Lecture 5

Generics and Collections

Midterm Info

- **Date**: 5 March 2018
  Monday after recess week
- **Time**: 1000am - 1130am 90 minutes
- **Venue**: MPSH M2C

Midterm Info

- **Scope**: Lecture 1-6
- **Format:** MCQ + short questions
- **Open Book**

Previously, in cs2030..

Variance of Types

- Subtype relationship between complex types
- T <: S implies T[] <: S[]

(concrete)
class

interface

(concrete) class

(abstract) class

interface

```
abstract class PaintedShape {
  Color fillColor;
  void fillWith(Color c) {
    fillColor = c;
  }
    :
  abstract double getArea();
  abstract double getPerimeter();
    :
}
```

(concrete) class	(abstract) class	interface with default methods	(pure) interface

```java
interface Shape {
  double getArea();
  double getPerimeter();
  boolean contains(Point p);
}

class Circle implements Shape { .. }
class Rectangle implements Shape { .. }
 :
```

```
interface Shape {
  double getArea();
  double getPerimeter();
  boolean contains(Point p);
  boolean excludes(Point p);
}

class Circle implements Shape { .. }
class Rectangle implements Shape { .. }
```

```
interface Shape {
  double getArea();
  double getPerimeter();
  boolean contains(Point p);
  default boolean excludes(Point p) {
    return !contains(p);
  }
}

class Circle implements Shape { .. }
class Rectangle implements Shape { .. }
```

Generics

```
class Queue<T> {
    :
}

Queue<Point> q = new Queue<>(4);
```

`Queue<T>` is the *generic class*

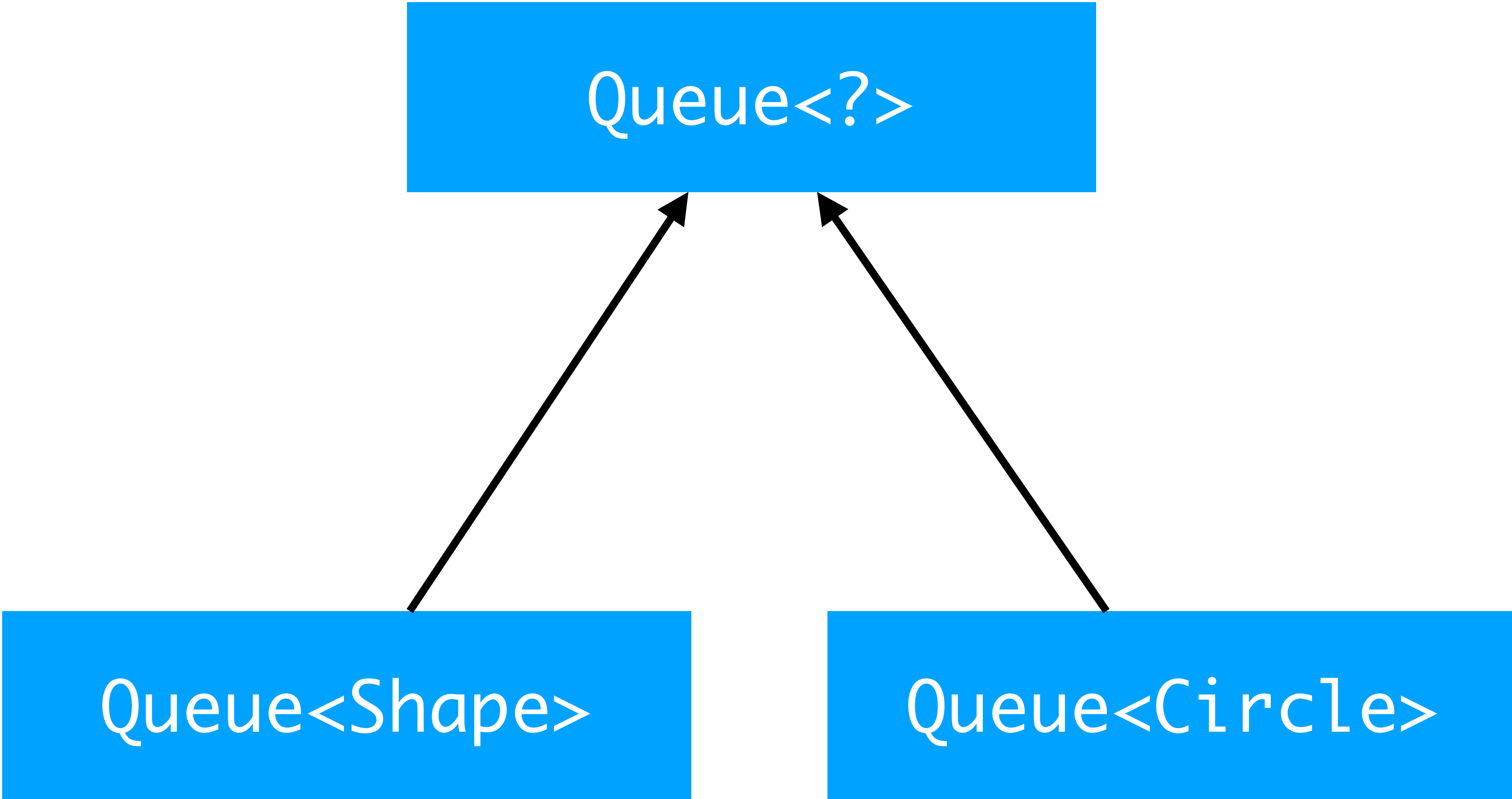
`Queue<Point>` is a *parameterised type*

`T` is the *type parameter*

`Point` is the *type argument*

Variance of Generics

- T <: S, then
    - T<X> <: S<X> (covariant)
    - but X<T> and X<S> are invariant


Queue<?>
Queue<Shape>
Queue<Circle>

Variance of Generics

- T <: S, then
  - T<X> <: S<X> (covariant)
  - X<T> and X<S> are invariant
  - `X<T>` <: `X<? extends S>`
  - `X<S>` <: `X<? super T>`

Queue<?>

Queue<? extends Shape>

Queue<Shape>

Queue<? extends Circle>

Queue<Circle>

Queue<?>

Queue<? super Circle>

Queue<Circle>

Queue<? super Shape>

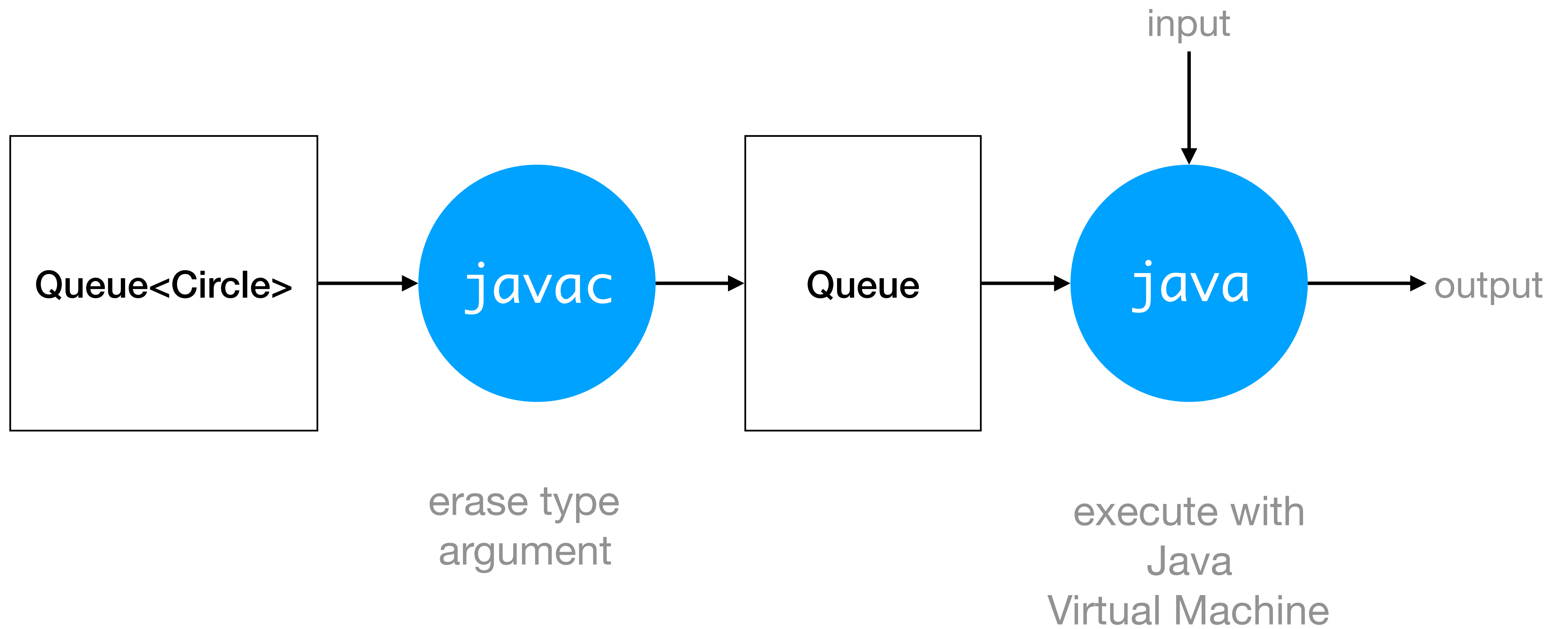
Queue<Shape>

Type Erasure

```
class Queue<T> {
   T[] objects;
}
```

type erasure

```
class Queue {
   Object[] objects;
}
```


input
Queue<Circle>
javac
Queue
java
output
erase type
argument
execute with
Java
Virtual Machine

```
class A {
  void foo(Queue<Circle> c) {}
  void foo(Queue<Point> c) {}
}
```

↓ type erasure

```
class A {
  void foo(Queue c) {}
  void foo(Queue c) {}
}
```

```
class Queue<T> {
   static int x = 1;
   static T y; // error
   static T foo(T t) {}; // error
}
```

```
class Queue<T> {
  static int x = 1;
  static T y; // error
  static <X> X foo(X t) {}; //👍
}
```

```
Queue<Circle>[] twoQs = new Queue<Circle>[2];
twoQs[0] = new Queue<Circle>();
twoQs[1] = new Queue<Point>();
```

type erasure

```
Queue[] twoQs = new Queue[2];
twoQs[0] = new Queue();
twoQs[1] = new Queue();
```

```
Queue<int> q; // error
Queue<Integer> q; // ok
```

```
Queue<Integer> q = new Queue();
```

```
q.enqueue(new Integer(8));
q.enqueue(8);
```

this is NOT a widening type conversion
this is called autoboxing

```
int i = q.dequeue();
```

Auto-unboxing type conversion

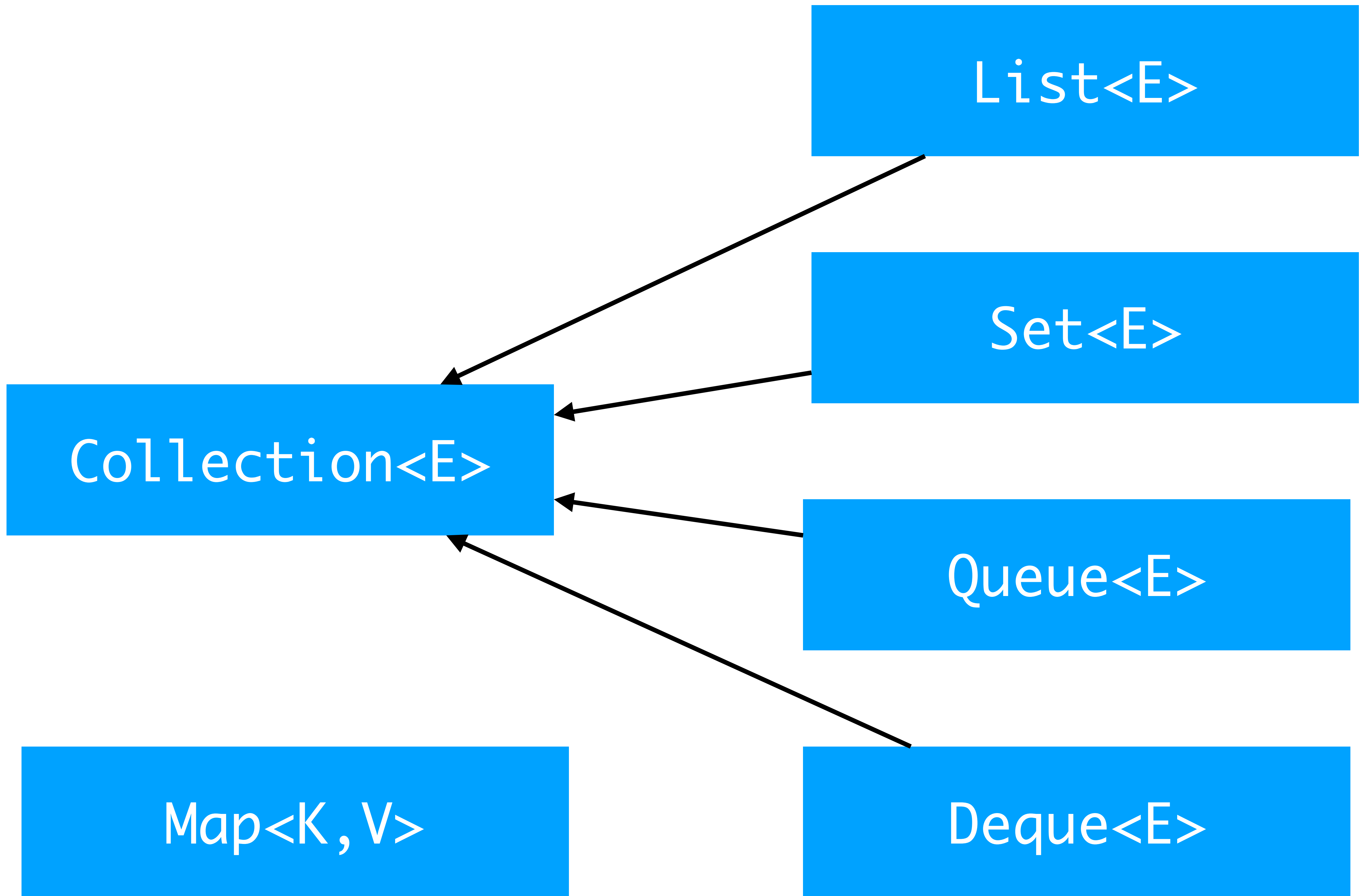
Autoboxing/unboxing is convenient but beware of performance penalty

Type Inference

```
Queue<Integer> q = new Queue<>();

Queue.foo(new Point(0,4));
```

Java Collections


List<E>
Set<E>
Collection<E>
Queue<E>
Map<K,V>
Deque<E>

```java
public interface Collection<E> extends
    Iterable<E> {
  boolean add(E e);
  boolean contains(Object o);
  boolean remove(Object o);
  void clear();
  boolean isEmpty();
  int size();
```

`contains(o)` **and** `remove(o)`

uses

`equals(o)`

```
Object[] toArray();
<T> T[] toArray(T[] a);
```

```java
boolean addAll(Collection<? extends E> c);
boolean containsAll(Collection<?> c);
boolean removeAll(Collection<?> c);
boolean retainAll(Collection<?> c);
    :
```

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

Duplicate: OK
Order: important

`List<E>`

`Collection<E>`

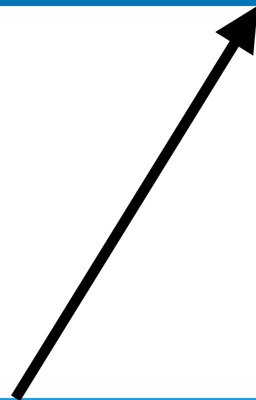
Duplicate: OK
Order: don’t care

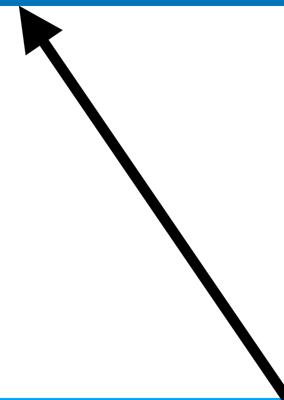
`Set<E>`

Duplicate: No
Order: Not important

AbstractList<E>

- add(i, e)
- set(i, e)
- get(i)
- remove(i)
- sort(cmp)





ArrayList<E>

-

LinkedList<E>

-

```
public interface List<E> {
      :
   default void
     sort(Comparator<? super E> c)

}
```

AbstractSet<E>

–

HashSet<E>

–

AbstractMap<K,V>

–

HashMap<K,V>

–

```
ArrayList<E>
LinkedList<E>
HashSet<E>
HashMap<K,V>
```

and many more..