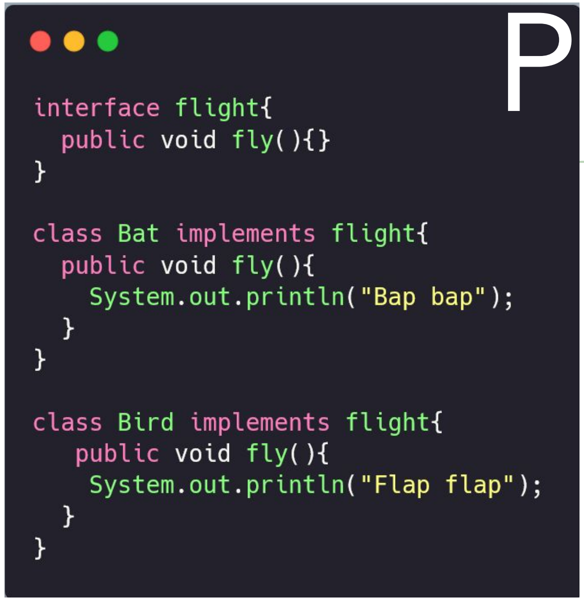
Iterators, Iterable, Polymorphism

Exam-Level 05



```
interface flight{
  public void fly(){}
}

class Bat implements flight{
  public void fly(){
    System.out.println("Bap bap");
  }
}

class Bird implements flight{
   public void fly(){
    System.out.println("Flap flap");
  }
}
```

CS 61B Fall 2023

Announcements

- Weekly Survey 4 due Monday 9/18
- Lab 5 Timing (optional) released
- Project 1B due Monday 9/18
- Midterm 1 Thursday 9/21 7-9PM
- Project 1C due Monday 9/25

Content Review

Subtype Polymorphism

Polymorphism in programming describes the ability for methods to work on a variety of types. This gives us more general code, or in other words, a single uniform interface that can work with many types.

Subtype polymorphism describes the fact that subtypes of a Class or Interface are also instances of that Class or Interface. Any method that takes in the parent type will take in an instance of the subtype, ie:

```
public ComparableArray <T implements Comparable> implements Comparable { ... }
```

Our ComparableArray is polymorphic: it works with any type that is comparable (and we bind our generic T to be Comparable items only). Now we can do:

```
    T item1 = ...;
    T item2 = ...;
    item1.compareTo(item2);
```

Example: Comparators

Comparators are an example of how subtype polymorphism is commonly used.

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

The `Comparator` interface's `compare` function takes in two objects of the same type and outputs:

- A negative integer if `o1` is "less than" `o2`
- A positive integer if `o1` is "greater than" `o2`
- Zero if `o1` is "equal to" `o2`

An object is determined to be "less than" or "greater than" or "equal to" another object based on how a class that `implements Comparator` fills in its own `compare.` This allows us to compare things that aren't inherently numerical (ie. `Dogs`)

Interfaces vs. Classes Refresher

- A class can **`implement`** many interfaces and **`extend`** only one class
- Interfaces tell us what we want to do but not how; classes tell us how we want to do it
- Interfaces can have empty method bodies (that must be filled in by subclasses) or default methods (do not need to be overridden by subclasses)
- With extends, subclasses inherit their parent's (non-private) instance and static variables, methods (can be overridden), nested classes
    - But not constructors!
    - Use **`super`** to refer to the parent class

The Iterator & Iterable Interfaces

Iterators are objects that can be iterated through in Java (in some sort of loop).

```
    public interface Iterator<T> {
        boolean hasNext();
        T next();
    }
```

Iterables are objects that can produce an iterator.

```
    public interface Iterable<T> {
        Iterator<T> iterator();
    }
```

You might have seen syntax like

```
    for (String x : lstOfStrings) // Lists, Sets, Arrays are all Iterable!
```

which is shorthand for

```
    for (Iterator<String> iter = lstOfStrings.iterator(); iter.hasNext();) {
        SomeObject x = iter.next();
    }
```

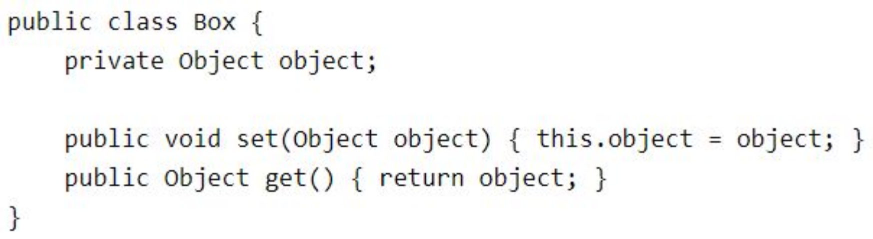
Generics and Parameterization

Generics are how we parameterize over types.

This is an example from the Oracle java documentation.

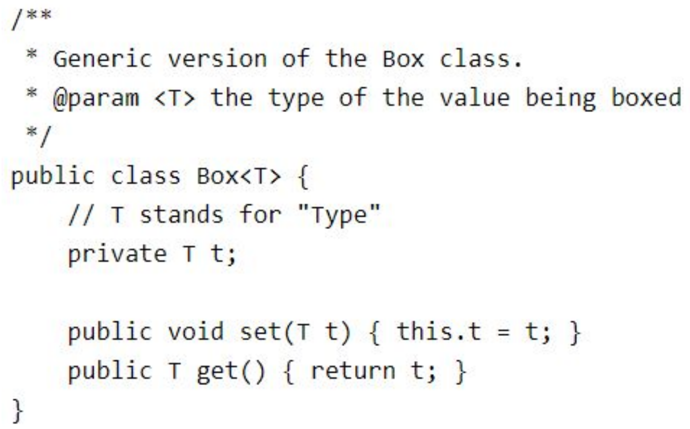
Using generics allows us to specify some degree of information beyond Object (which is not very useful for us).

Autoboxing allows us to leverage things like int to become Integers (primitive in their reference variations).



```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```



```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

An Aside: Parameterization and Casting

- Use this slide as you see fit/change it up/write your own/ignore it, but the tldr is that:
    - A lot of students don't really understand parameterization (the <>)
    - I got a really really good question about the purpose of casting because for most of the discussion 04 problems, casting didn't make a difference in what method ultimately got run (it's possible, but often comes from the gnarlier DMS cases we don't really teach anymore)
- This student's post on Ed sums things up pretty nicely!
    - Basically, casting is nice when we have generic/general types or contexts but we want more specialized behavior in certain contexts where we know for sure/are guaranteed that treating the object like the cast type will work
    - ie. equals(Object o1, Object o2) in proj 1c!
- In this worksheet, we don't really force students' hands to parameterize their parent classes in the implements lines, but it's necessary to avoid nastiness (aka casting), because otherwise the method signatures that must be overridden have argument/return types that are super generic, like Object
- This might be better demonstrated with an actual example, possibly in IntelliJ, so students can see this

Check for Understanding

1. If we were to define a class that implements the interface `Iterable<Dog>`, what method(s) would this class need to define?

2. If we were to define a class that implements the interface `Iterator<Integer>`, what method(s) would this class need to define?

3. What's one difference between `Iterator` and `Iterable`?

Check for Understanding

1. If we were to define a class that implements the interface `Iterable<Dog>`, what method(s) would this class need to define?

```
public Iterator<Dog> iterator()
```

2. If we were to define a class that implements the interface `Iterator<Integer>`, what method(s) would this class need to define?

```
public boolean hasNext()
public Integer next()
```

3. What's one difference between `Iterator` and `Iterable`?

`Iterators` are the actual object we can iterate over (ie. think a Python generator over a list).
`Iterables` are object that can produce an iterator that somehow iterate over its contents, usually some kind of collection (ie. an array is iterable; an iterator over the array could go through the element at every index of the array).

`==` vs. `.equals()`

- `==` compares the literal bits at the location of the variable; typically only used for primitive types
    - it will compare the memory addresses of two reference types, which can be useful when trying to determine if two variables point to the same object in memory
    - the exception to this is `null` - special pointer that is compared with `==`
- Alternative for reference types: .equals() (ex. `myDog.equals(yourDog)`)
    - This can be overridden on a class-by-class basis, but defaults to `Object’s .equals()` (which just compares memory addresses, like ==!)
    - ie. say we make the `Dog .equals()` method return true if both `Dogs` have the same name
        - `Dog fido = new Dog(“Fido”); Dog otherFido = new Dog(“Fido”);`
        - `fido == otherFido -> false, but fido.equals(otherFido) -> true`

Worksheet

1 Take Us to Your "Yrnqre"

Given the `AlienAlphabet` class, fill in `AlienComparator` class so that it compares strings lexicographically, based on the order passed into the `AlienAlphabet` constructor. For simplicity, you may assume all words passed into `AlienComparator` have letters present in `order`.

For example, if the alien alphabet has the order `"dba..."`, which means that `'d'` is the first letter, `'b'` is the second letter, and so on. `AlienAlphabet.AlienComparator.compare("dab", "bad")` should return a value less than `0`, since "`dab`" comes before "`bad`".

If one word is an exact prefix of another, the longer word comes later. For example, `"bad"` comes before `"badly"`.

```java
public class AlienAlphabet {
    private String order;

    public AlienAlphabet(String o) {
        order = o;
    }
}
```

1 Take Us to Your "Yrnqre"

```
public class AlienComparator implements Comparator<____________> {
    public int compare(String word1, String word2) {
        int minLength = Math.min(__________________, __________________);
        for (__________________________________________________________________) {
            int char1Rank = _______________________________________________;
            int char2Rank = _______________________________________________;
            if (_______________________________________________) {
                return -1;
             else if (_______________________________________________) {
                return 1;
             }
        }
        return _______________________________ -
________________________________;
    }
}
```

1 Take Us to Your "Yrnqre"

```
public class AlienComparator implements Comparator<String> {
    public int compare(String word1, String word2) {
        int minLength = Math.min(word1.length(), word2.length());
        for (int i = 0; i < minLength; i++) {
            int char1Rank = order.indexOf(word1.charAt(i));
            int char2Rank = order.indexOf(word2.charAt(i));
            if (char1Rank < char2Rank) {
                return -1;
            } else if (char1Rank > char2Rank) {
                return 1;
            }
        }
        return word1.length() - word2.length();
    }
}
```

2 Iterator of Iterators

```
private class IteratorOfIterators  _____________ {
    public IteratorOfIterators(List<Iterator<Integer>> a) {

    }

    public boolean hasNext() {



    }

    public Integer next() {



    }

}
```

2 Iterator of Iterators - Solution 1

```java
private class IteratorOfIterators implements Iterator<Integer> {
    LinkedList<Iterator<Integer>> iterators;

    public IteratorOfIterators(List<Iterator<Integer>> a) {
        iterators = new LinkedList<>();
        for (Iterator<Integer> iterator : a) {
            if (iterator.hasNext()) {
                iterators.add(iterator);
            }
        }
    }

    ...
}
```

2 Iterator of Iterators - Solution 1

```
private class IteratorOfIterators implements Iterator<Integer> {
    LinkedList<Iterator<Integer>> iterators;
    public IteratorOfIterators(List<Iterator<Integer>> a) { ... }

    public boolean hasNext() {
        return !l.isEmpty();
    }

    public Integer next() {

    }
}
```

2 Iterator of Iterators - Solution 1

```
private class IteratorOfIterators implements Iterator<Integer> {
    LinkedList<Iterator<Integer>> iterators;
    public IteratorOfIterators(List<Iterator<Integer>> a) { ... }

    public boolean hasNext() {
        return !iterators.isEmpty();
    }

    public Integer next() {
        Iterator<Integer> nextIter = iterators.removeFirst();
        Integer nextItem = nextIter.next();
        if (nextIter.hasNext()) {
            l.addLast(nextIter);
        }
        return nextItem;
    }
}
```

2 Iterator of Iterators - Alternate

```
private class IteratorOfIterators implements Iterator<Integer> {
    LinkedList<Integer> l;

    public IteratorOfIterators(List<Iterator<Integer>> a) {
        l = new LinkedList<>();
        while (!a.isEmpty()) {
            Iterator<Integer> curr = a.remove(0);
            if (curr.hasNext()) {
                l.add(curr.next());
                a.add(curr);
            }
        }
    }

    ...
}
```

2 Iterator of Iterators

```
private class IteratorOfIterators implements Iterator<Integer> {
    LinkedList<Integer> l;
    public IteratorOfIterators(List<Iterator<Integer>> a) { ... }

    public boolean hasNext() {
        return !l.isEmpty();
    }

    public Integer next() {

    }
}
```

2 Iterator of Iterators

```
private class IteratorOfIterators implements Iterator<Integer> {
    LinkedList<Integer> l;
    public IteratorOfIterators(List<Iterator<Integer>> a) { ... }

    public boolean hasNext() {
        return !l.isEmpty();
    }

    public Integer next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        return l.removeFirst();
    }
}
```

CS 61B
Fall 2023

Comparators, Iterators

Exam-Level 5: September 18, 2023

1 Take Us to Your "Yrnqre"

You're a traveler who just landed on another planet. Luckily, the aliens there use the same alphabet as the English language, but in a different order.

Given the `AlienAlphabet` class below, fill in `AlienComparator` class so that it compares strings lexicographically, based on the order passed into the `AlienAlphabet` constructor. For simplicity, you may assume all words passed into `AlienComparator` have letters present in `order`.

For example, if the alien alphabet has the order `"dba..."`, which means that `d` is the first letter, `b` is the second letter, etc., then `AlienComparator.compare("dab", "bad")` should return a negative value, since `dab` comes before `bad`.

If one word is an exact prefix of another, the longer word comes later. For example, `"bad"` comes before `"badly"`. *Hint:* `indexOf` *might be helpful.*

Key Insight 1:
Use the order instance variable and indexOf to get values of characters.

Order of Completion: (in order)
6
9 11 13 15
17 20 25

```
1   public class AlienAlphabet {
2       private String order;
3       public AlienAlphabet(String alphabetOrder) {
4           order = alphabetOrder;
5       }
6       public class AlienComparator implements Comparator<String> {
7           public int compare(String word1, String word2) {
8
9               int minLength = Math.min(word1.length(), word2.length());
10
11              for (index = 0 ; index < minLength; index++) {
12
13                  int char1Rank = order.indexOf( word1.charAt(index));
14
15                  int char2Rank = order.indexOf(word2.charAt(index));
16
17                  if (char1Rank < char2Rank) {
18                      return -1;
19
20                  } else if (char1Rank > char2Rank) {
21                      return 1;
22                  }
23              }
24
25              return word1.length() - word2.length();
26          }
27      }
28  }
```

2 Iterator of Iterators

Implement an `IteratorOfIterators` which takes in a `List` of `Iterators` of `Integers` as an argument . The first call to `next()` should return the first item from the first iterator in the list. The second call should return the first item from the second iterator in the list. If the list contained `n` iterators, the `n+1`th time that we call `next()`, we would return the second item of the first iterator in the list.

Note that if an iterator is empty in this process, we continue to the next iterator. Then, once all the iterators are empty, `hasNext` should return **`false`**. For example, if we had 3 `Iterators` A, B, and C such that A contained the values `[1, 3, 4, 5]`, B was empty, and C contained the values `[2]`, calls to `next()` for our `IteratorOfIterators` would return `[1, 2, 3, 4, 5]`.

There are many approaches to this problem.
- keep an index to track the next iterator to use, but potentially tricky to skip over empty iterators correctly
- This approach (adding/removing iterators)

Order of Completion: In Order (N/A)

```java
import java.util.*;
public class IteratorOfIterators implements Iterator<Integer> {
    LinkedList<Iterator<Integer>> iterators = new LinkedList<>();

    public IteratorOfIterators(List<Iterator<Integer>> a) {
        for (Iterator<Integer> it : a) {
            if (it.hasNext()) {
                a.addLast(it);
            }
        }
    }

    @Override
    public boolean hasNext() {
        return !iterators.isEmpty();
    }

    @Override
    public Integer next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        Iterator<Integer> front = iterators.removeFirst();
        int returnValue = front.next();
        if (front.hasNext()) {
            iterators.addLast(front);
        }
        return returnValue;
    }
}
```

Technically undefined behavior but commonly used

could also use a deque