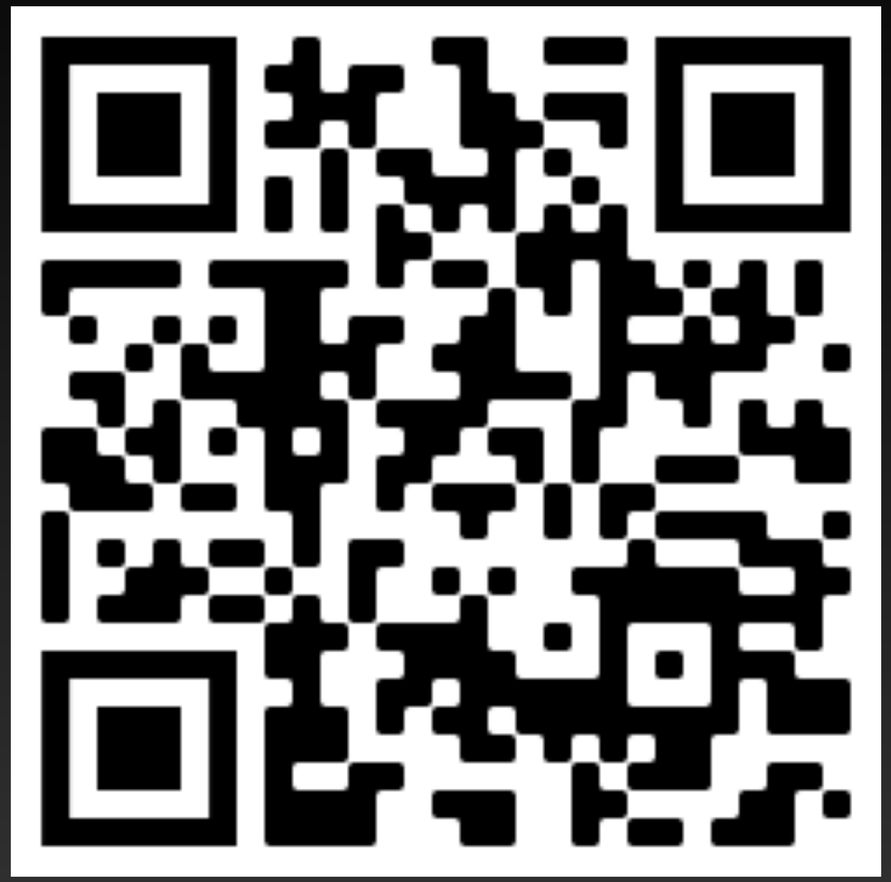
Generics

September 23, 2025
Cornell CS 2112



https://forms.gle/UifFJSxTJULPi9V7A

Generics

Types that take *another type* as a parameter

"A _____ of _____"

```
ArrayList<Point>

HashMap<String, Integer>

Matrix<Integer>
```

```
public class List<T> {
    // T can be any type
    public void add(T obj) {
        . . .
    }
}
```

A great way to reduce code duplication!

```
public class IntegerList {
    public void add(Integer obj) {
        . . .
    }
}
public class StringList {
    public void add(String obj) {
        . . .
    }
}
public class PointList {
    public void add(Point obj) {
        . . .
    }
}
```

Can group all this repeated code together, since it doesn’t depend on the type

```
public class List<T> {
    // T can be any type
    public void add(T obj) {
        . . .
    }
}
```

What if we want to access operations specific to T?

```
<T> void sort(T[] arr) {
    . . .
}
```

To sort an array of T's, we need some way of comparing them!

```
interface Comparator<T> {
    /** Returns 0 if x and y are equal,
     * a negative number if x < y,
     * and a positive number if x > y */
    int compare(T x, T y);
}

<T> void sort(T[] arr, Comparator<T> cmp) {
    . . .
    if (cmp.compare(arr[i], arr[j]) > 0) {
        . . .
    }
    . . .
}
```

Comparator Exercise!

Type Erasure

```
public class List<T> {

    T[] array;

    // T can be any type
    public void add(T obj) {
        . . .
    }
}
```

```
public class List {

    Object[] array;

    public void add(Object obj) {
        . . .
    }
}
```

The compiler does not want to create new classes, so it replaces all mentions of T with Object, adding casts if necessary

Raw Types

How Java used to work

- What if we just leave the generic parameter blank?
- This will work! But it is dangerous
- Avoid using raw types unless you absolutely must

```
HashSet names = new HashSet();
names.add(“Sylvan”);
names.add(“Zebulon”);
names.add(“Michael”);
names.add(42);
```

What happens here?

```
for (Object name : names) {
    String s = ((String) name).toLowerCase()
    System.out.println(s); // Runtime error!
}
```

Arrays of Generics

```
T[] array = new T[n];
```

```
HashSet<T>[] sets = new HashSet<T>[n];
```

You can’t do this

```
T[] array = (T[]) (new Object[n]);
```

```
HashSet<T>[] sets = new HashSet[n];
```

You have to do this

I still haven’t found a good explanation for why this is how it is,
other than “the Java designers decided to do it this way”

Collections

A bunch of helpful Java types

- When you need a list of generic types, use a collection type instead of an array!
- Java provides many classes that obey this `Collection` interface
  - `ArrayList`, `LinkedList`, `HashSet`

```
interface Collection<T> {
    boolean contains(T x);
    boolean remove(T x);
    boolean add(T x);
    int size();
}
```

Collections Exercise!

Subtyping Implications of Generics

It’s not super straightforward!

```
LinkedList<T> implements Collection<T>
```

```
LinkedList<T>  <: Collection<T>
String  <: Object
LinkedList<String>  <: LinkedList<Object>
```

Thank you!