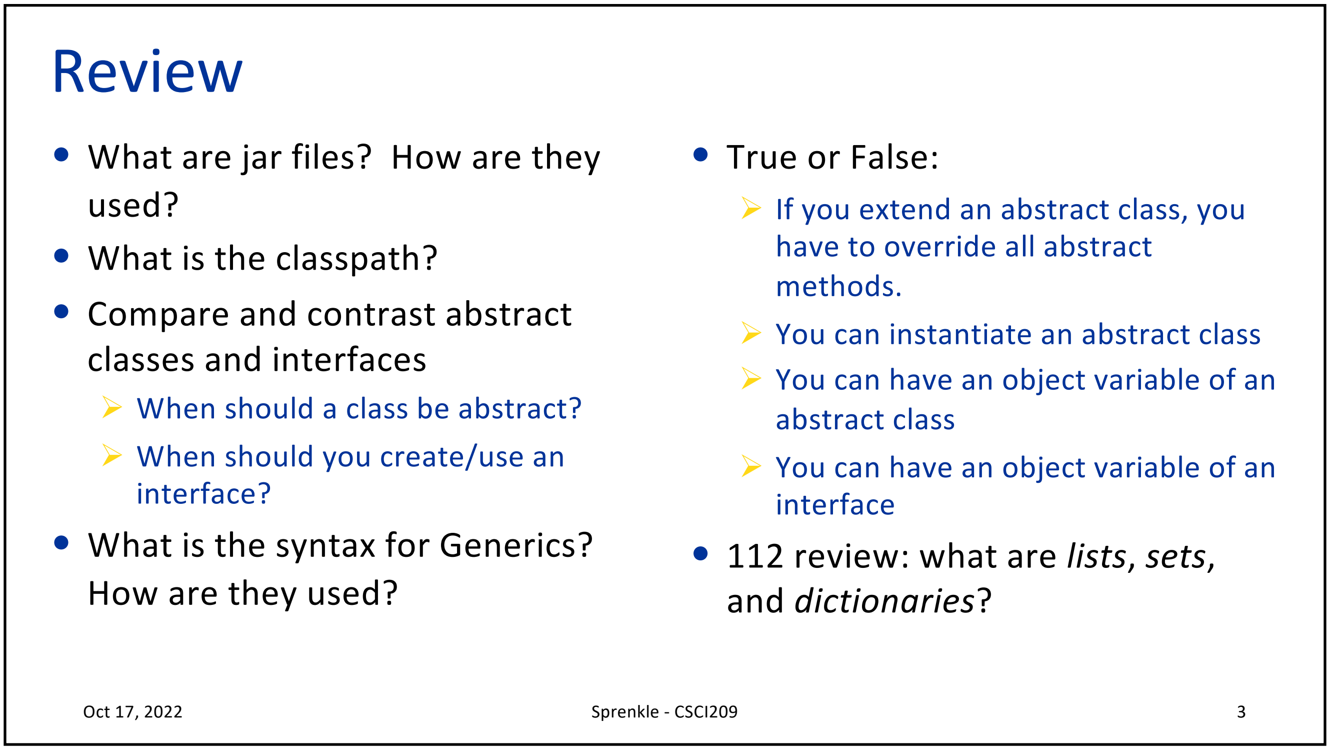
Objectives

- Collection Framework

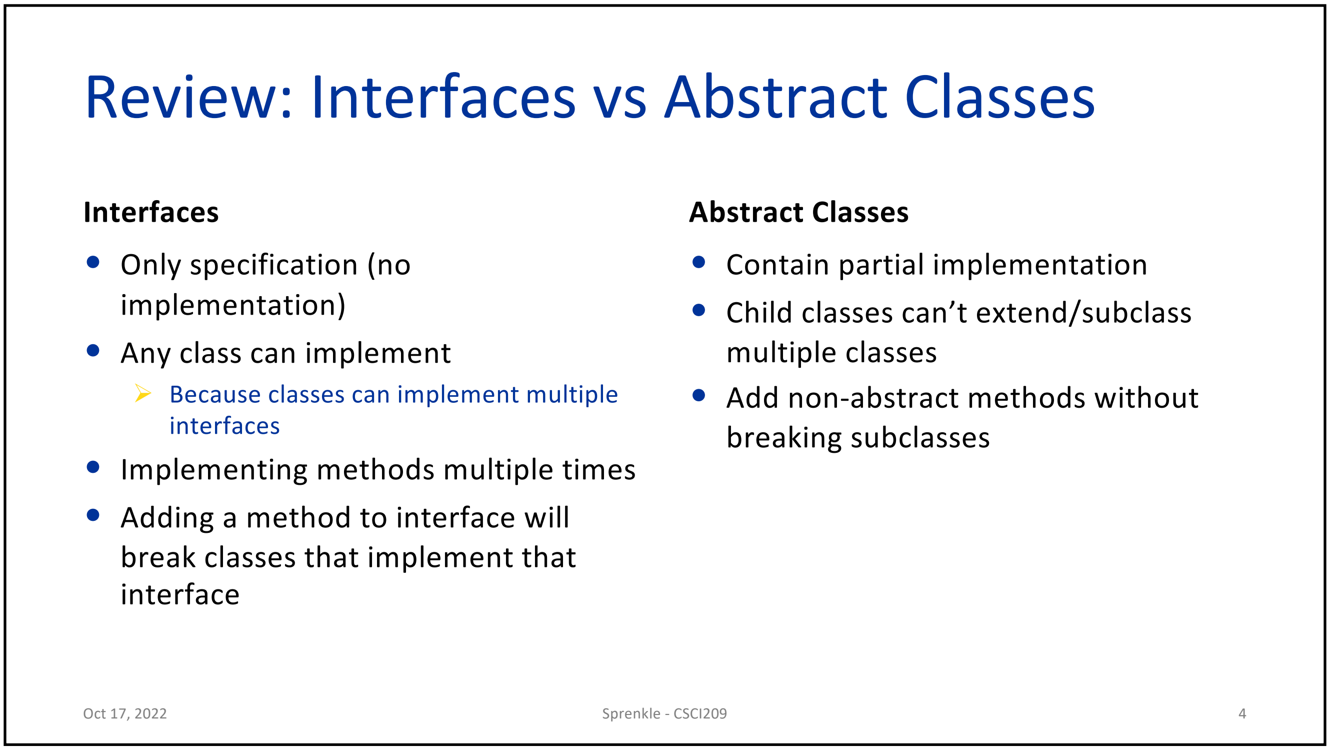
Iteration over Code: Assignment 4

- Demonstrates typical design/implementation process
  - ➢ Start with original code design
    - Inheritance from GamePiece class
  - ➢ Realize it could be designed better
    - Make GamePiece class abstract
    - Use an array of GamePiece objects
    - Easier to add new functionality to Game
- Major part of problem-solving is figuring out how to break problem into smaller pieces
- Reminders
  - ➢ Heed my warnings
  - ➢ Start simple, small (e.g., Goblin only moves left)



Review

- What are jar files? How are they used?
- What is the classpath?
- Compare and contrast abstract classes and interfaces
  - When should a class be abstract?
  - When should you create/use an interface?
- What is the syntax for Generics? How are they used?
- True or False:
  - If you extend an abstract class, you have to override all abstract methods.
  - You can instantiate an abstract class
  - You can have an object variable of an abstract class
  - You can have an object variable of an interface
- 112 review: what are *lists*, *sets*, and *dictionaries*?



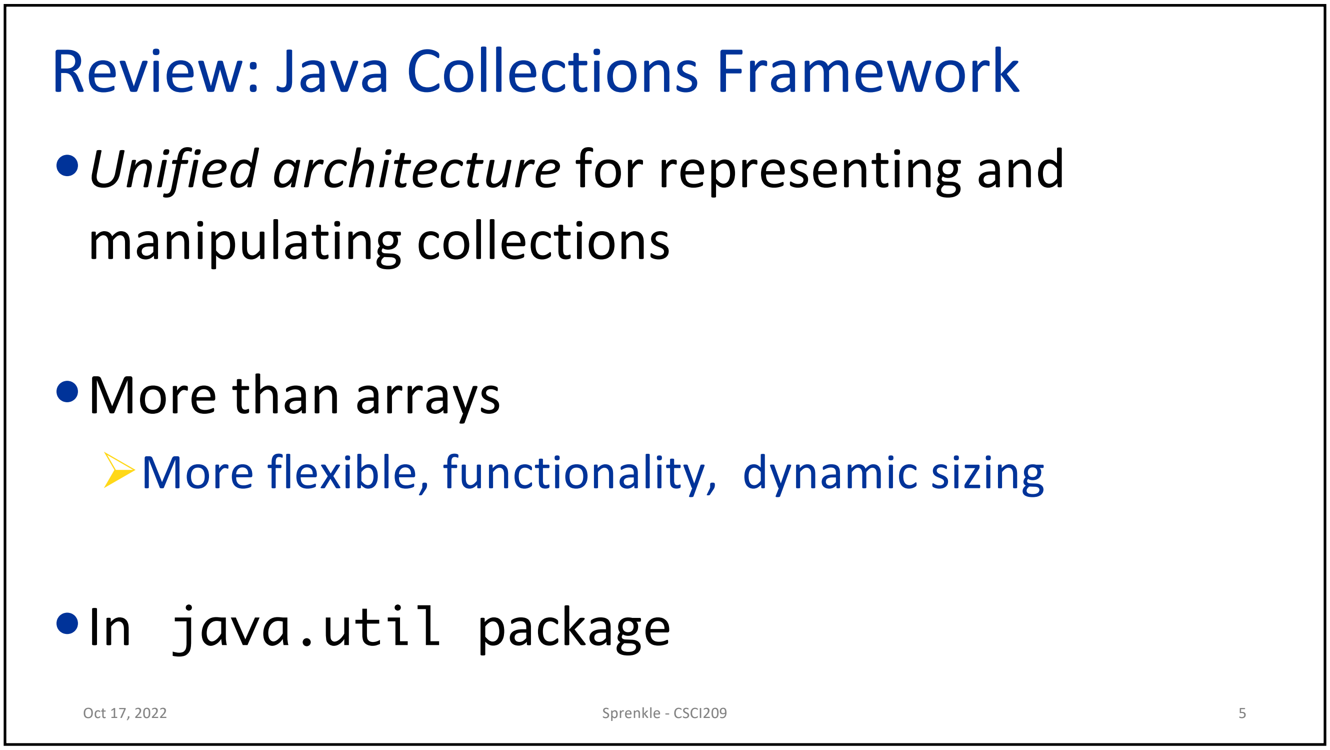
Review: Interfaces vs Abstract Classes

Interfaces

- Only specification (no implementation)
- Any class can implement
  - Because classes can implement multiple interfaces
- Implementing methods multiple times
- Adding a method to interface will break classes that implement that interface

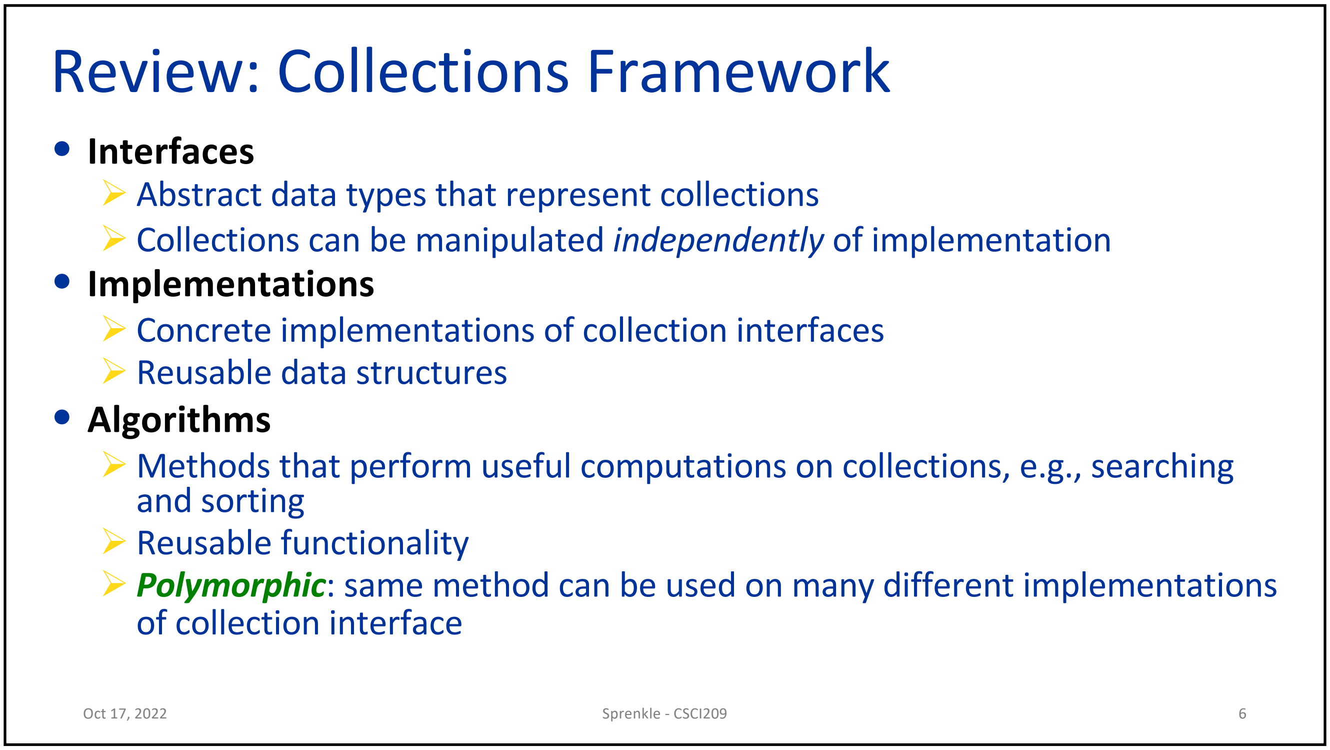
Abstract Classes

- Contain partial implementation
- Child classes can't extend/subclass multiple classes
- Add non-abstract methods without breaking subclasses



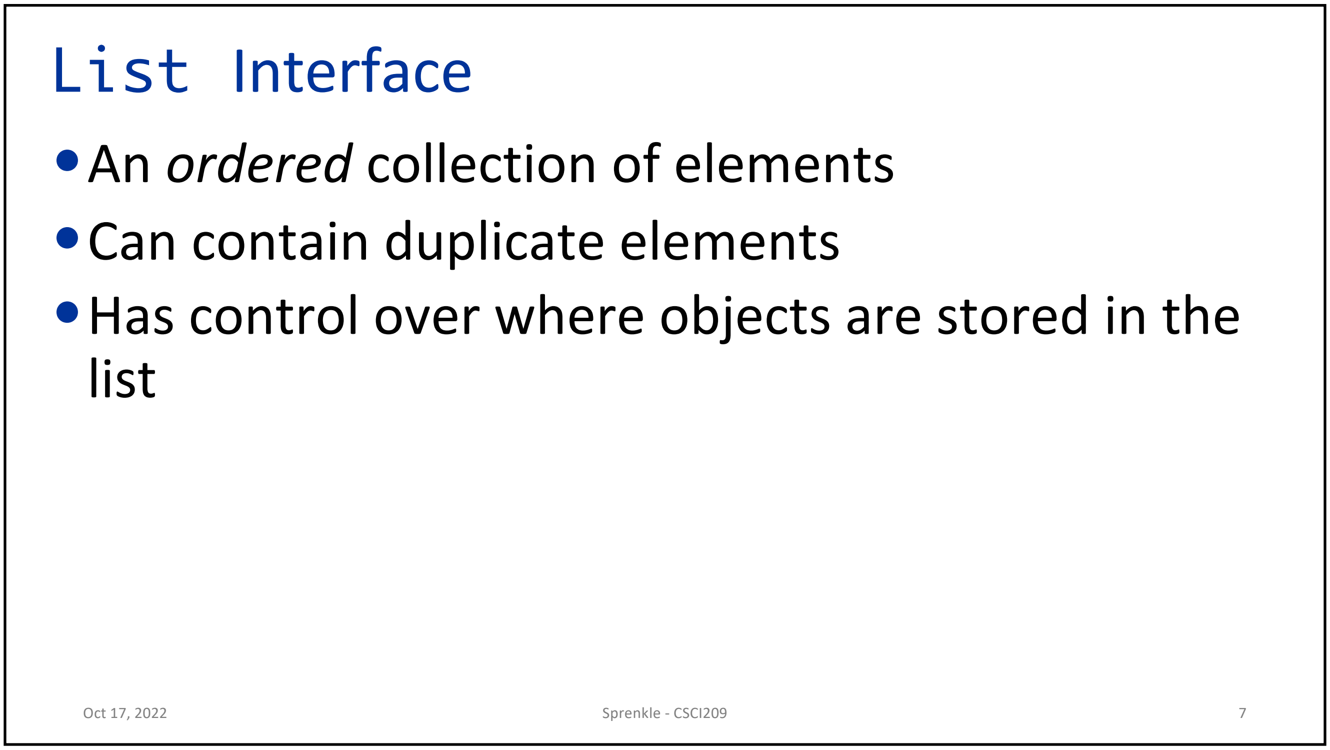
Review: Java Collections Framework

- *Unified architecture* for representing and manipulating collections
- More than arrays
  - More flexible, functionality, dynamic sizing
- In `java.util` package



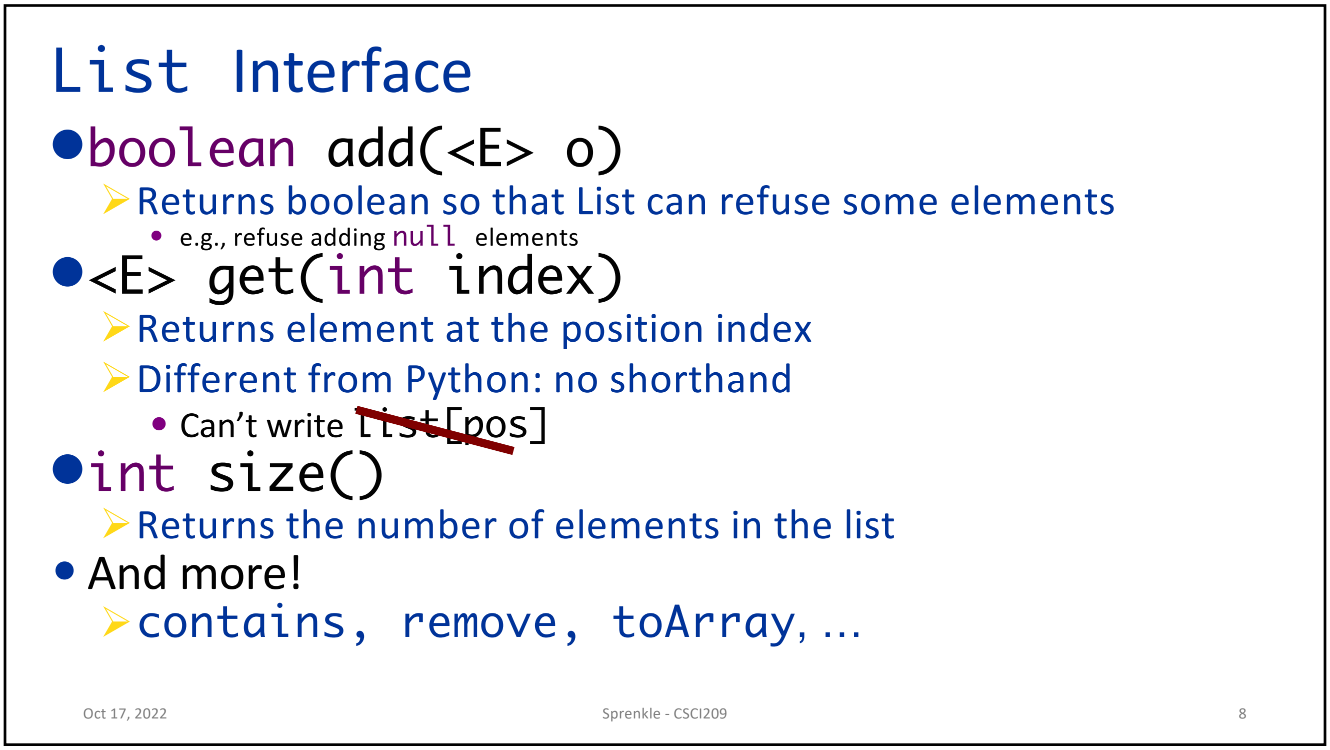
Review: Collections Framework

- **Interfaces**
  - Abstract data types that represent collections
  - Collections can be manipulated *independently* of implementation
- **Implementations**
  - Concrete implementations of collection interfaces
  - Reusable data structures
- **Algorithms**
  - Methods that perform useful computations on collections, e.g., searching and sorting
  - Reusable functionality
  - ***Polymorphic***: same method can be used on many different implementations of collection interface



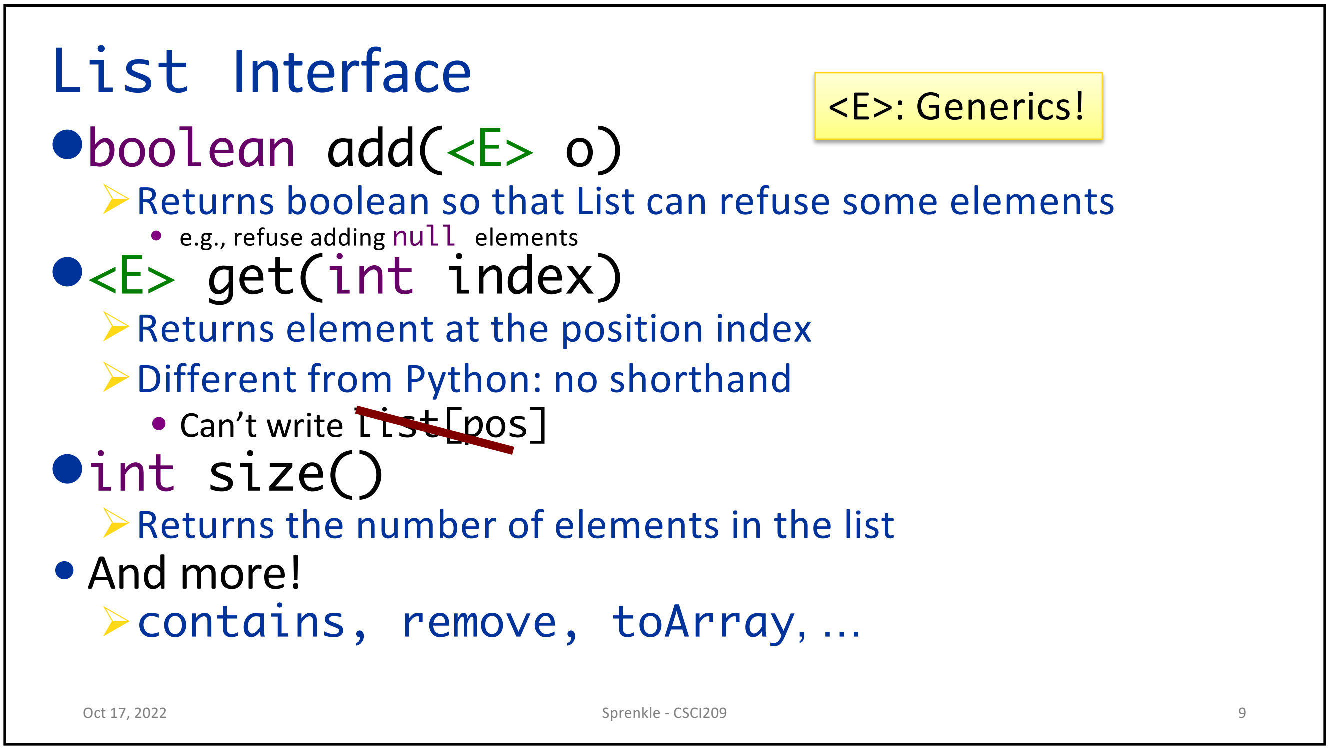
List Interface

- An *ordered* collection of elements
- Can contain duplicate elements
- Has control over where objects are stored in the list



List Interface

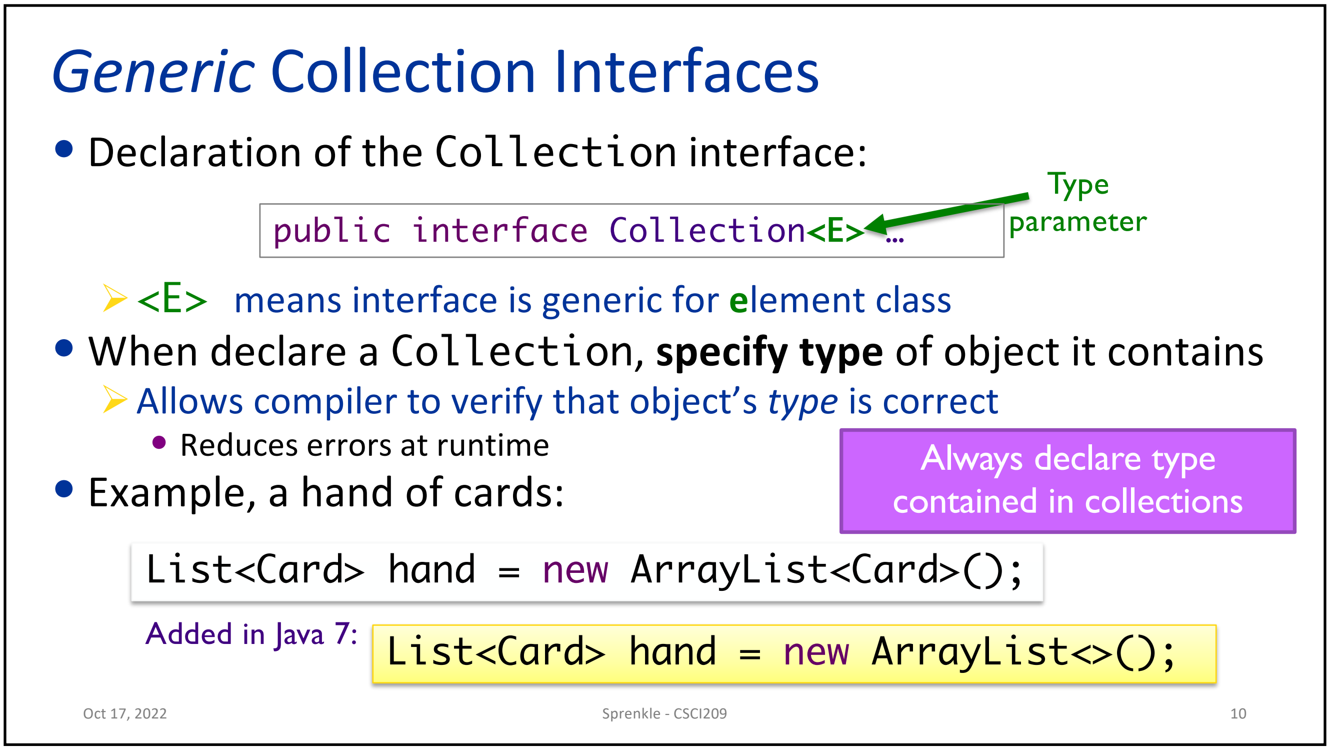
- `boolean add(<E> o)`
  - Returns boolean so that List can refuse some elements
    - e.g., refuse adding `null` elements
- `<E> get(int index)`
  - Returns element at the position index
  - Different from Python: no shorthand
    - Can't write ~~`list[pos]`~~
- `int size()`
  - Returns the number of elements in the list
- And more!
  - `contains, remove, toArray`, …



List Interface

<E>: Generics!

- boolean add(<E> o)
  - Returns boolean so that List can refuse some elements
    - e.g., refuse adding null elements
- <E> get(int index)
  - Returns element at the position index
  - Different from Python: no shorthand
    - Can't write ~~list[pos]~~
- int size()
  - Returns the number of elements in the list
- And more!
  - contains, remove, toArray, …



Generic Collection Interfaces

- Declaration of the Collection interface:

```
public interface Collection<E> …
```

Type parameter

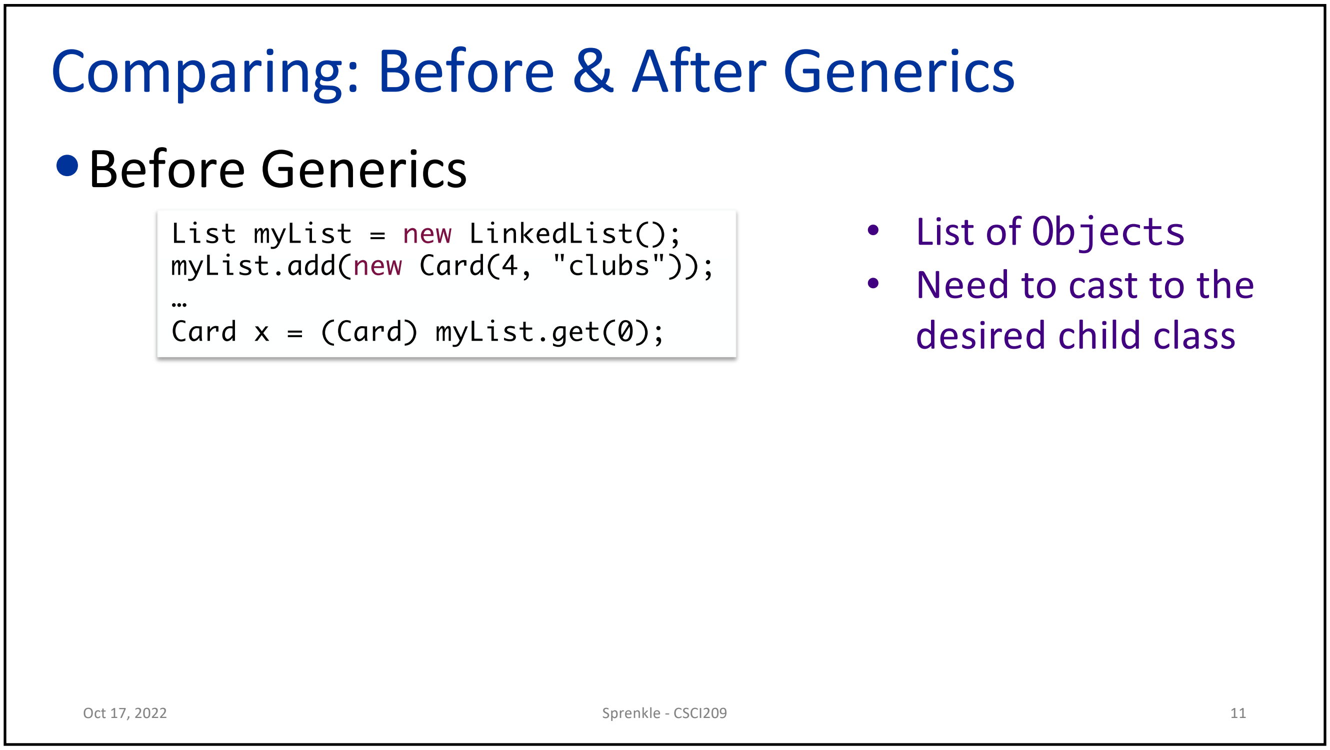
  - <E> means interface is generic for **e**lement class
- When declare a Collection, **specify type** of object it contains
  - Allows compiler to verify that object's *type* is correct
    - Reduces errors at runtime
- Example, a hand of cards:

Always declare type contained in collections

```
List<Card> hand = new ArrayList<Card>();
```

Added in Java 7:

```
List<Card> hand = new ArrayList<>();
```



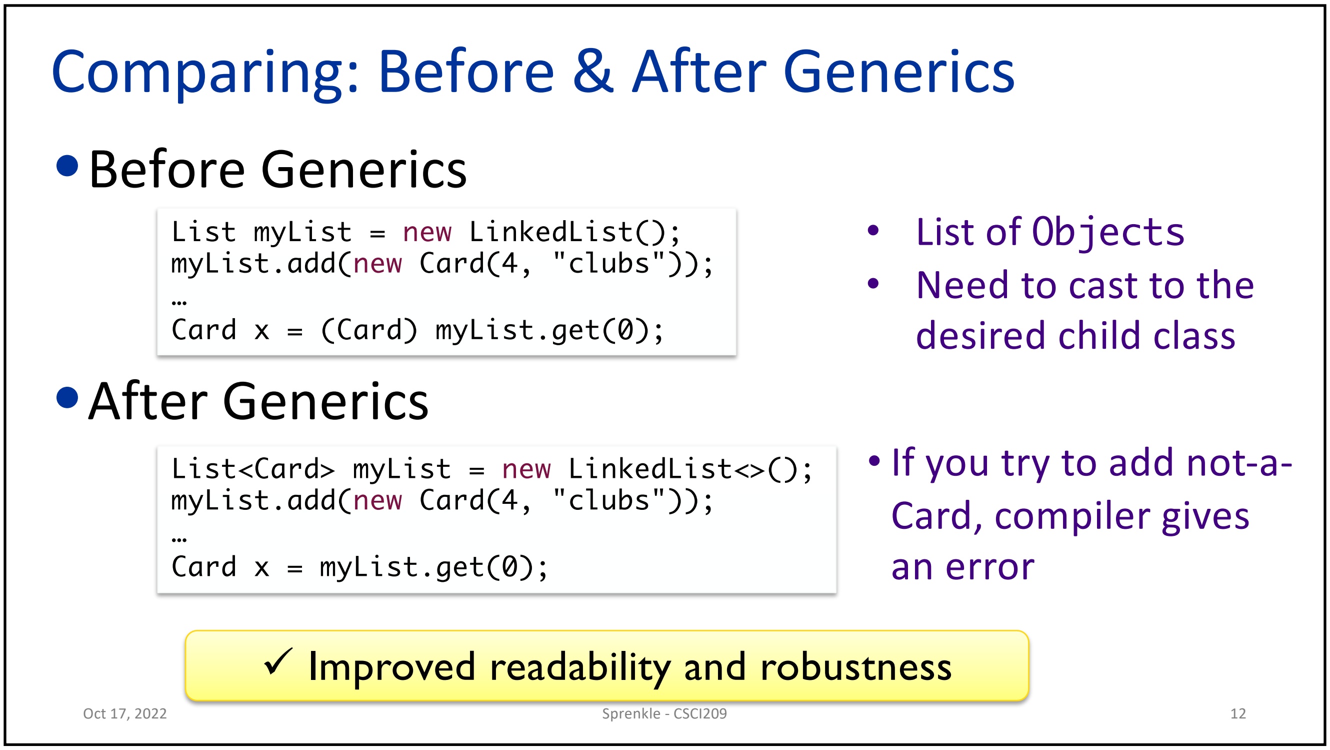
Comparing: Before & After Generics

- Before Generics

```
List myList = new LinkedList();
myList.add(new Card(4, "clubs"));
…
Card x = (Card) myList.get(0);
```

- List of Objects
- Need to cast to the desired child class

11



Comparing: Before & After Generics

- Before Generics

```
List myList = new LinkedList();
myList.add(new Card(4, "clubs"));
…
Card x = (Card) myList.get(0);
```

- List of Objects
- Need to cast to the desired child class

- After Generics

```
List<Card> myList = new LinkedList<>();
myList.add(new Card(4, "clubs"));
…
Card x = myList.get(0);
```

- If you try to add not-a-Card, compiler gives an error

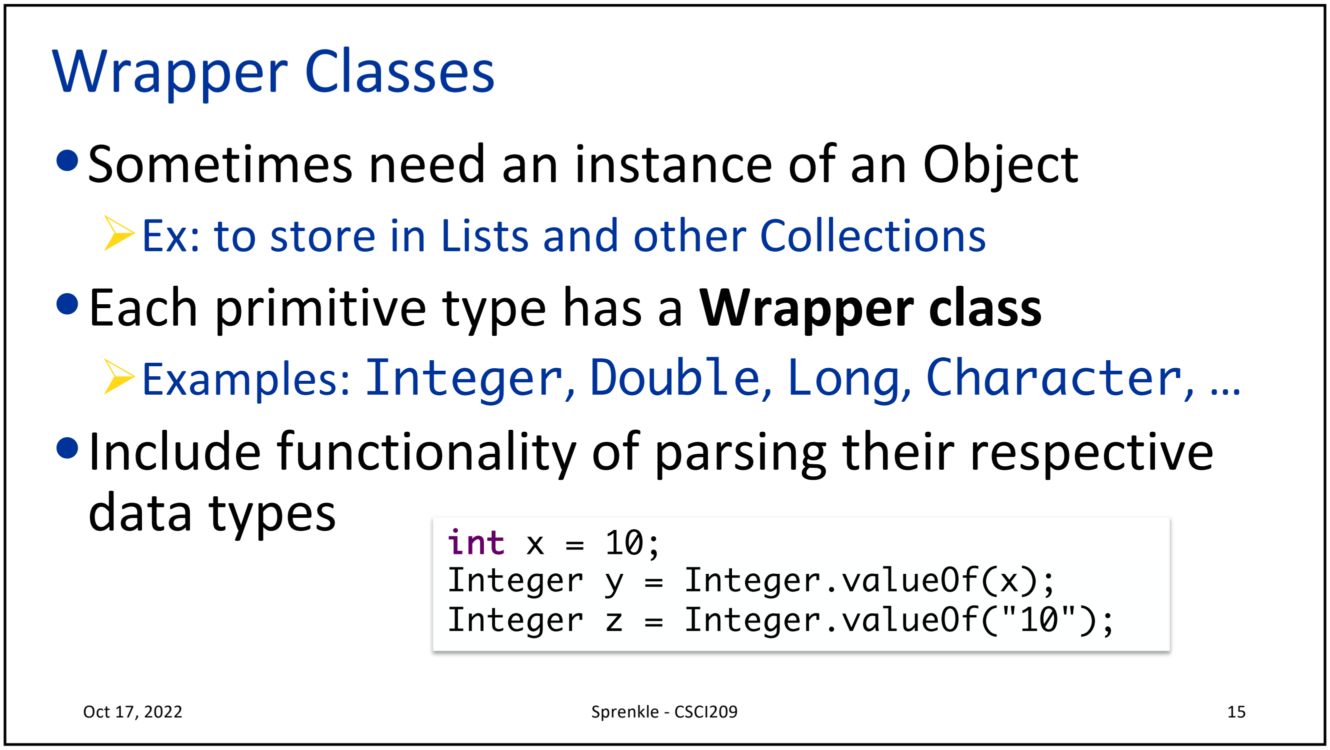
✓ Improved readability and robustness

12

Types Allowed with Generics

- Can only contain `Objects`, not primitive types
- Autoboxing and Autounboxing to the rescue!

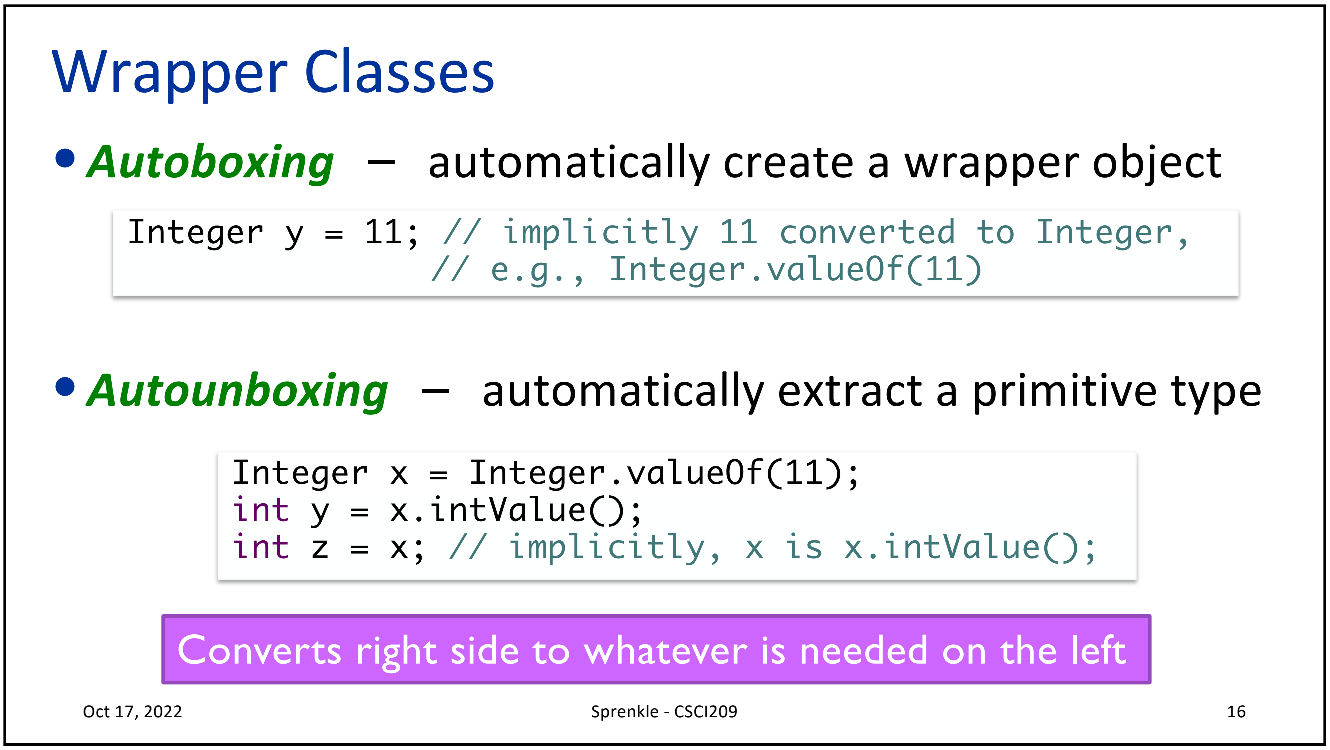
WRAPPER CLASSES



Wrapper Classes

- Sometimes need an instance of an Object
  - Ex: to store in Lists and other Collections
- Each primitive type has a **Wrapper class**
  - Examples: `Integer`, `Double`, `Long`, `Character`, …
- Include functionality of parsing their respective data types

```
int x = 10;
Integer y = Integer.valueOf(x);
Integer z = Integer.valueOf("10");
```



Wrapper Classes

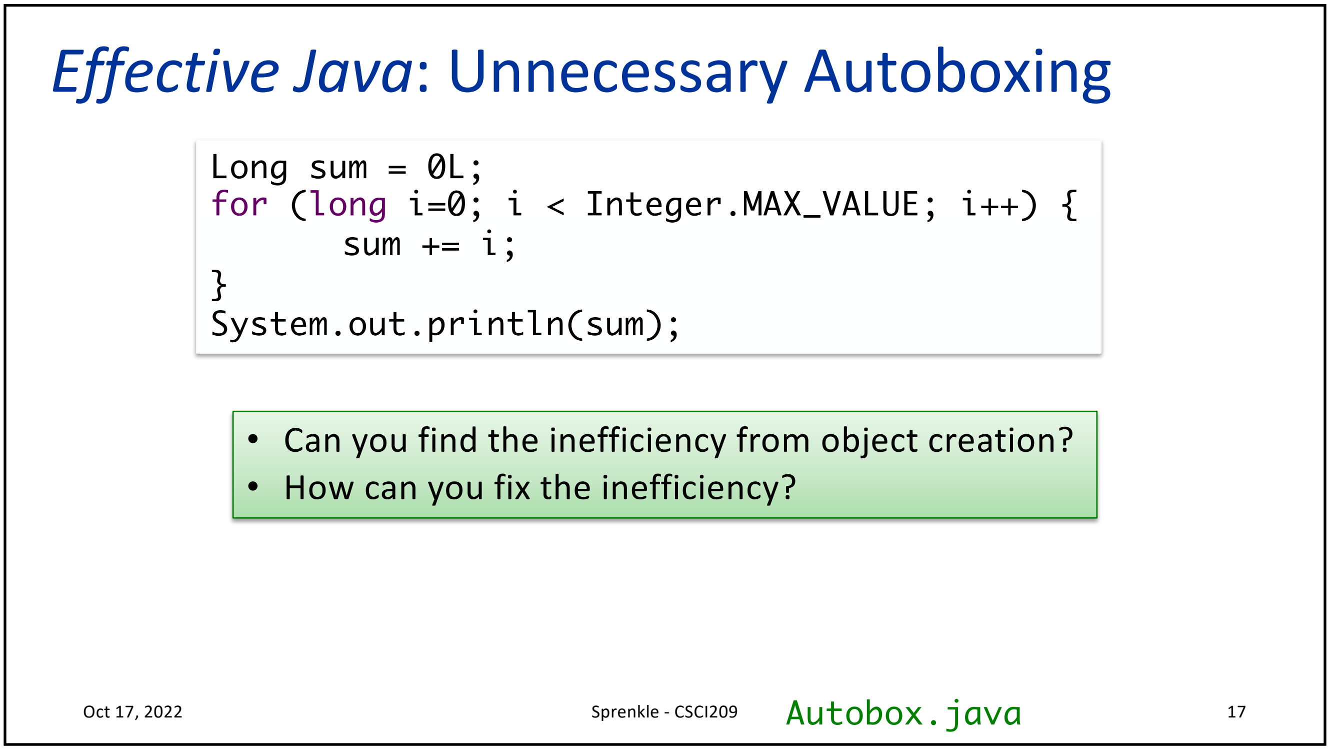
- ***Autoboxing*** – automatically create a wrapper object

```
Integer y = 11; // implicitly 11 converted to Integer,
                // e.g., Integer.valueOf(11)
```

- ***Autounboxing*** – automatically extract a primitive type

```
Integer x = Integer.valueOf(11);
int y = x.intValue();
int z = x; // implicitly, x is x.intValue();
```

Converts right side to whatever is needed on the left

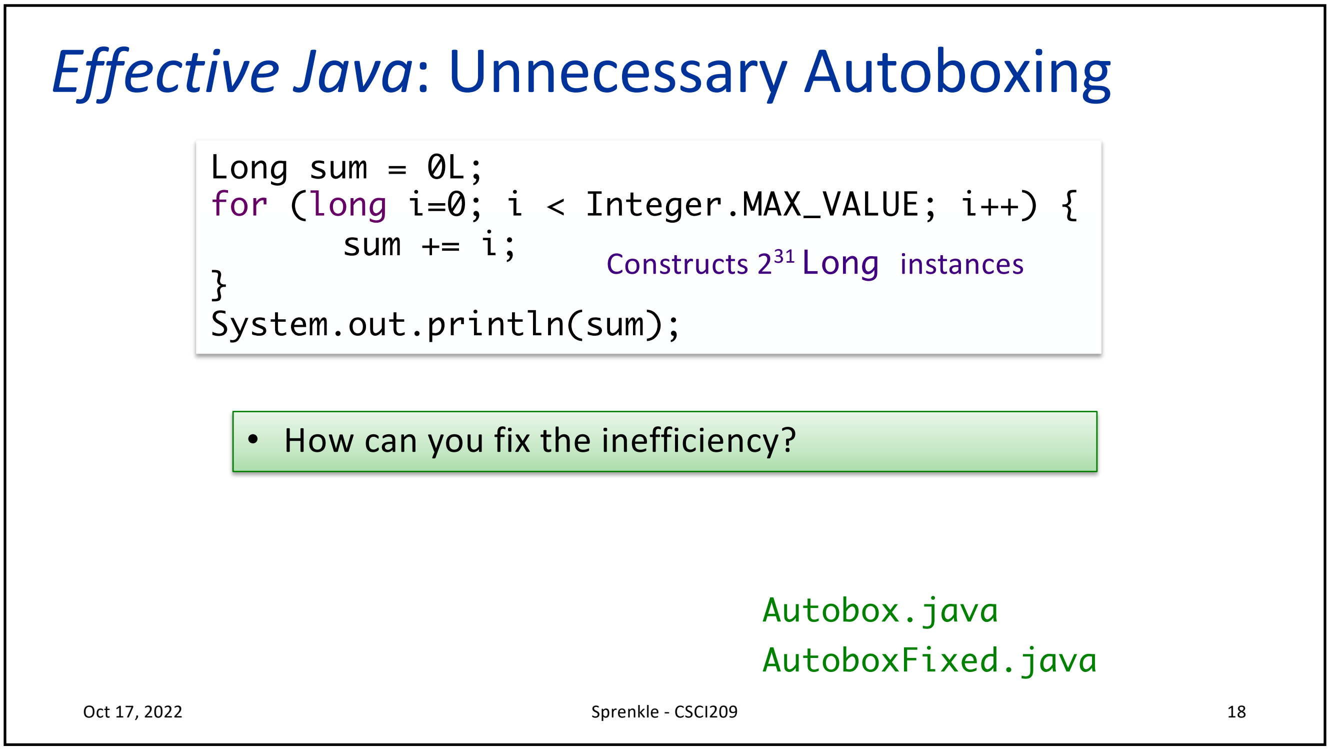


Effective Java: Unnecessary Autoboxing

```
Long sum = 0L;
for (long i=0; i < Integer.MAX_VALUE; i++) {
        sum += i;
}
System.out.println(sum);
```

- Can you find the inefficiency from object creation?
- How can you fix the inefficiency?

Autobox.java



Effective Java: Unnecessary Autoboxing

```
Long sum = 0L;
for (long i=0; i < Integer.MAX_VALUE; i++) {
        sum += i;
}
System.out.println(sum);
```

Constructs 2^{31} `Long` instances

- How can you fix the inefficiency?

Autobox.java
AutoboxFixed.java

Effective Java: Unnecessary Autoboxing

```
Long sum = 0L;
for (long i=0; i < Integer.MAX_VALUE; i++) {
        sum += i;
}
System.out.println(sum);
```

Constructs 2^{31} `Long` instances

Lessons:

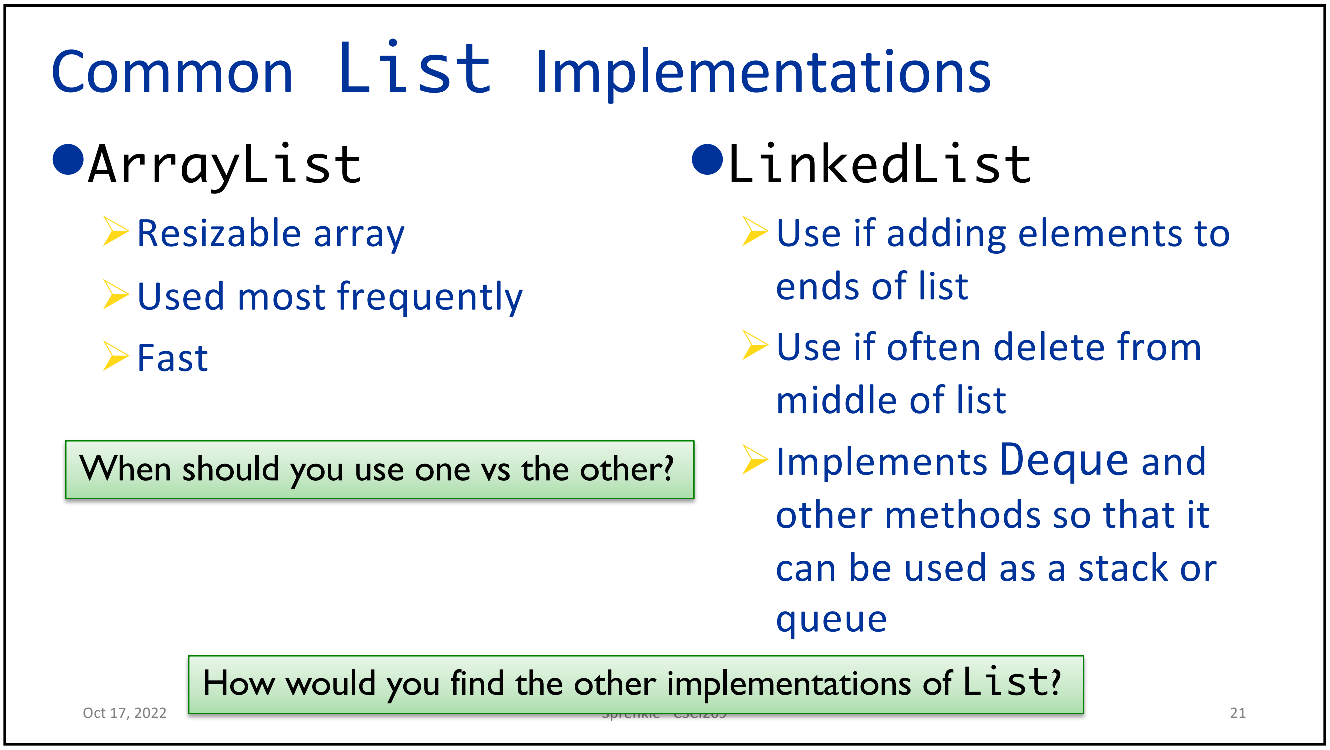
- Prefer primitives to boxed primitives
- Watch for unintentional autoboxing

Autobox.java

AutoboxFixed.java

List Interface

- boolean add(<E> o)
  - Boolean so that List can refuse some elements
    - e.g., refuse adding null elements
- <E> get(int index)
  - Returns element at the position index
  - Different from Python: no shorthand
    - Can't write ~~list[pos]~~
- int size()
  - Returns the number of elements in the list
- And more!
  - contains, remove, toArray, …

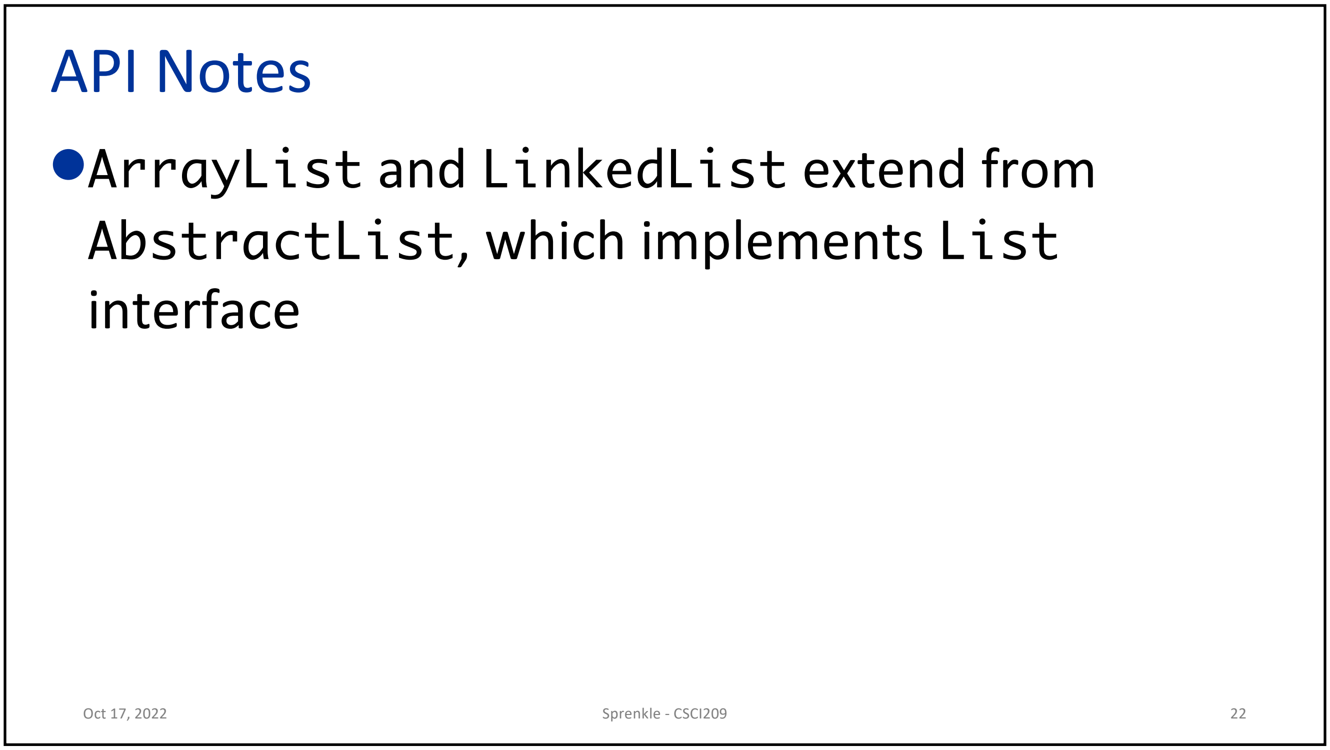


Common List Implementations

- ArrayList
  - Resizable array
  - Used most frequently
  - Fast
- LinkedList
  - Use if adding elements to ends of list
  - Use if often delete from middle of list
  - Implements Deque and other methods so that it can be used as a stack or queue

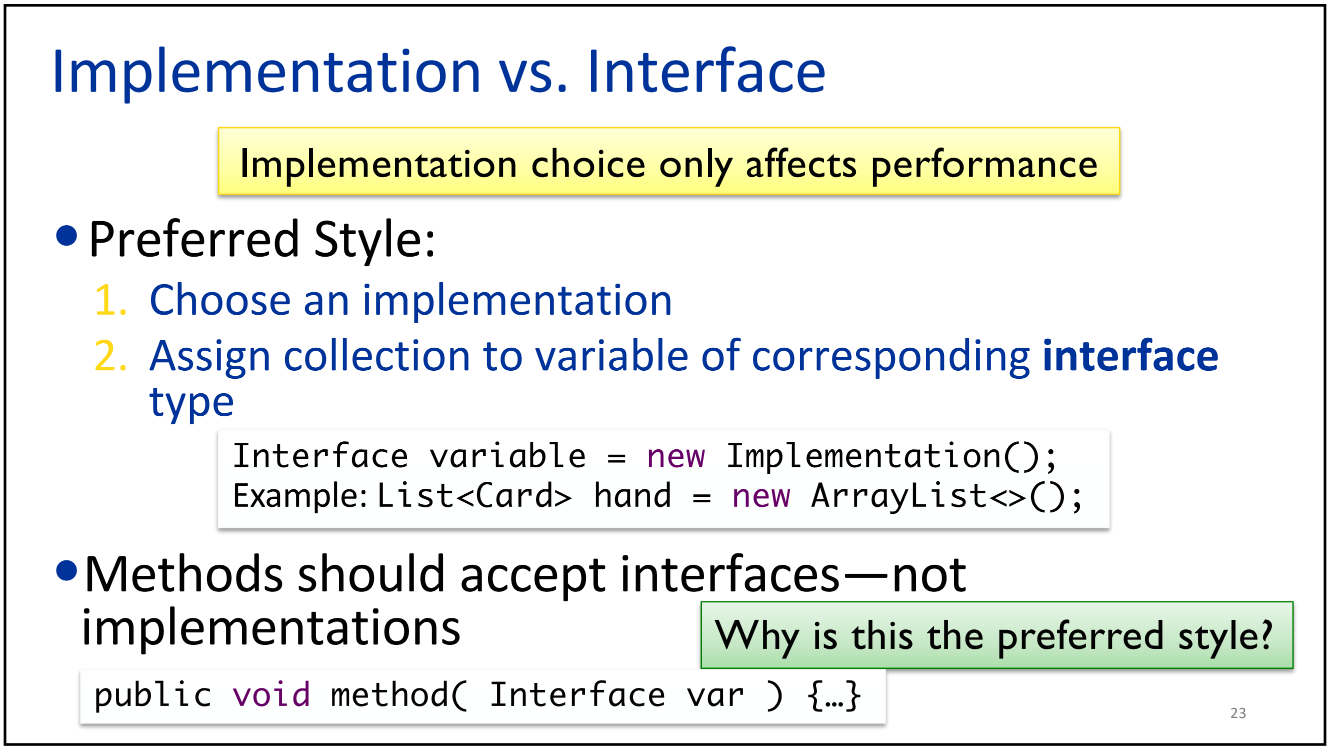
When should you use one vs the other?

How would you find the other implementations of List?



API Notes

- ArrayList and LinkedList extend from AbstractList, which implements List interface



Implementation vs. Interface

Implementation choice only affects performance

- Preferred Style:
  1. Choose an implementation
  2. Assign collection to variable of corresponding **interface** type

```
Interface variable = new Implementation();
Example: List<Card> hand = new ArrayList<>();
```

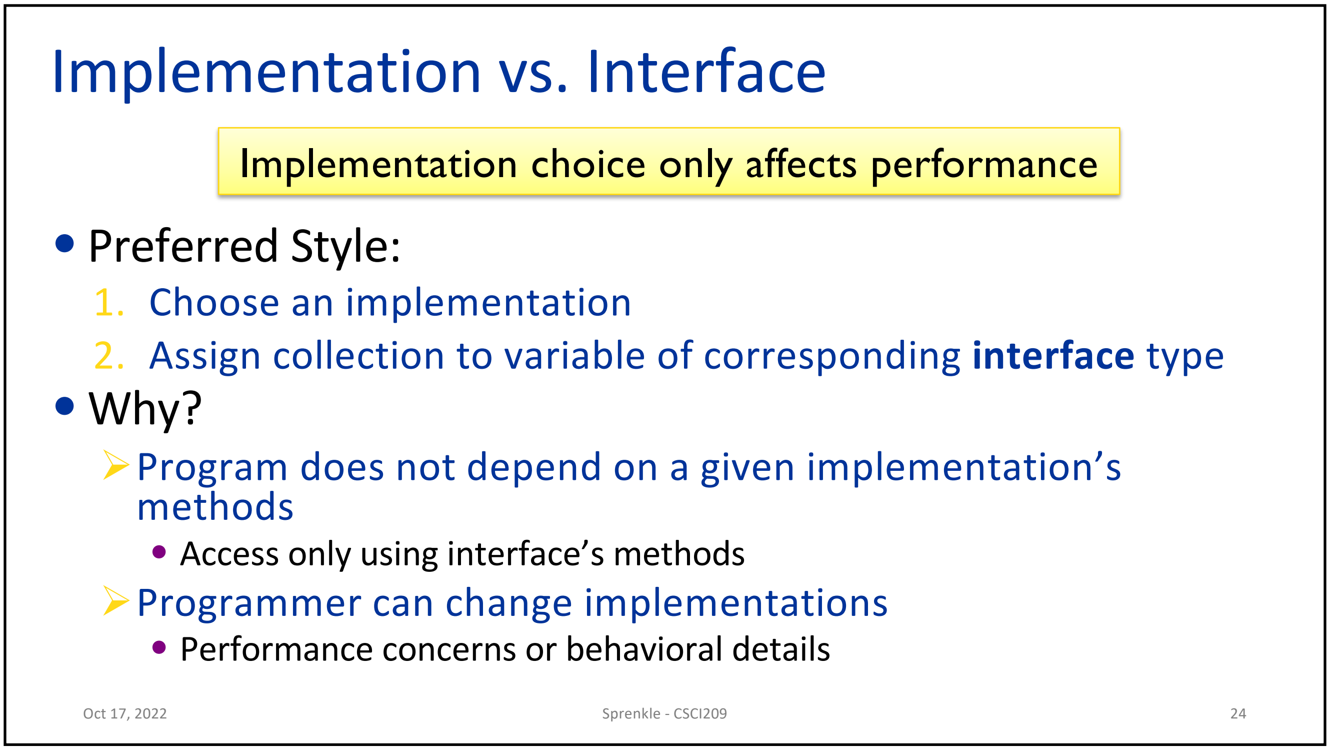
- Methods should accept interfaces—not implementations

Why is this the preferred style?

```
public void method( Interface var ) {…}
```

23

23



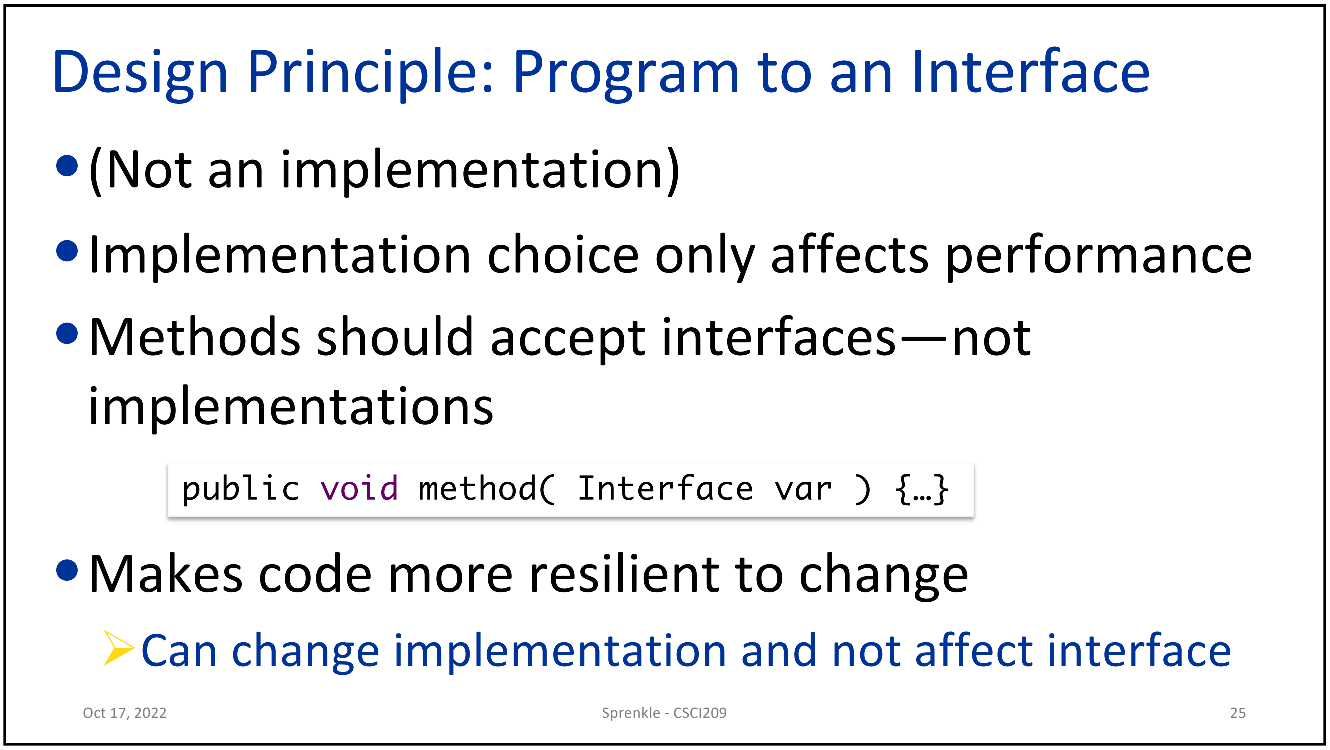
Implementation vs. Interface

Implementation choice only affects performance

- Preferred Style:
  1. Choose an implementation
  2. Assign collection to variable of corresponding **interface** type
- Why?
  - Program does not depend on a given implementation's methods
    - Access only using interface's methods
  - Programmer can change implementations
    - Performance concerns or behavioral details

Oct 17, 2022 Sprenkle - CSCI209 24

24

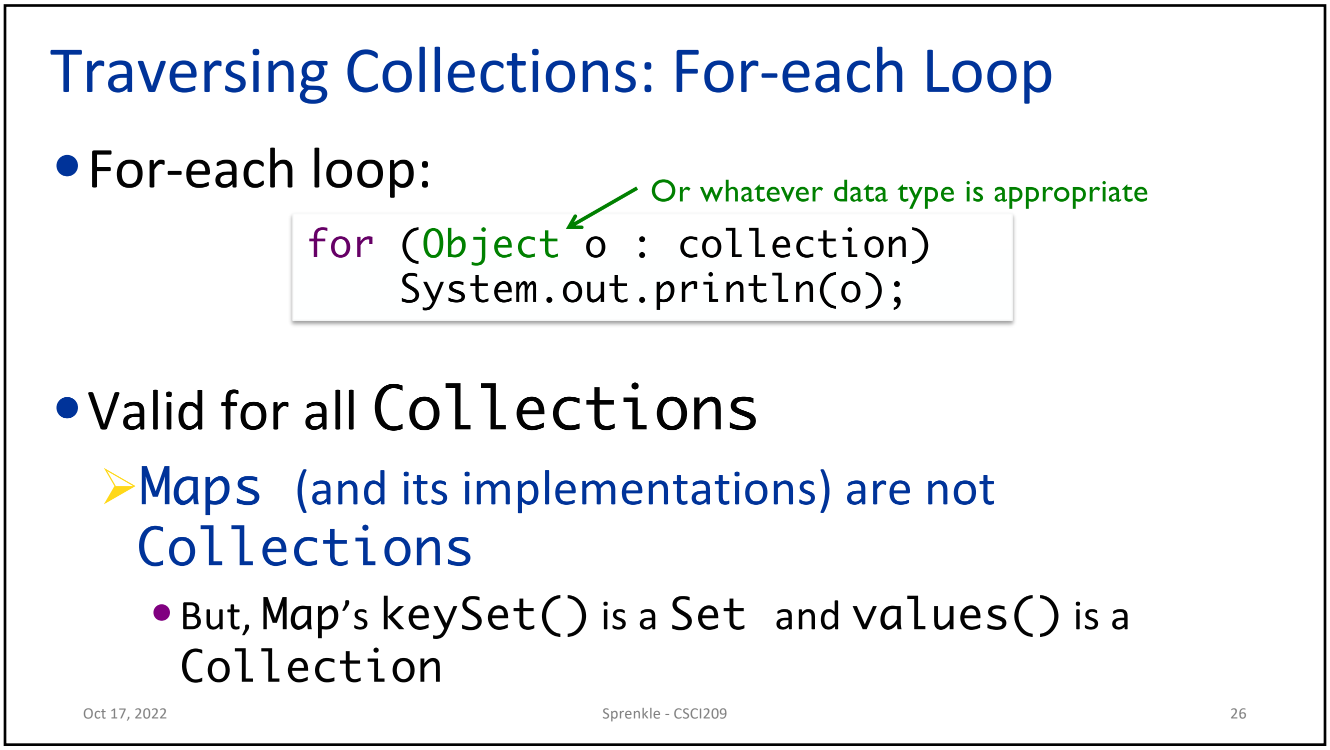


Design Principle: Program to an Interface

- (Not an implementation)
- Implementation choice only affects performance
- Methods should accept interfaces—not implementations

```
public void method( Interface var ) {…}
```

- Makes code more resilient to change
  - Can change implementation and not affect interface



Traversing Collections: For-each Loop

- For-each loop:

Or whatever data type is appropriate

```
for (Object o : collection)
    System.out.println(o);
```

- Valid for all `Collections`
  - `Maps` (and its implementations) are not `Collections`
    - But, `Map`'s `keySet()` is a `Set` and `values()` is a `Collection`

Discussion of Deck Class

`cards.Deck.java`

27

SETS

28

Set Interface

- No duplicate elements
  - Needs to determine if two elements are "logically" the same (`equals` method)
- Models mathematical set abstraction

Set Interface

- `boolean add(<E> o)`
  - Add to set, only if not already present
- `int size()`
  - Returns the number of elements in the list
- And more! (`contains`, `remove`, `toArray`, …)
  - Note: no `get` method – can't `get` #3 from the set because sets aren't ordered.

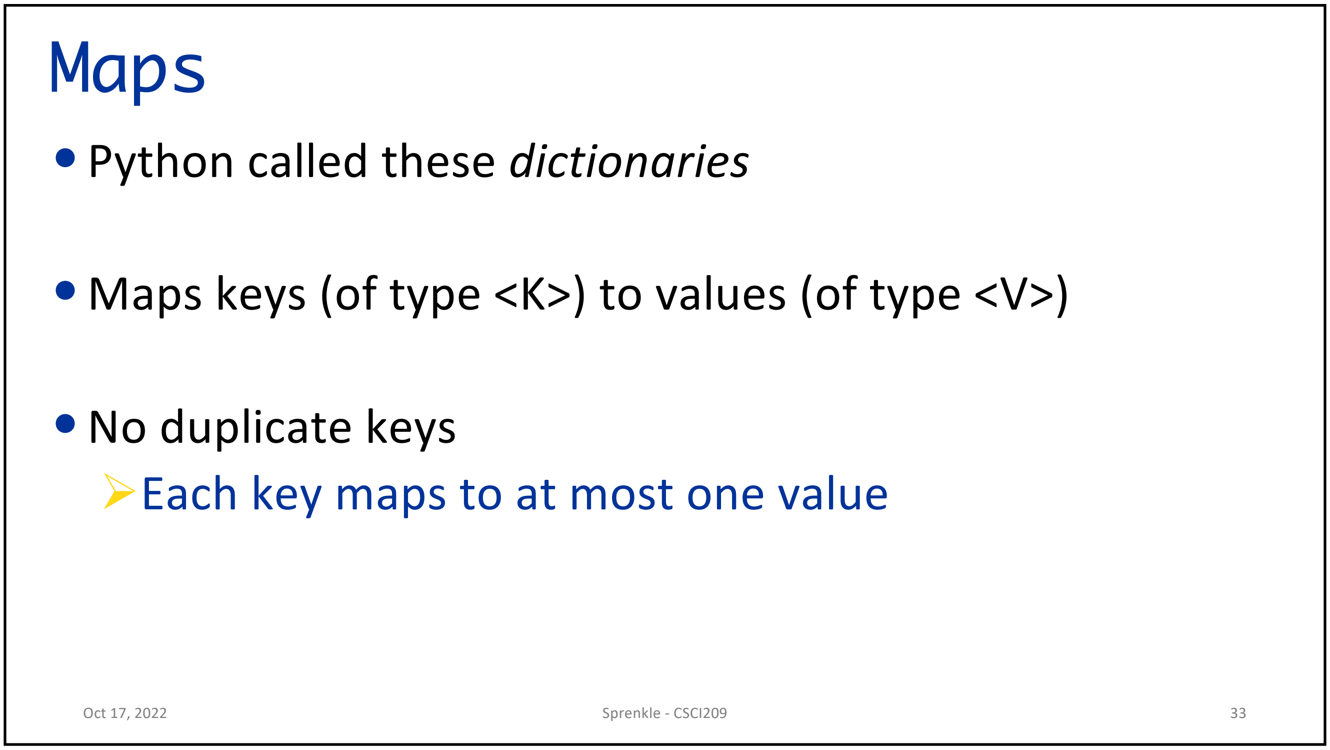
Some Set Implementations

- HashSet
  - Implements set using *hash table*
    - add, remove, and contains each execute in O(1) time
  - Used more frequently
  - Faster than TreeSet
  - No ordering
- TreeSet
  - Implements set using a *tree*
    - add, remove, and contains each execute in O(log n) time
  - Sorts

31

MAPS

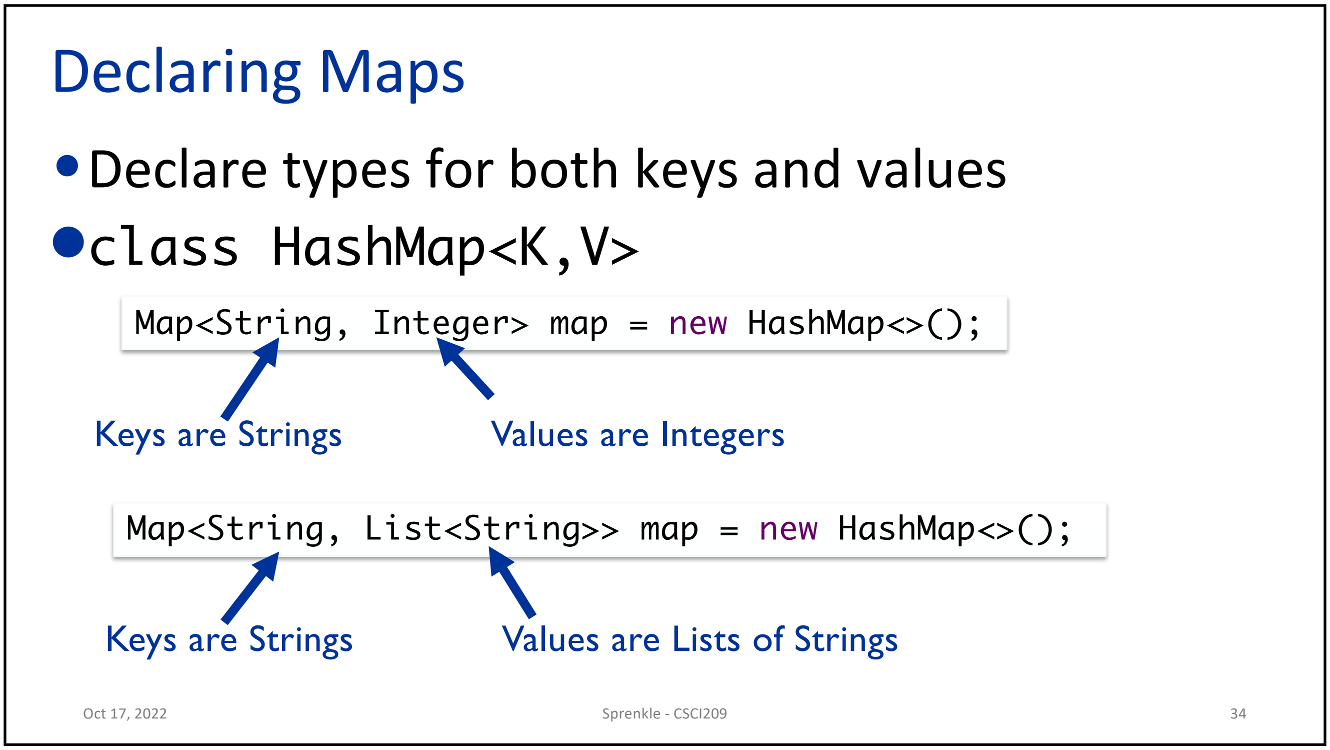
32



Maps

- Python called these *dictionaries*
- Maps keys (of type <K>) to values (of type <V>)
- No duplicate keys
  - Each key maps to at most one value

33



Declaring Maps

- Declare types for both keys and values
- `class HashMap<K,V>`

```
Map<String, Integer> map = new HashMap<>();
```

Keys are Strings

Values are Integers

```
Map<String, List<String>> map = new HashMap<>();
```

Keys are Strings

Values are Lists of Strings

34

Map Interface

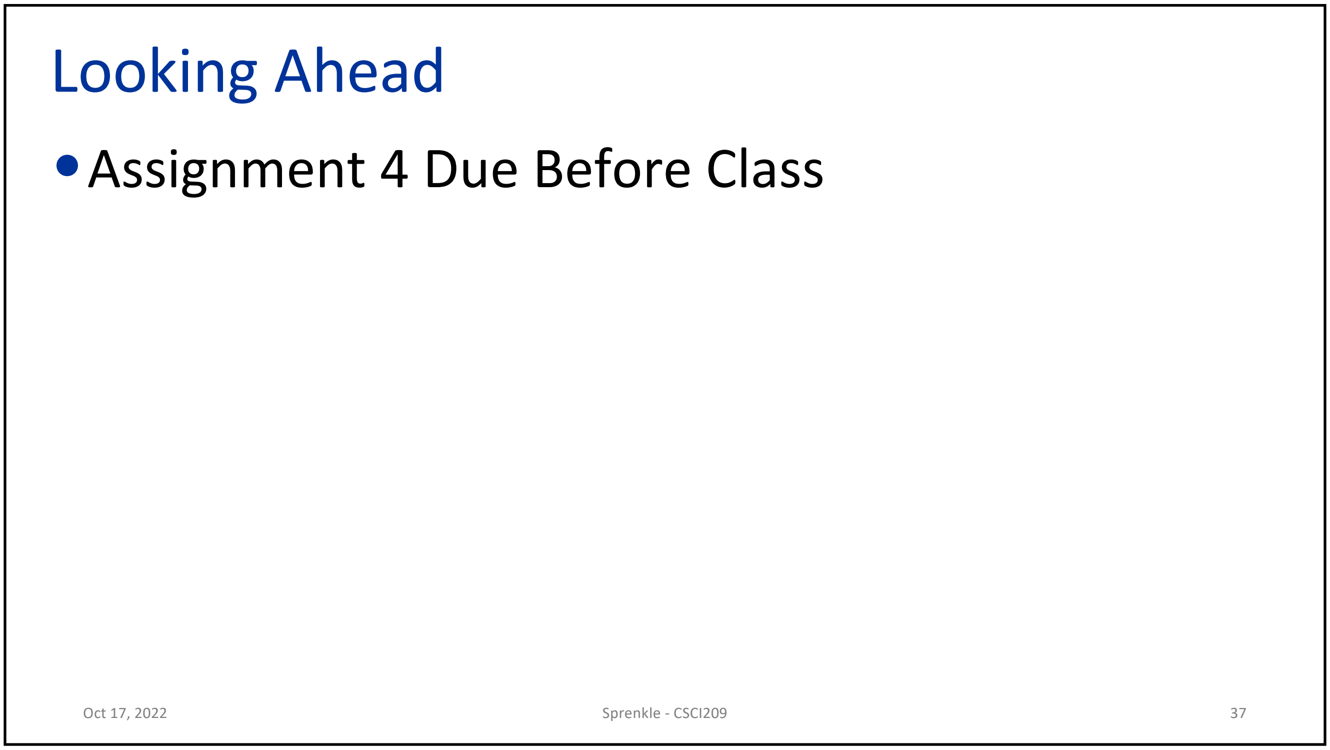
- `<V> put(<K> key, <V> value)`
  - Returns old value that key mapped to
- `<V> get(Object key)`
  - Returns value at that key (or null if no mapping)
- `Set<K> keySet()`
  - Returns the set of keys

And more …

A few Map Implementations

- `HashMap`
  - Fast
- `TreeMap`
  - Sorting
  - Key-ordered iteration
- `LinkedHashMap`
  - Fast
  - Insertion-order iteration

`MapExample.java`



Looking Ahead

- Assignment 4 Due Before Class