). They can only be implemented by classes or extended by other interfaces.

Example

```
public interface Enrollable{
    void enrollInCourse(String course);
    void withdrawFromCourse(String course);
    void viewCourseSchedule();

    default int getMaxCredits(){
        return 4;
    }
}
```

Here is an example of an interface in our running example of the registrar application. The interface Enrollable has three abstract methods (methods that don't have a body and just indicate what the signature should be) and one default method (methods that provide a default implementation).
An abstract method within an interface is followed by a semicolon, but no braces since it does not contain an implementation. Default methods are defined with the default modifier. All methods in an interface are implicitly public, so we can omit the public modifier.



Example

```
class PomonaStudent implements Enrollable{

…
   public void enrollInCourse(String course) {
        // implementation
    }

   public void withdrawFromCourse(String course) {
        // implementation
    }

   public void viewCourseSchedule() {
        // implementation
    }
```

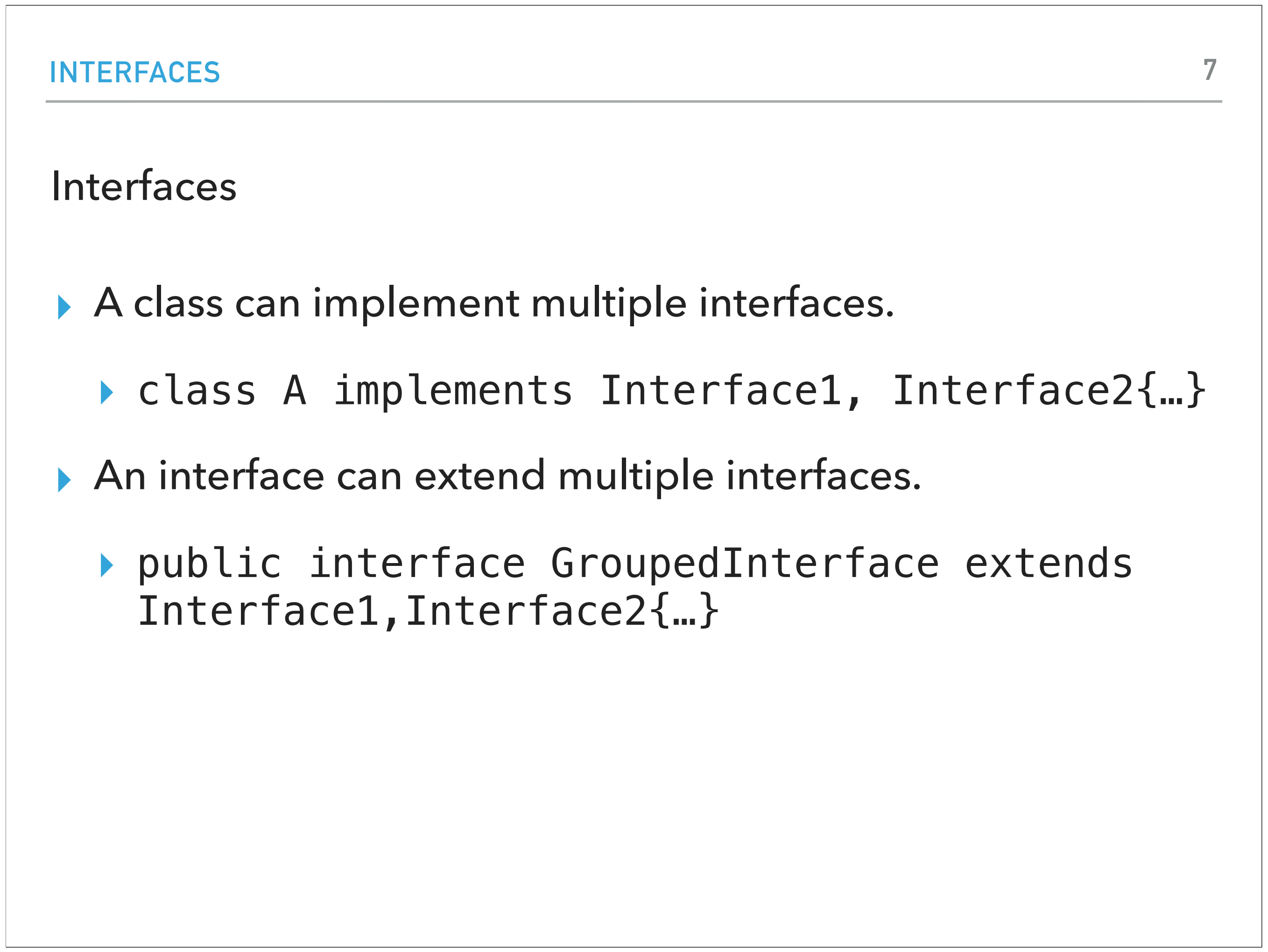
We can now have our PomonaStudent implement the Enrollable interface which forces it to implement the three abstract methods of the interface. Any object of type PomonaStudent now supports the default implementation of the default method getMaxCredits.

Example

```
class FourthYearPomonaStudent extends PomonaStudent{

…
   public int getMaxCredits(){
        return 6;
    }
}
```

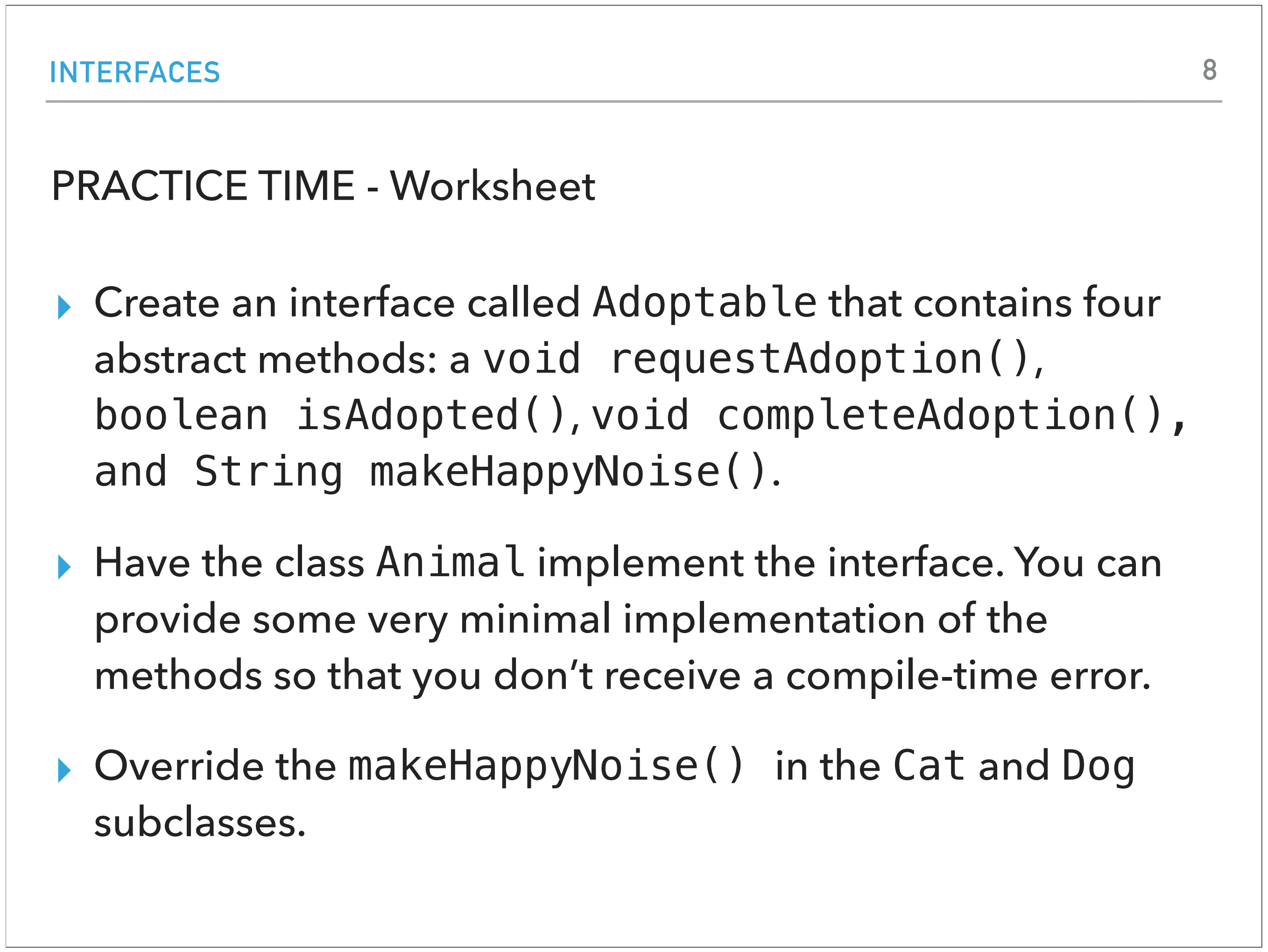
But we can also override a default method, for example, here, we return 6 instead of 4.



Interfaces

- A class can implement multiple interfaces.
  - `class A implements Interface1, Interface2{…}`
- An interface can extend multiple interfaces.
  - `public interface GroupedInterface extends Interface1,Interface2{…}`

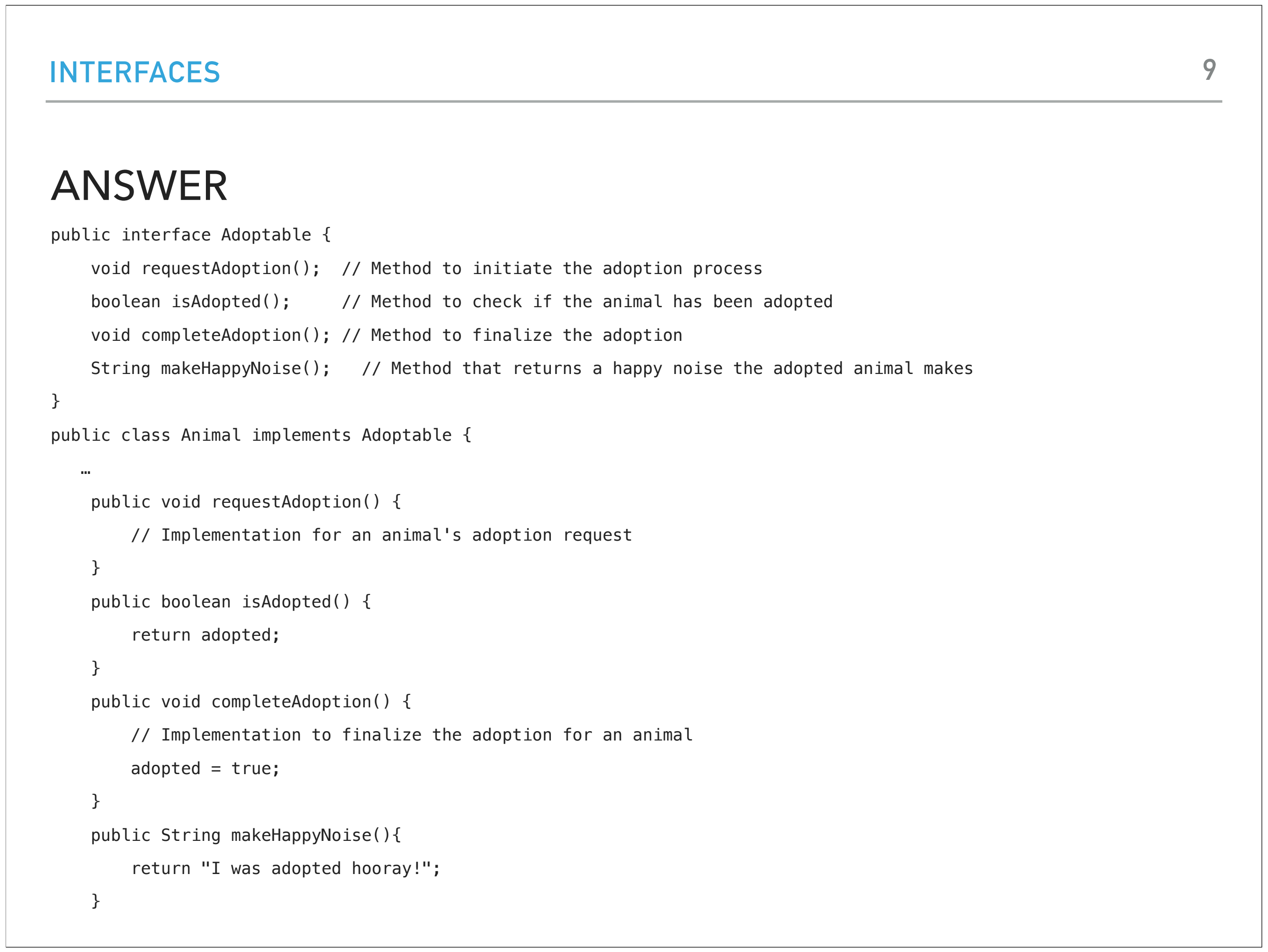
Interfaces are quite convenient and in contrast to inheritance where a class can only extend one class, here, a class can implement multiple interfaces, separated by comma. In fact, interfaces themselves can extend multiple interfaces.



PRACTICE TIME - Worksheet

- Create an interface called `Adoptable` that contains four abstract methods: a `void requestAdoption()`, `boolean isAdopted()`, `void completeAdoption()`, `and String makeHappyNoise()`.
- Have the class `Animal` implement the interface. You can provide some very minimal implementation of the methods so that you don't receive a compile-time error.
- Override the `makeHappyNoise()` in the `Cat` and `Dog` subclasses.

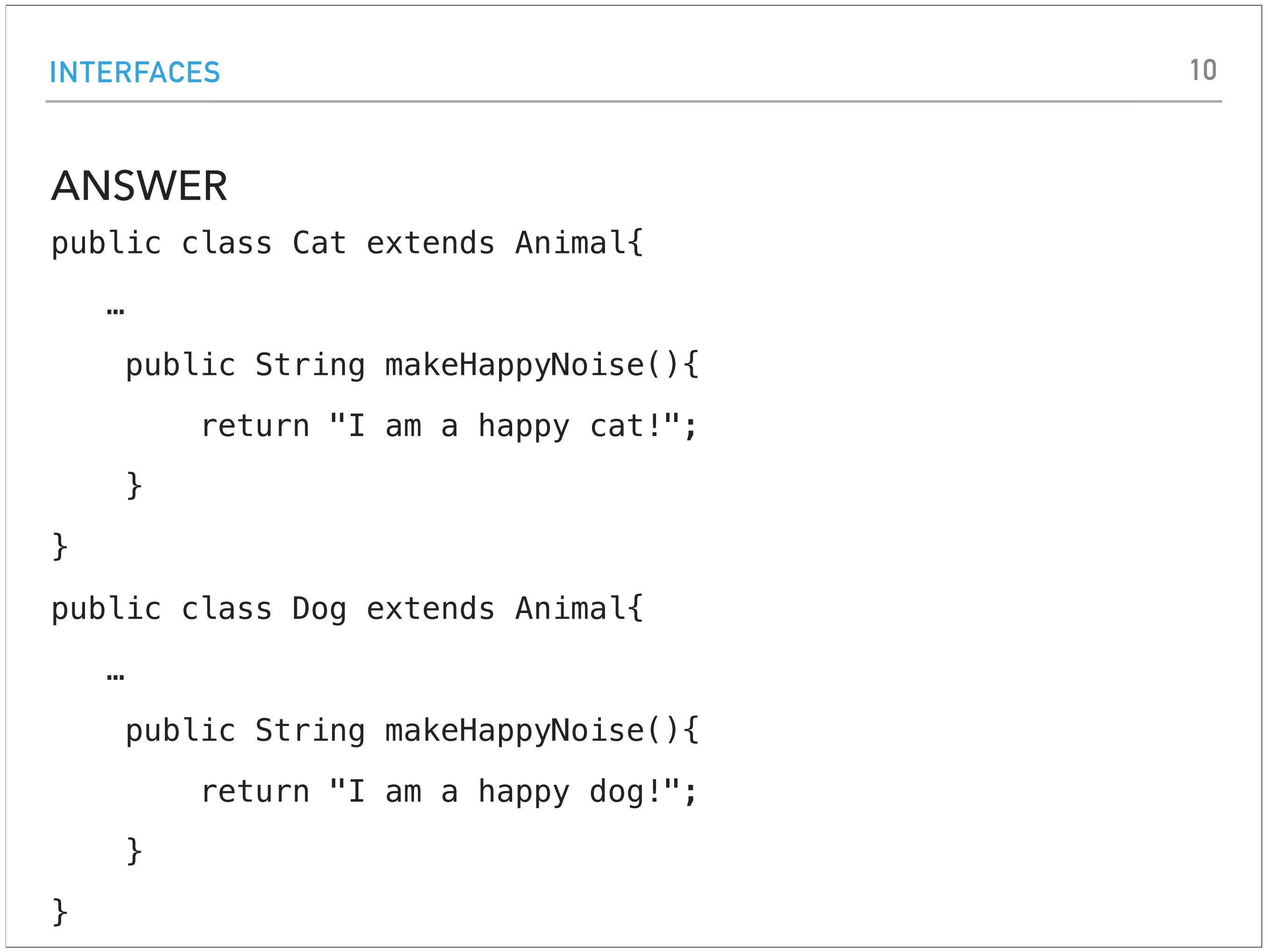
Let's now quickly create an interface called Adoptable that contains four abstract methods: a void requestAdoption(), boolean isAdopted(), void completeAdoption(), and String makeHappyNoise().
Have the class Animal implement the interface. You can provide some very minimal implementation of the methods so that you don't receive a compile-time error.
Override the makeHappyNoise() in the Cat and Dog subclasses.



ANSWER

```
public interface Adoptable {
    void requestAdoption();  // Method to initiate the adoption process
    boolean isAdopted();     // Method to check if the animal has been adopted
    void completeAdoption(); // Method to finalize the adoption
    String makeHappyNoise();   // Method that returns a happy noise the adopted animal makes
}
public class Animal implements Adoptable {
   …
    public void requestAdoption() {
        // Implementation for an animal's adoption request
    }
    public boolean isAdopted() {
        return adopted;
    }
    public void completeAdoption() {
        // Implementation to finalize the adoption for an animal
        adopted = true;
    }
    public String makeHappyNoise(){
        return "I was adopted hooray!";
    }
```

Here is one such attempt that conveys the basic idea of interfaces.



ANSWER

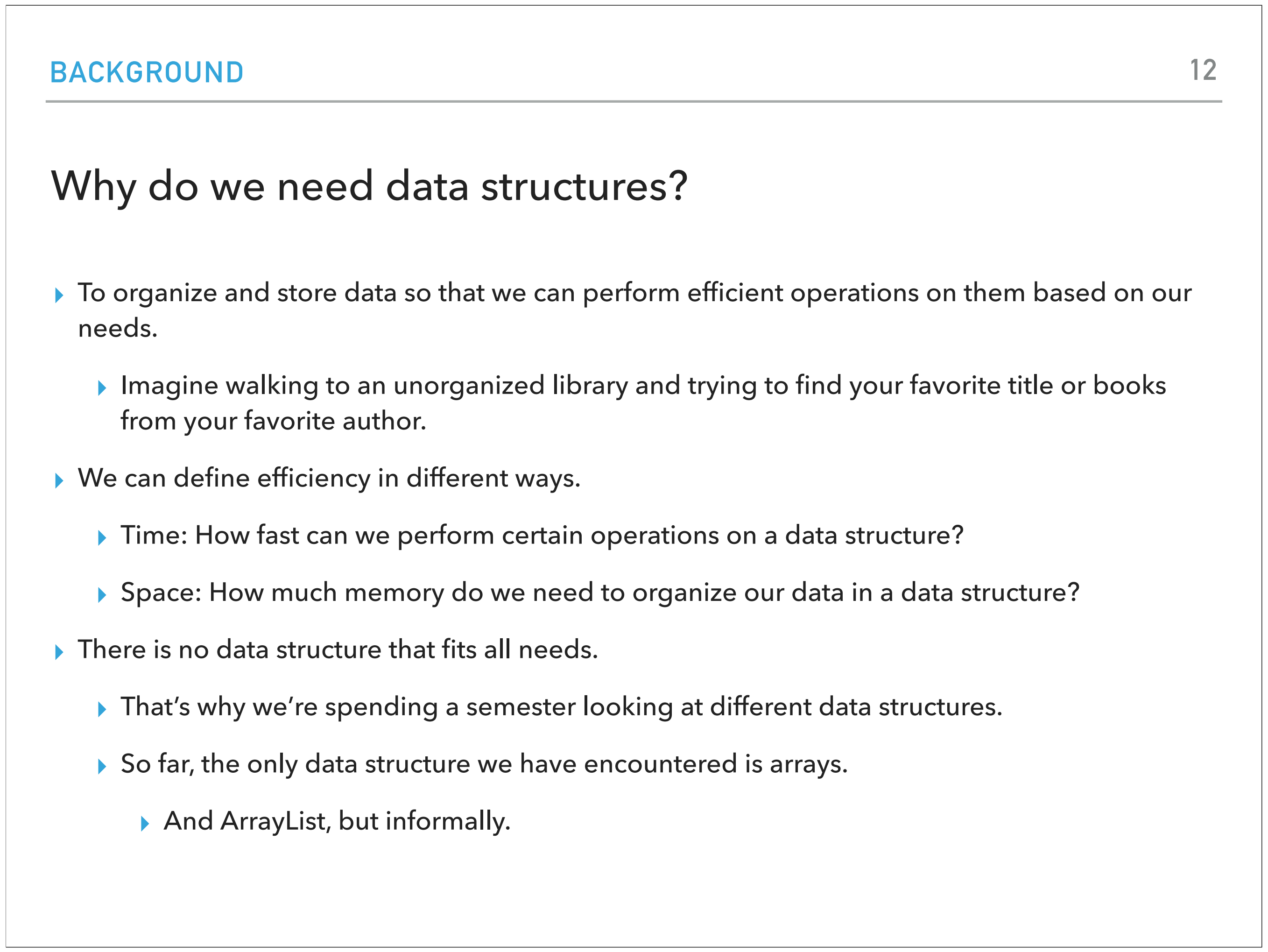
```
public class Cat extends Animal{
   …
    public String makeHappyNoise(){
        return "I am a happy cat!";
    }
}
public class Dog extends Animal{
   …
    public String makeHappyNoise(){
        return "I am a happy dog!";
    }
}
```

Here is one such attempt that conveys the basic idea of interfaces.

Lecture 6: Interfaces and Generics

- Interfaces
- Background
- Generics

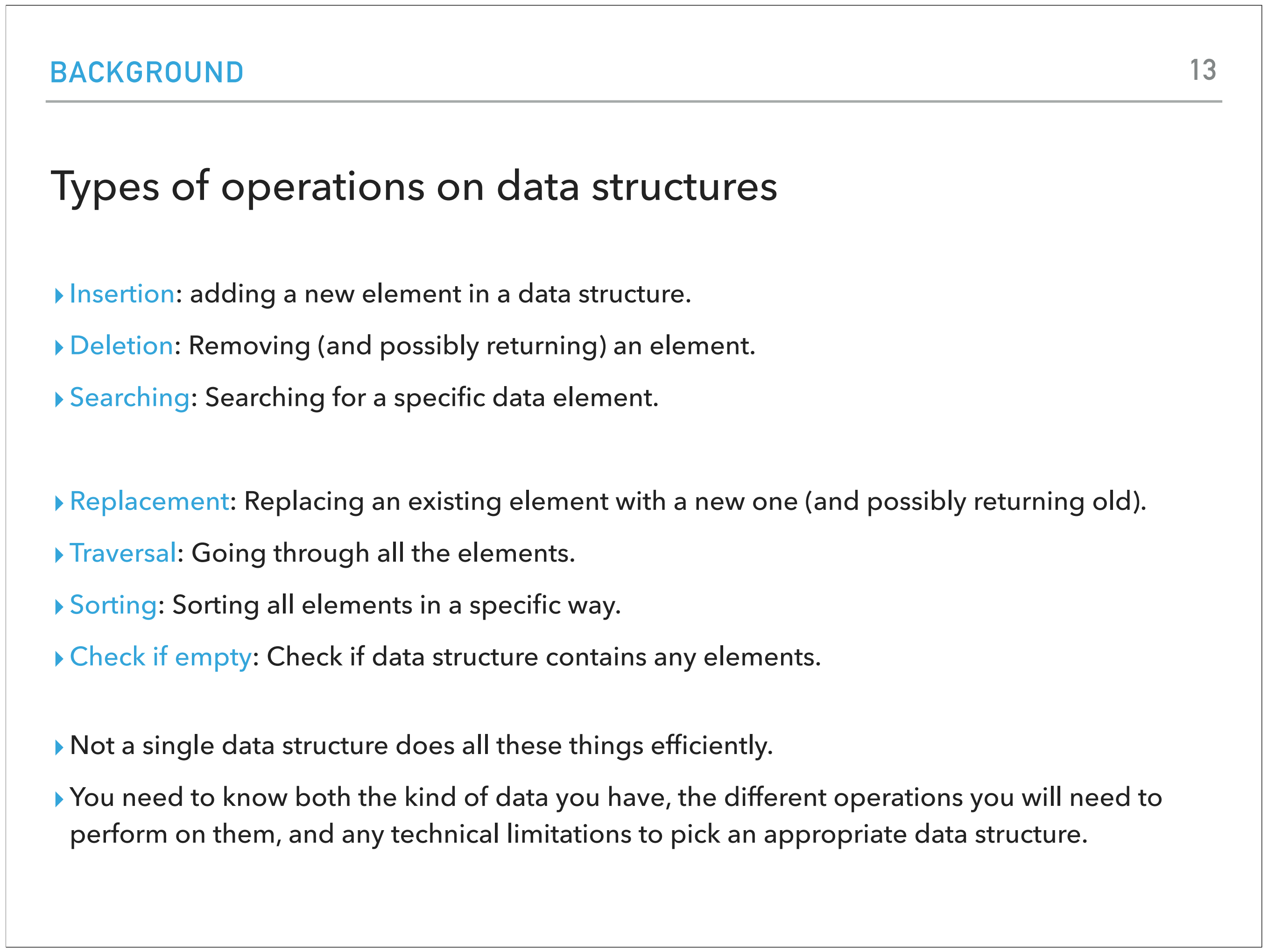
Now let's see an example of where interfaces are useful in data structures. For that, we will need to have a conversation first about the different types of data structures.



Why do we need data structures?

- To organize and store data so that we can perform efficient operations on them based on our needs.
  - Imagine walking to an unorganized library and trying to find your favorite title or books from your favorite author.
- We can define efficiency in different ways.
  - Time: How fast can we perform certain operations on a data structure?
  - Space: How much memory do we need to organize our data in a data structure?
- There is no data structure that fits all needs.
  - That's why we're spending a semester looking at different data structures.
  - So far, the only data structure we have encountered is arrays.
    - And ArrayList, but informally.

We use data structures to organize and store data in an efficient way based on our needs. Efficiency can be defined either in terms of time or space. There is no data structure that fits all needs and this course is all about the trade offs based on our circumstances. So far, the only data structure we have seen is the array (and the ArrayList in the lab).



Types of operations on data structures

- Insertion: adding a new element in a data structure.
- Deletion: Removing (and possibly returning) an element.
- Searching: Searching for a specific data element.

- Replacement: Replacing an existing element with a new one (and possibly returning old).
- Traversal: Going through all the elements.
- Sorting: Sorting all elements in a specific way.
- Check if empty: Check if data structure contains any elements.

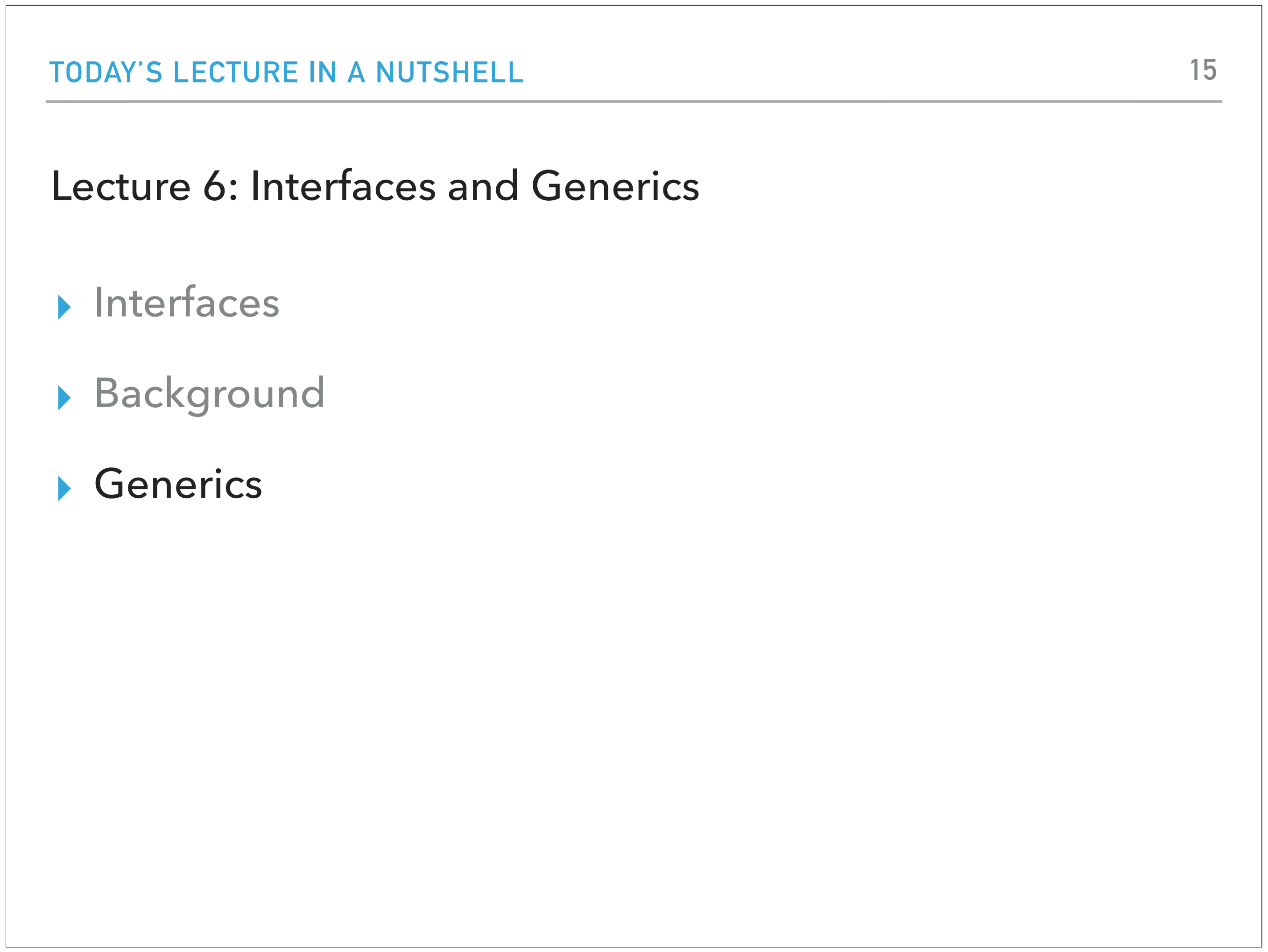
- Not a single data structure does all these things efficiently.
- You need to know both the kind of data you have, the different operations you will need to perform on them, and any technical limitations to pick an appropriate data structure.

When thinking of what operations we want a data structure to support, we quickly come to the conclusion that we would like to add new elements to it, remove (and possibly return) an element, and search for a specific element. We might also like some fancier operations, like replacing elements, going through all the elements, sorting our data structure, checking if it is empty, etc. As we already saw, not a single data structure does all these things efficiently. You need to know both the kind of data you have, the different operations you will need to perform on them, and any technical limitations to pick an appropriate data structure.

Linear vs non-linear data structures

- Linear: elements arranged in a linear sequence based on a specific order.
  - E.g., Arrays, ArrayLists, linked lists, stacks, queues.
  - Linear memory allocation: all elements are placed in a contiguous block of memory. E.g., arrays and ArrayLists.
  - Use of pointers/links: elements don't need to be placed in contiguous blocks. The linear relationship is formed through pointers. E.g., singly and doubly linked lists.
- Non-linear: elements arranged in non-linear, mostly hierarchical relationship.
  - E.g., trees and graphs.

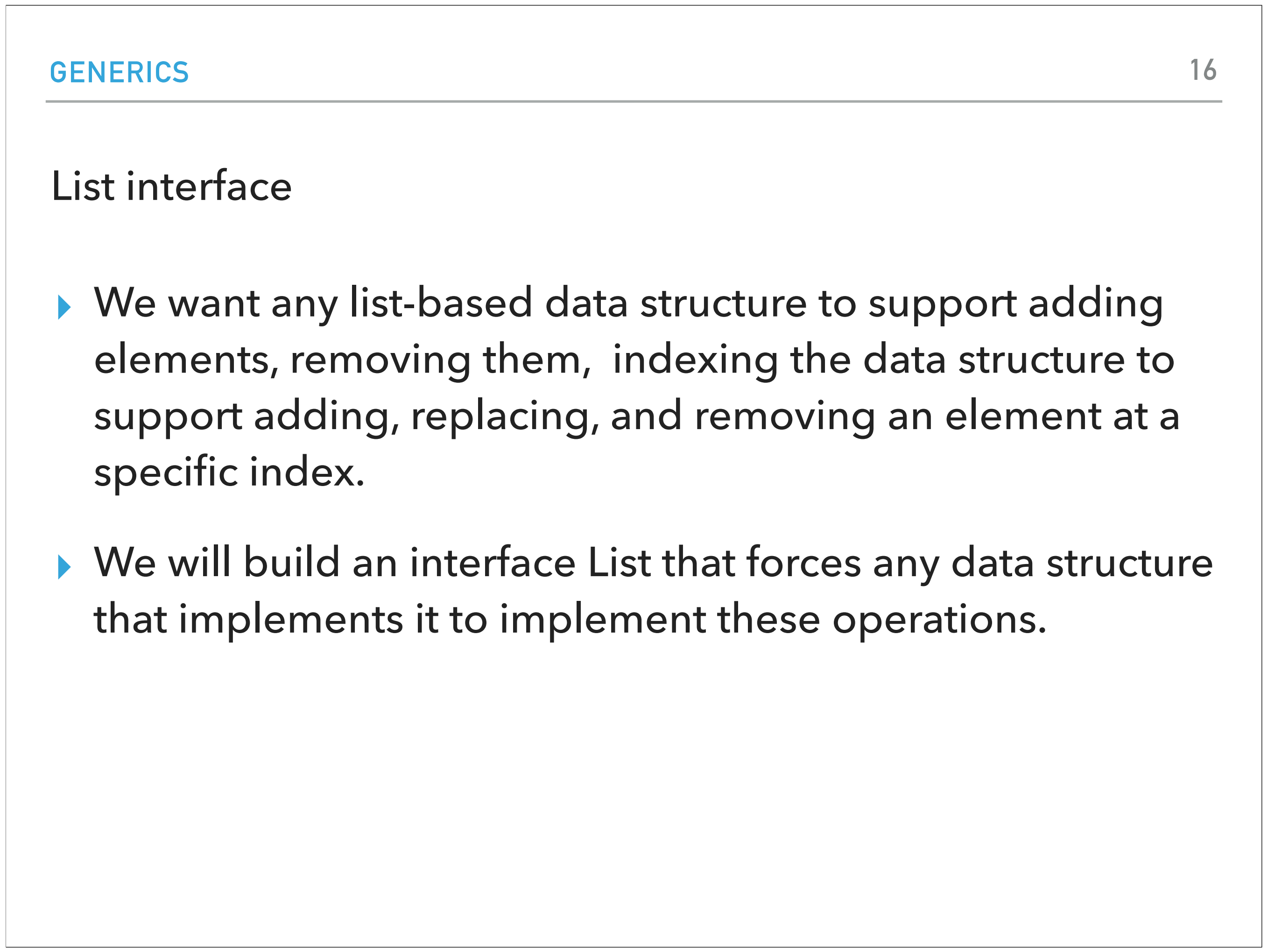
Data structures are divided in two main families: linear and non-linear. In linear data structures, elements are arranged in a linear sequence and there is a notion of order. Most data structures we will see (E.g., arrays, array lists, linked lists, stacks, and queues) are linear. Linear data structures can either rely on linear memory allocation, where all elements are placed in one contiguous block of memory (e.g., in arrays and array lists), or we might use pointers/links to form this linear relationship across different memory locations (this is what happens with singly and doubly linked lists). In non-linear data structures, there is typically a hierarchical relationship. We will see trees and graphs in this family.



Lecture 6: Interfaces and Generics

- Interfaces
- Background
- Generics

Let's focus on the first family of data structures, the linear data structures. Let's assume we are interested in building a data structure that resembles a list.



List interface

- We want any list-based data structure to support adding elements, removing them, indexing the data structure to support adding, replacing, and removing an element at a specific index.
- We will build an interface List that forces any data structure that implements it to implement these operations.

We want any list-based data structure to support adding elements, removing them, indexing the data structure to support adding, replacing, and removing an element at a specific index. There are multiple ways to achieve this but for now, we won't focus on the implementation but just on the fact that we want our data structure to support this operations. This is where interfaces can be perfect. We will build an interface List that forces any data structure that implements it to implement these operation.

Lists should support any type of element

- We want our data structure to support any type of elements, as long as they are of the same type. We could use the class Object but this requires casting to the desired type:

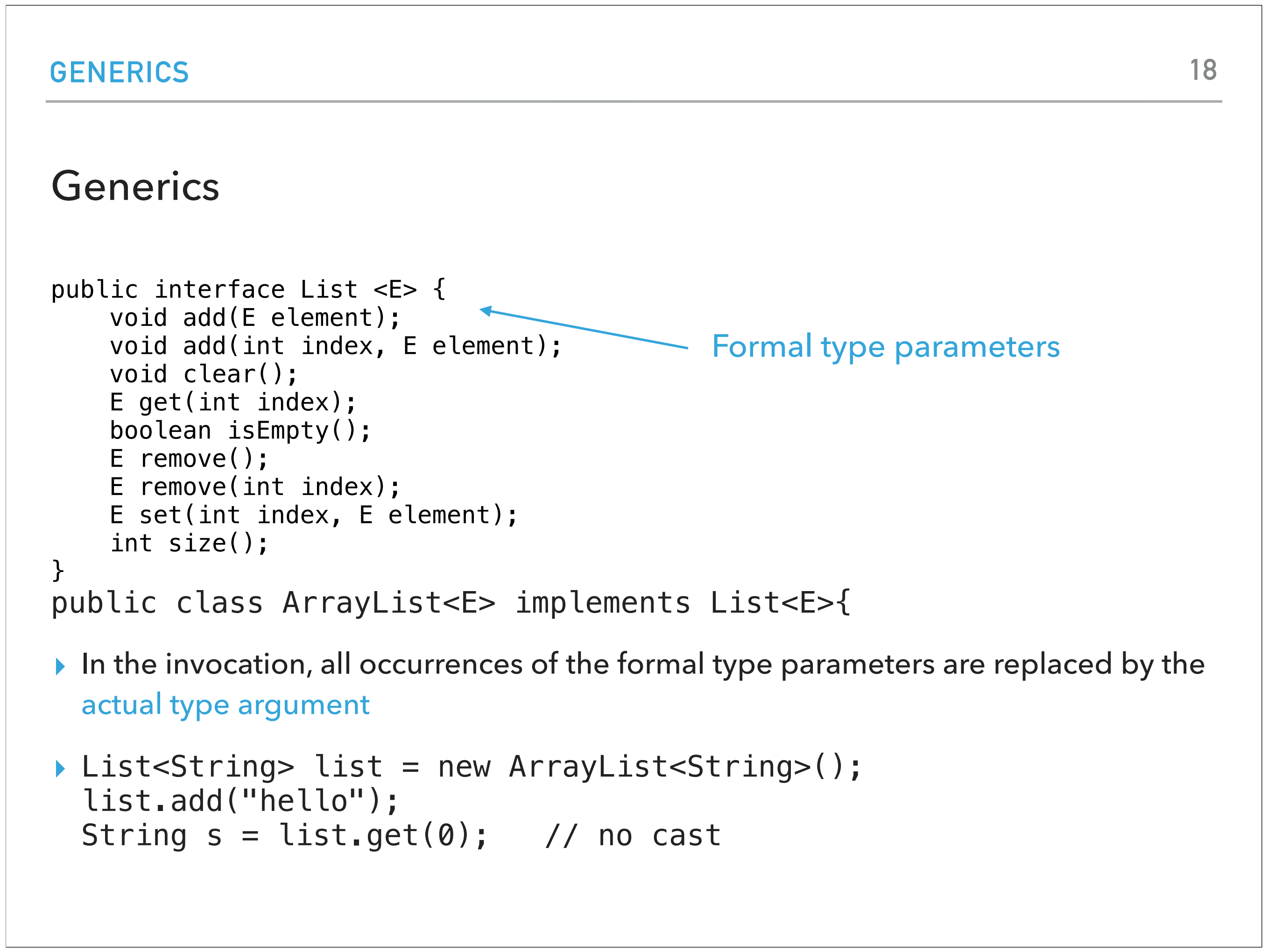
```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

- Instead, we will use generics.

A consideration we should take into account, is that we want our data structure to support any type of elements, as long as they are of the same type. We could use the class Object but this requires casting to the desired type:

```
  List list = new ArrayList();
  list.add("hello");
  String s = (String) list.get(0);
```

Instead, we will use generics.



Generics

```
public interface List <E> {
    void add(E element);
    void add(int index, E element);
    void clear();
    E get(int index);
    boolean isEmpty();
    E remove();
    E remove(int index);
    E set(int index, E element);
    int size();
}
public class ArrayList<E> implements List<E>{
```

Formal type parameters

- In the invocation, all occurrences of the formal type parameters are replaced by the actual type argument

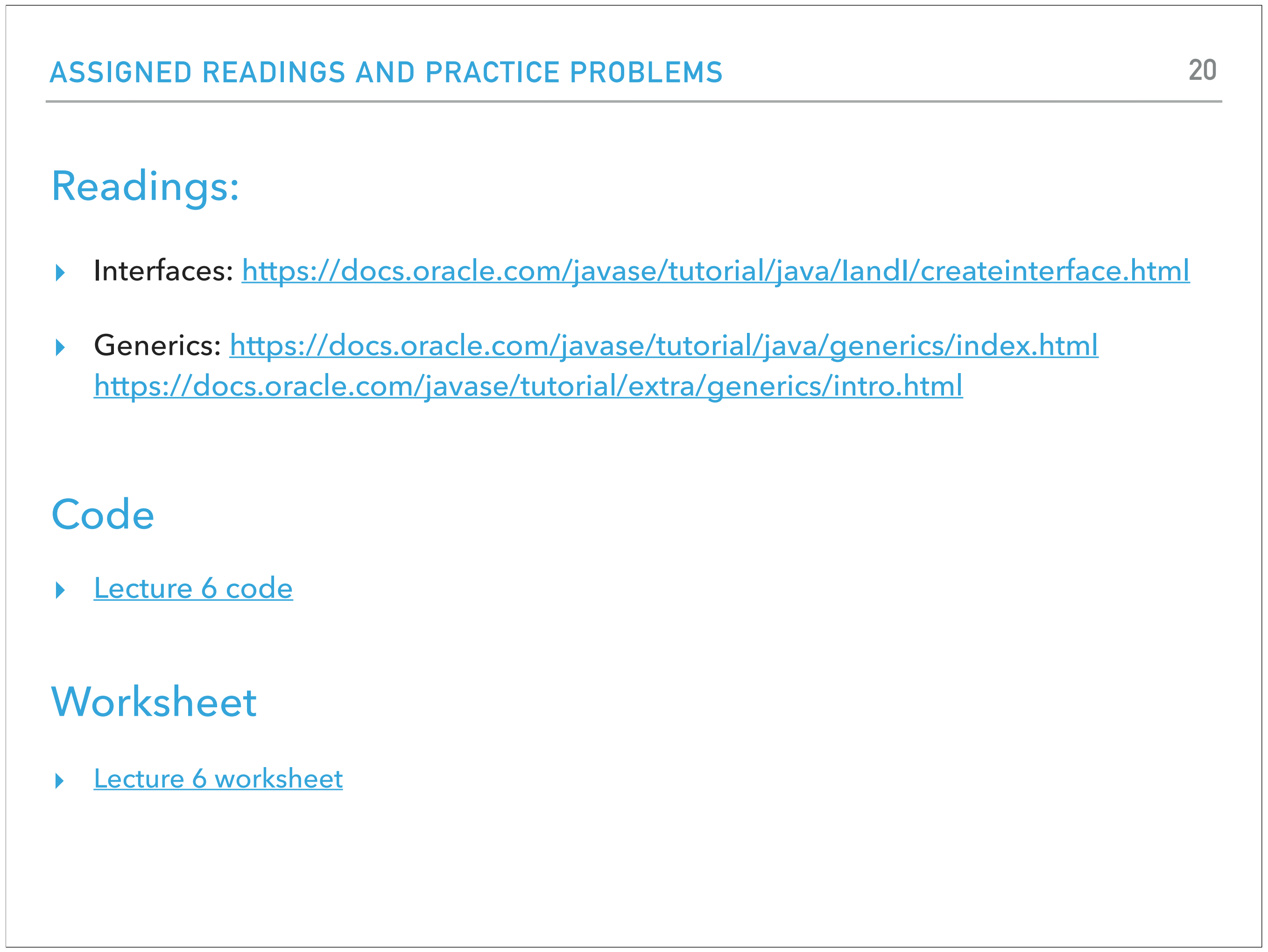
-
  ```
  List<String> list = new ArrayList<String>();
  list.add("hello");
  String s = list.get(0);   // no cast
  ```

As we saw in the lab, to work with generics we will use the diamond bracket notation, Let's say we have the interface List and we want it to support methods add, clear, etc. We will define a formal type parameter E which will act as a placeholder for any actual type argument at the time of invocation. For example, if we have a class ArrayList that implements the List interface, we see that the ArrayList class can work with any type of element, as long as they are of the same type. When instantiating a specific list, we have to say what the actual type is, e.g., String. List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
This does not require casting!

Generics

- Generics enable types (that is classes and interfaces) to be used as parameters when defining classes, interfaces, and methods.
- E: element (common in data structures), T: type, K: key, V: value, N: number.
- The additional advantage is that bugs are now caught at compile time instead of runtime (much easier to fix!)

That’s particularly important because it allows us to find bugs at compile time instead of runtime. Generics enable both classes and interfaces to be used as parameters when defining classes (E.g., ArrayList<E>), interfaces (e.g., List<E>), and methods (e.g., add(E element)). The recommendation is we use a single letter for the formal type parameter. E is very common in data structures and stands for element. Others you might encounter are T: type, K: key, V: value, N: number.



ASSIGNED READINGS AND PRACTICE PROBLEMS

Readings:

- Interfaces: https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html
- Generics: https://docs.oracle.com/javase/tutorial/java/generics/index.html
  https://docs.oracle.com/javase/tutorial/extra/generics/intro.html

Code

- Lecture 6 code

Worksheet

- Lecture 6 worksheet

And that's it. Today's meeting concludes our introduction to Java. From now on we know the basics of the language and OOP and we are ready to start learning about data structures!