Lecture 7

Functions and Lambdas

Midterm Q12

```
class A {
    A copy() { .. }
}
```

```
class B extends A {
    @Override
    B copy() { … }
}
```

Q: why is it safe for Java to allow overriding?

A: Because Java 5 onwards allows this

(It’s very good that you know this, but that didn't answer the question)

A: B is a subclass of A, widening conversion, no need to cast!

(You are explaining why we can write A a = b.copy();)

A: B's copy is creating and returning a B object, so the type is compatible.

(You are explaining why we can change the return type to B)

A: Because return type is not part of method signature, so can override.

```
class A {
    A copy() { .. }
}
```

```
class B extends A {
    @Override
    String copy() { … }
}
```

A: Everywhere A is used, B can be used.

```
class A {
    void copy(A a) { .. }
}

class B extends A {
    @Override
    void copy(B b) { … }
}
```

A: It does not violate LSP.
Behaviour is unchanged.

But compilers does not check
the semantics or verify LSP

Overriding means we can change existing code that uses A (possibly written without knowing B exists)

A: Anywhere return type of A is expected *in existing code*, it is type safe to return object of type B

Q: Anywhere arg of type A is expected *in existing code*, is it type safe to pass in object of type B?

```
class A {
    void copy(A a) { .. }
}
```

```
class A {
    void copy(A a) { .. }
}

a.copy(new A()); // ok
// C extends A
a.copy(new C()); // ok
```

```
class B extends A {
    @Override
    void copy(B b) { … }
}
```

```
a.copy(new A()); // <- error
// C extends A
a.copy(new C()); // ???
```

Previously, in cs2030..

> Each significant piece of functionality in a program should be implemented in just one place in the source code.

> Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts.

```
double genInterArrivalTime() {
    return -Math.log(this.rngArrival.nextDouble()) /
        this.customerArrivalRate;
 }

 double genServiceTime() {
    return -Math.log(this.rngService.nextDouble()) /
        this.customerServiceRate;
}
```

```
double randomExponentialValue(Random rng, double rate) {
  return -Math.log(rng.nextDouble()) / rate;
}

double generateServiceTime() {
  return randomExponentialValue(this.rngService,
      this.serviceRate);
}

double generateInterArrivalTime() {
  return randomExponentialValue(this.rngArrival,
      this.arrivalRate);
}
```

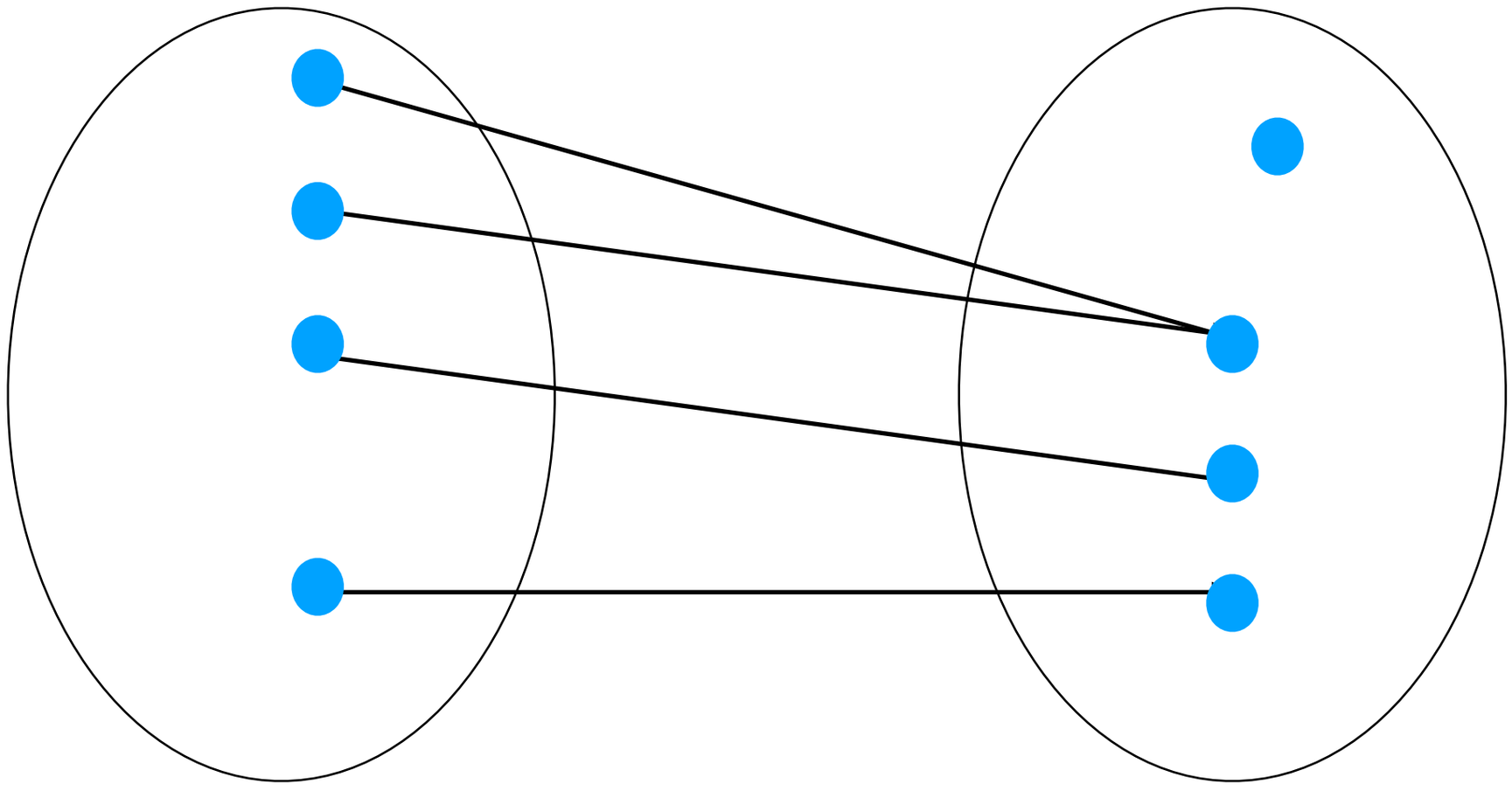
```
class Queue<T> {
    :
    }
}
```

```
Queue<Double> q = new Queue<>(..);
```

```
class EventComparator implements
    Comparator<Event> {
  public int compare(Event e1, Event e2) {
    return e1.compareTo(e2);
  }
}
events = new PriorityQueue<Event>(
    new EventComparator());
```

Functions allow abstractions over snippet of code

f : X -> Y



domain

co-domain

```
int square(int i) {
   return i * i;
}
```

```
int add(int i, int j) {
   return i + j;
}
```

```
int div(int i, int j) {
  return i / j;
}

int incrCount(int i) {
  return this.count + i;
}
```

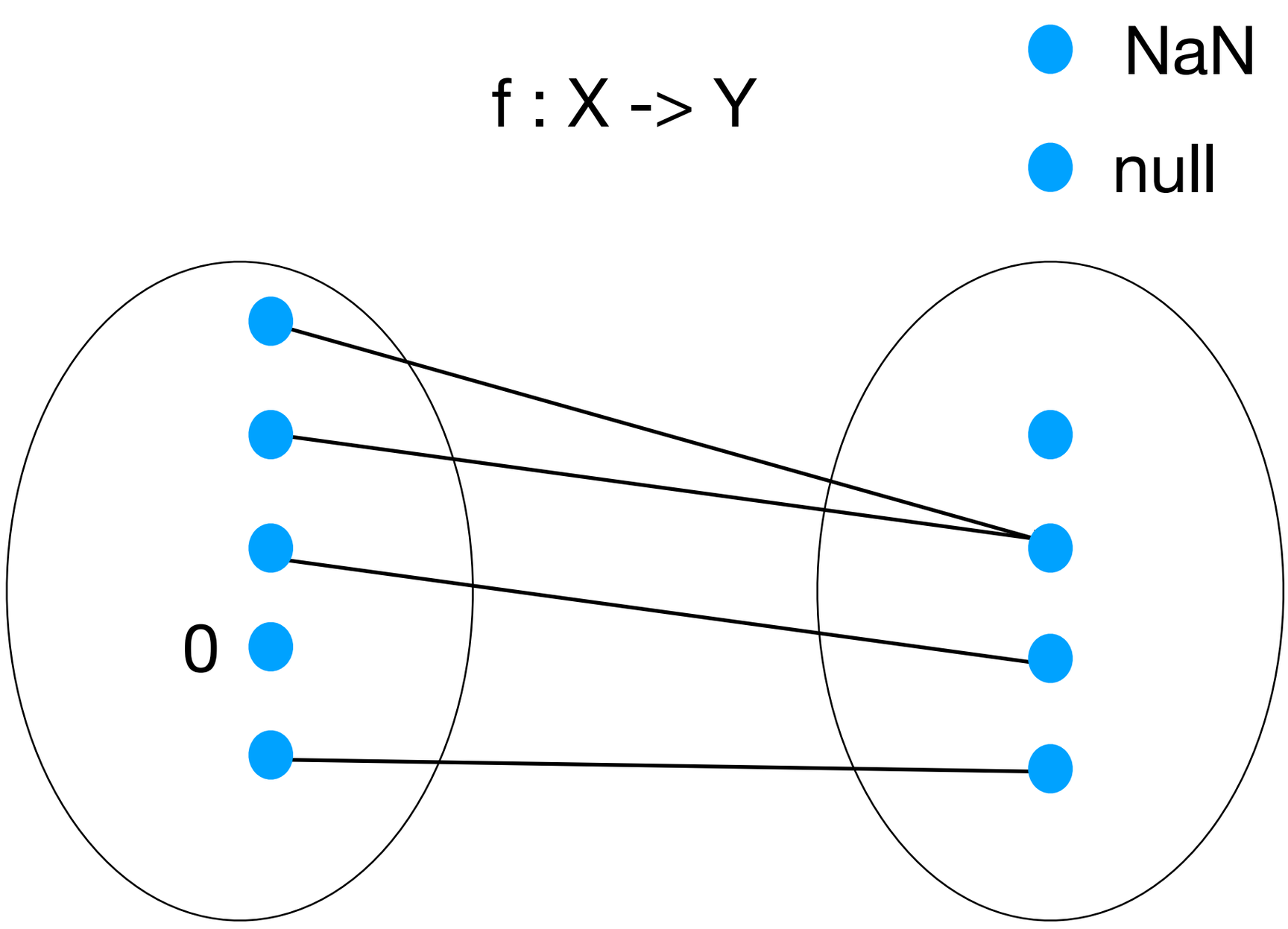
```
void incrCount(int i) {
  this.count += i;
}

int addToList(List queue, int i) {
  queue.add(i);
  return queue.size();
}
```

pure functions has no side effects

many bugs arises from:

- treating partial functions as functions
- producing values not in co-domain
- side effects


f : X -> Y
NaN
null
0
domain
co-domain

```
server.serves(customer);
```

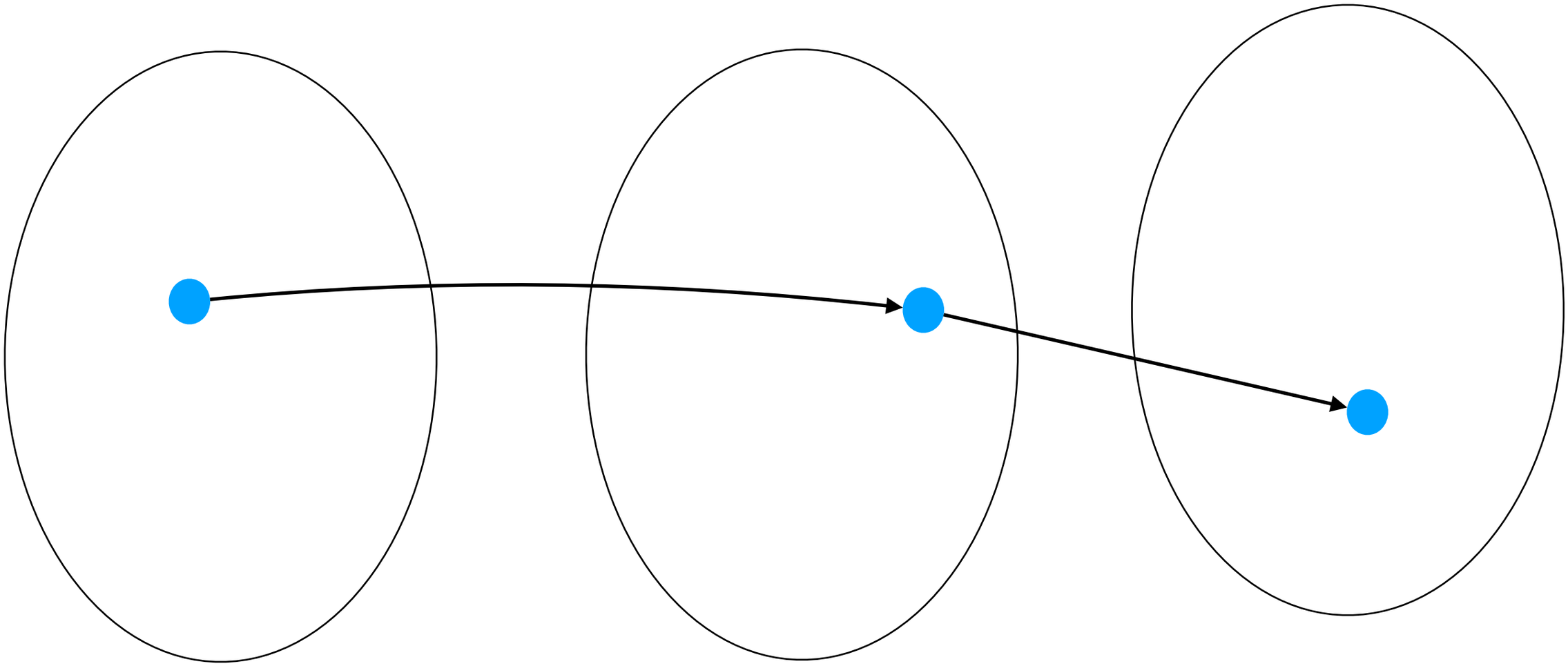
Function<T, R>

```
class Square implements Function<Integer, Integer>
{
   public Integer apply(Integer x) {
      return x*x;
   }
}
```

f ∘ g

T -> R

R -> V



- Predicate<T>
  - boolean test(T t)

- Supplier<T>
  - T get()

- Consumer<T>
  - void accept(T t)

- BiFunction<T,U,R>
  - R apply(T t, U u)