Java Generics

Warmup Question

```
class Base { void baseMethod(){} }
class Derived extends Base {
    @Override void baseMethod() {}
    void derivedMethod()
}
```

What happens in these cases (which method versions are called)?:

1. `Derived x1 = new Derived(); x1.baseMethod();`
2. `Base x2 = new Derived(); x2.baseMethod();`
3. `Base x3 = new Derived(); x3.derivedMethod();`
4. `Derived x4 = new Base(); x4.derivedMethod();`

Reverse method

```
public static void reverse(int [] arr){
    for(int i = 0; i < arr.length/2; i++){
        var tmp = ar[i];
        arr[i] = arr[arr.length - 1 - i];
        arr[arr.length - 1 - i] = tmp;
    }
}
```

What would I have to change to make this work with doubles/Points/Students ?

Growable Array

```
public class GrowableArray{
    private int[] arr = new int[10]; private int size = 0;
    public void add(int x){
        if(size ≥ arrlength){ throw ... }
        arr[size] = x;
        size++;
    }
    public int get(int index){
        if(index ≥ size){ throw ... }
        return arr[index];
    }
}
```

What would I have to change to make this work with doubles/Points/Students?

Java Generics

- Interfaces and inheritance let us write code that will be able to use objects of types that haven't been written yet
- However, the authors of those classes must be aware of our code becuase they will need to either implement interfaces we've defined, or inherit from classes we've written
- Java Generics let us write code that uses “placeholder” types which we can fill in with any object types, which don't need to implement any particular interface/inherit from a particular class
- The syntax is similar to C++ templates but the mechanics are quite different!

Generic Reverse

```
public static < T > void reverse(T[] arr){
    for(int i = 0; i < arr.length/2; i++){
        T tmp = arr[i];
        arr[i] = arr[arr.length - 1 - i];
        arr[arr.length - 1 - i] = tmp;
    }
}
```

- T is a “Generic Type” or “Type Parameter”
- It is a “Placeholder” type that is filled in when we call the method and pass an array
- T can be filled be any object type

Generic Methods

- Generic methods are specified with a “Type Parameter List” in angle brackets before the return type
- These are almost always used as parameter types
- When we call the method, the compiler “infers” the actual types based on the arguments passed
- If we call `reverse(arrayOfStrings)` our type parameter `T` would be inferred as `String`
- We can use `T` inside or our method, and for this particular method call, `T` will be a synonym for `String` (sort of)

Generic GrowableArray

```
public class GrowableArray< T > {
    T[] arr = ... //sort of
    int size = 0;
    void add(T obj){ arr[size] = obj; size++; }
    public T get(int index){ return arr[index]; }
}
```

- T is a placeholder type which can be any object type
- To create Growable Array, I have to specify what T is: `var arr = new GrowableArray<string>()`
- For that object, everywhere I use T inside the class, T would be String

Generic Classes

- Generic classes have a “Type Parameter List” after the name of the class
- Unlike generic methods, you DO need to fill them in when creating an object of generic type
- If you forget, the type parameters are basically `Object` which is almost never what you want
- You'll get compiler warnings about “raw types” if you forget
- We can also have generic interfaces, which work just like classes, except we won't have any method implementations

Comparison with just using `Object`

- Java Generics work basically like using `Object` except that the compiler will do some extra checking and automatic casting for us
- The `GrowableArray` shows us the advantage of this. Using generics, our `get(index)` method can return a `T`
- If we just stored `Object[]` we'd have to return `Object` and everytime someone called get they'd have to cast it to use any methods besides `toString()` or `equals()`

The problem with `equals(Object o)`

- When Java was first created, it didn't have generics, so the `equals` method in `Object` takes an `Object` to compare to
- In class you write, you CAN'T change the signature when you override `equals` so you must also take an `Object` parameter
- This requires you to use an `instanceof` and a cast to get a reference to your class type
- When it came time to add a similar method in an interface, Java designers avoided that issue

Comparable<t>

- Here's an interface that we'll use a LOT in this class:

```
public interface Comparable < T > {
    int compareTo( T o);
}
```

- The method is like “less than” and returns
  - a negative value if `this` is “less than” `o`
  - 0 if `this` is “equal to” `o`
  - a positive value if `this` is “greater than” `o`
- Compare this to `Object.equals(Object o)`... why is this better?

Example: Name

```
public class Name implements Comparable< Name >{
    string first, last;
    @Override public int compareTo(Name other){
        var lastCompare = last.compareTo(other.last);
        if(lastCompare =/= 0){ return lastCompare; }
        else { return first.compareTo(other.first); }
    }
}
```

- `Name` objects will be compared by last name, and then first name if necessary
- Note that we implement `Comparable < Name >` because we can compare `Name` objects with other `Name`s
- Note this is the Java equivalent of overloading `operator <` in C++

`Comparable` issues

- What if usually we want to compare by last name, then first name, but for some operations we'd like to compare first names only?
- Since we can only have one version of `compareTo` in our class, we can't implement 2 different compare functions via the `Comparable` interface
- What if we didn't write the `Name` class and so can't make it implement `Comparable` at all?
- The `Comparator` class addresses these issues

Comparator< T >

```
public interface Comparator < T > { //Java stdlib interface
    int compare(T o1, T o2);
}
public FirstNameComparator implements Comparator < Name > { // class we're writing
    public int compare(Name n1, Name n2){
        return n1.getFirst().compareTo(n2.getFirst());
    }
}
//somewhere else:
ArrayList<name> names = ...;
Collections.sort(names, new FirstNameComparator());
```

Comparator

- A class that implements `Comparator` is called a “function object” because they typically exist only to call the `compare` method
- In other languages, we might actually just pass a function to the `sort` method, but Java only allows to pass objects
- On the previous slide, we saw the “Long Form” way of defining a comparator: writing a class that implements `Comparator` and then creating an object with `new`
- There are some shorter ways to do it that are often more convenient

Shorter: Anonymous Inline Class

```
Collections.sort(names, new Comparator< Name > (){
    public int compare(Name n1, Name n2) { ... }
}
```

- We often only need a `Comparator` in one place, so we just define it where it's needed
- Usually `new InterfaceType()` is something we're not allowed to do because interfaces are abstract
- Here we're defining an anonymous (no name) class and creating an object of that new type
- In addition to keeping the definition near where it's used, we also don't have to think of a name for the class

Even Shorter: Lambda function

```
Collections.sort(names, (Name n1, Name n2) -> {
    return n1.getFirst().compareTo(n2.getFirst());
});
```

- This is called a “lambda function”
- This is actually shorthand for the previous method, except that the compiler infers which Interface and which method we're implementing based on the parameter type that `sort` is expecting
- This version let's us write, pretty much only the relevant info: how do we want to compare `Name` objects

Shortest: Lambda function “expression”

```
Collections.sort(names, (n1, n2) → n1.getFirst().compareTo(n2.getFirst()) );
```

- Now the compiler infers LOTS of stuff for us:
  - Like before, the interface and method we're implementing
  - The types of the parameters to the method (`n1` and `n2` are inferred to be type `Name`
- Instead of a function body in {} we just have a single `expression` which in this case evaluates to `int`

Choosing between `Comparable` and `Comparator`

- The biggest difference between the two is that `Comparable` is implemented **in the class you want to compare** while `Comparator` is implemented outside
- If your objects have a “natural ordering” which will be unsurprising to users of the class, it should implement `Comparable`
- If not, for example the `Point` class, there's not a “natural order” so we should define a `Comparator` when we need to order `Point` objects in a particular way
- If we ever want to compare objects using something different than the natural order, we should define a `Comparator`

GrowableArray problem

```
//in GrowableArray
public void addAll(ArrayList< T > list){
    for(var item : list){
        add(item);
    }
}
// elsewhere
GrowableArray< Student > students = ...;
ArrayList< TA > tas = ...;
students.addAll(tas); //OK?
```

- This is a compiler error. Should it work?

ArrayList< Derived > !instanceof ArrayList< Base >

- Generic classes of different types do NOT have the same relationship as the type parameters
- `ArrayList< TA >` has basically no relationship to `ArrayList< Student >`!!
- On the previous slide, since `T` was `Student` we could only a parameter whose type was `ArrayList< Student >`
- However, the body of the method would work correctly if we were passed a list of `TA` objects...

Another Problem

```
public T getSmallest(Comparator< T > comp){ //in GrowableArray
    //...
    int res = comp.compare(smallest, arr[i]); //in a loop
}
//elsewhere
GrowableArray< TA > tas = //...
tas.getSmallest(new Comparator< Student >(){
    @Override int compare(Student s1, Student s2){
        return s1.getTuitionCost() - s2.getTuitionCost();
    }
});
```

- This is a compiler error... does it make sense?

Generic “Type Bounds”

- For these 2 cases, we want to accept parameters of types that have a relationship to `T`
- Type bounds let us express those relationships in code
- `public void addAll(ArrayList< ? extends T > list)`
- This says `addAll` can take an arraylist of anything of any class with `T` in it's base class hierarchy (including `T` itself)
- The ? is called a “wildcard”
- `public T getSmallest(Comparator< ? super T > comp)`
- This method takes a comparator that accepts any class that `T` inherits from (including `T`)

Type “Upper bound”

- The `< ? extends T >` in `public void addAll(ArrayList< ? extends T > list)` is an “upper bounded type parameter”
- It means that the list must contain types that are “some type of `T`"
- This allows us to accept lists of types that inherit from `T`, which makes sense, since we'll be able to store those in our array of `T`s

Type “Lower Bound”

- The `< ? super T >` in `public T getSmallest(Comparator< ? super T > comp)`

is called a “lower bounded type parameter” * It accepts a comparator for any type that T inherits from (or T itself) * Why? Any of these comparators will accept Ts as arguments to their `compare` method * In our example, the comparator that takes `Student` objects will accept `TA` objecst since `TA`s are `Student`s

Bounded Generics

- We'll be using these a lot, but will generally provide guidance or an exact signature
- They are imporant when you mix generics with inheritance

Generics Gotchas

- Because Java generics are basically just using Objects under the hood, there's a couple of unexpected annoyances
- The biggest issue is that you can't instantiate a generic type or an array of generic types
- `new T(); new T[5]; //both are compiler errors`
- In this class, we'll have to create generic arrays, and the only way to do that is to create an `Object[]` and then cast it:
- `T[] myArr = (T[])(new Object[size]);`
- This will give you a warning, and is probably the ONLY warning that you should ignore/suppress