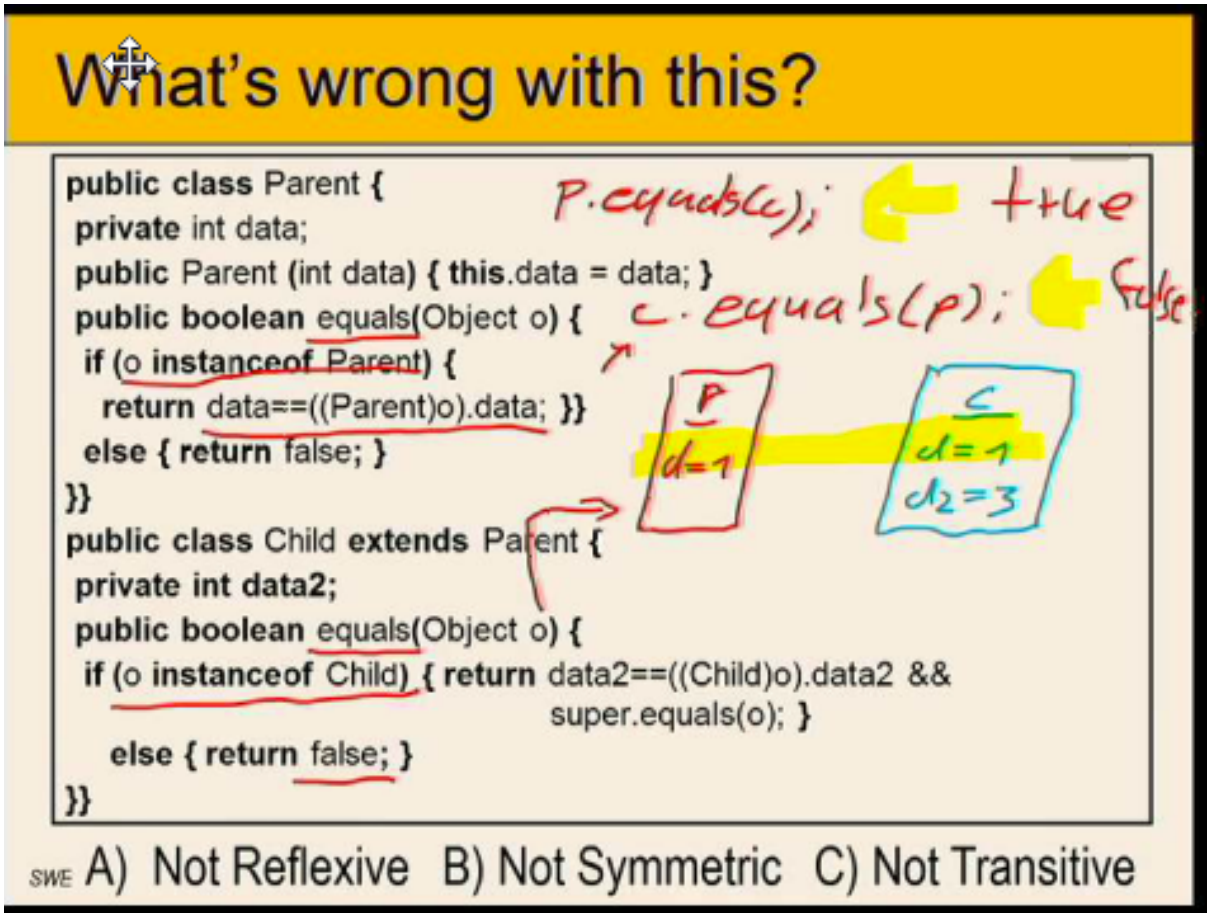
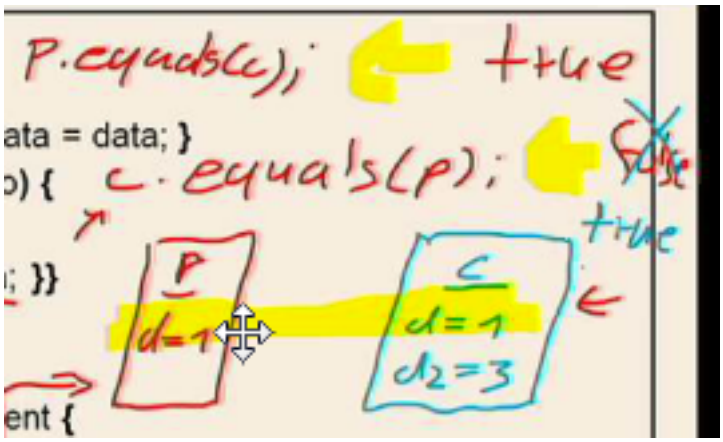
```
public class InsensitiveStr {
 private String s;
 public InsenstiveStr(String x) { s=x.toLowerCase(); }
 public boolean equals(Object o) {
  if (o instanceof InsensitiveStr) {
    InsensitiveStr c =(InsensitiveStr) o;
    return s.equals(c.s);
  } else if (o instanceof String) {
    return s.equalsIgnoreCase((String) o);
  }
  return false;
}}
```

A) Not Reflexive B) Not Symmetric C) Not Transitive

B)
InsensitiveStr. equals(str) → true
str.equals(InsensitiveStr) → false


What's wrong with this?
public class Parent {
private int data;
public Parent (int data) { this.data = data; }
public boolean equals(Object o) {
if (o instanceof Parent) {
return data==((Parent)o).data; }}
else { return false; }
}}
public class Child extends Parent {
private int data2;
public boolean equals(Object o) {
if (o instanceof Child) { return data2==((Child)o).data2 &&
super.equals(o); }
else { return false; }
}}
P.equals(c); true
c.equals(p); false
P
d=1
C
d=1
d2=3
SWE A) Not Reflexive B) Not Symmetric C) Not Transitive

B)


P.equals(c); true
c.equals(p); false
true
P
d=1
C
d=1
d2=3

```
public class Parent {
 private int data;
 public Parent (int data) { this.data = data; }
 public boolean equals(Object o) {
  if (o instanceof Parent) {
    return data==((Parent)o).data; }}
  else { return false; }
}}
public class Child extends Parent {
 private int data2;
 public boolean equals(Object o) {
  if (o instanceof Child) { return data2==((Child)o).data2 &&
                                     super.equals(o); }
    else { return super.equals(o); }
}}
```

WE A) Not Reflexive B) Not Symmetric C) Not Tra

Fix Attempt

```
public class Parent {
 private int data;
 public Parent (int data) { this.data = data; }
 public boolean equals(Object o) {
  if (o instanceof Parent) {
    return data==((Parent)o).data; }}
  else { return false; }
}}
public class Child extends Parent {
 private int data2;
 public boolean equals(Object o) {
  if (o instanceof Child) { return data2==((Child)o).data2 &&
                                     super.equals(o); }
    else { return super.equals(o); }
}}
```

c.equals(p);

x = y = z

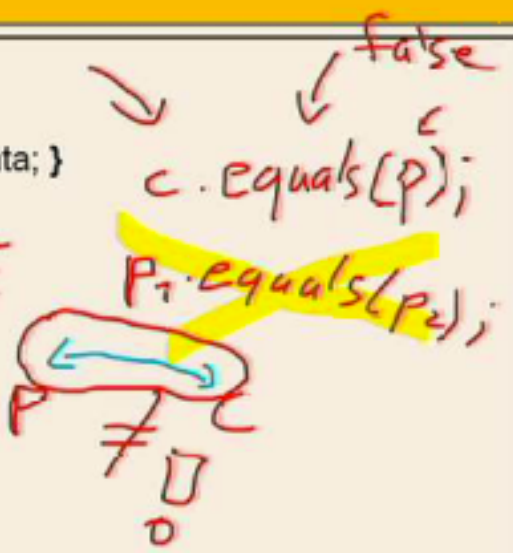
C1 d=1 d=3 — P d=1 — C2 d=1 d=4

x = z

SWE A) Not Reflexive B) Not Symmetric C) Not Transitive

Fix

```
public class Parent {
 private int data;
 public Parent (int data) { this.data = data; }
 public boolean equals(Object o) {
 if (this.getClass()==o.getClass()) {
   return data==((Parent)o).data; }}
  else { return false; }
}}
public class Child extends Parent {
 private int data2;
 public boolean equals(Object o) {
  if (o instanceof Child) { return data2==((Child)o).data2 &&
                                    super.equals(o); }
    else { return super.equals(o); }
}}
```




```
int x = (-1 / 2);
int y = (1 / 2);

System.out.println(x + "," + y);
```

A) 0,1 ✗ B) -1,0 ✗ C) 0, 0 ✓

```
int x = 0;
int y = x++ + x++ + x++;

System.out.println(y);
```

0, x=1
1, x=2
2, x=3

```
boolean isOdd(int x) {
  return (x%2) == 1;
}
```

```
static void main(String[] args) {
      System.out.println(f());
}

static boolean f() {
 try { return true; }
 finally { return false; }
}
```

```
try {
 try {
   String x = null;
   x.toString();
 } catch(NullPointerException e1) {
   int x = 10 / 0;
 } catch(ArithmeticException e2) {
   System.out.println("1");
 }
} catch(ArithmeticException e2) {
 System.out.println("2");
}
```

```
public class Test {
    Test() { f(); }
    void f() {}
}

public class Test2 extends Test {
    int i = 1;
    void f() { System.out.println(i); }

    public static void main(String[] args) {
       new Test2();
}   }
```

A) 0 ✓ B) 1 ✗ C) nothing ✗

Because: super constructor called before field initialisation!

```
public class Test {
    public static void main(String[] args) {
      int x = 60 * 60 * 24 * 1000 * 1000;

      System.out.println(x);
    }
}
```

A) 86400000000 ✗ B) 1 ✗ C) other ✓

Because: integer overflow!
Actually prints: 500654080

The int data type is a 32-bit signed two's complement integer. It has a minimum value of **-2,147,483,648** and a maximum value of **2,147,483,647**

```
class MyClass implements Cloneable {
 public int x;
 public double y;
 public List<String> z;
 public Object clone() {
  try { return super.clone(); }
  catch(CloneNotSupportedException e) { return null; }
}}
List<String> list = new ArrayList<String>();
MyClass c1 = new myClass();
c1.x = 1; c1.y = 1.2; c1.z = list;
MyClass c2 = (MyClass) c1.clone();
c1.x = 2; c1.y = 3.4; c1.z.add("a");
System.out.println(c1.x + "," + c1.y + "," + c1.z.size());
System.out.println(c2.x + "," + c2.y + "," + c2.z.size());
```

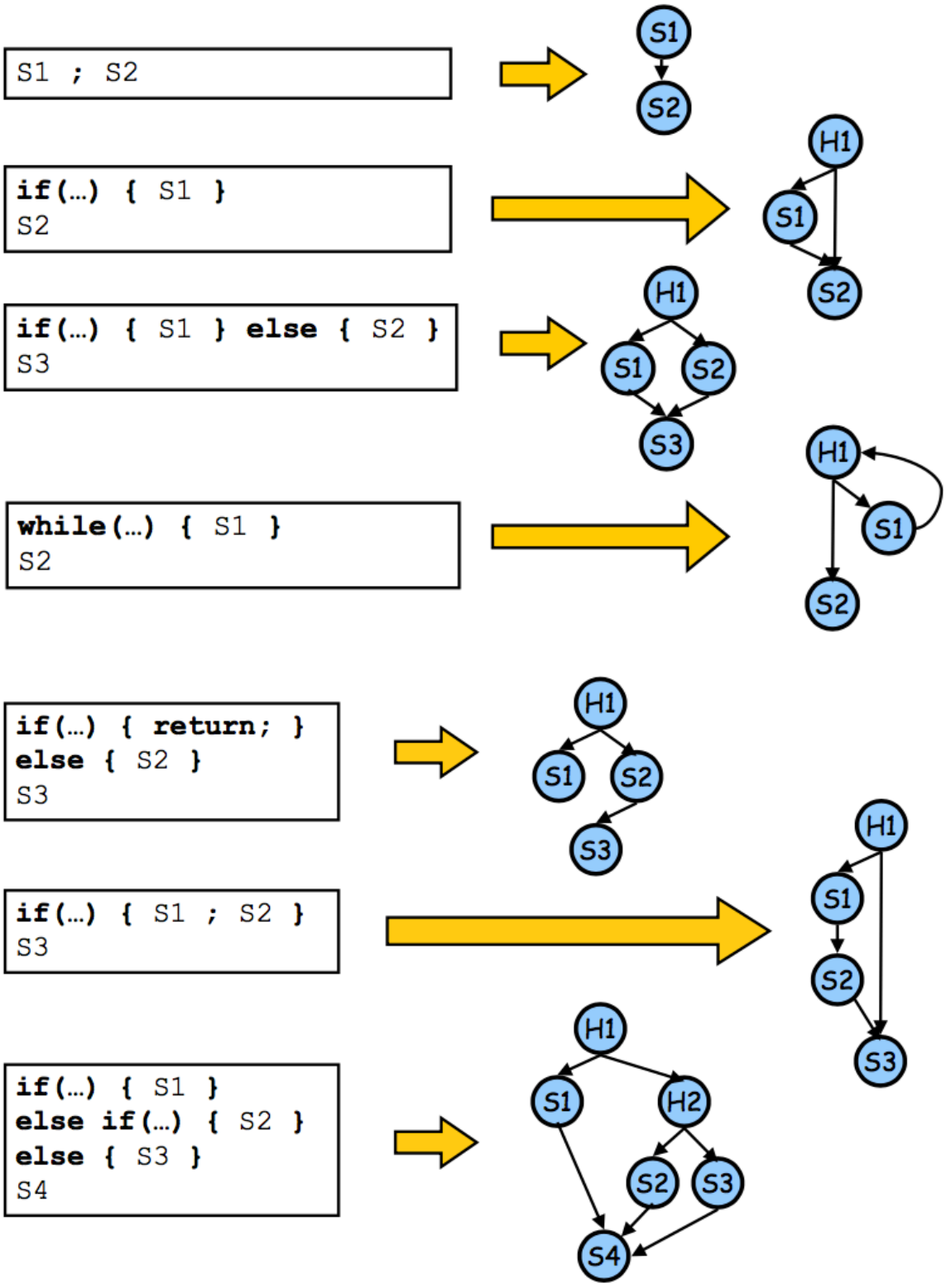
```
2,3.4,1
1,1.2,1
```

Can it fail?

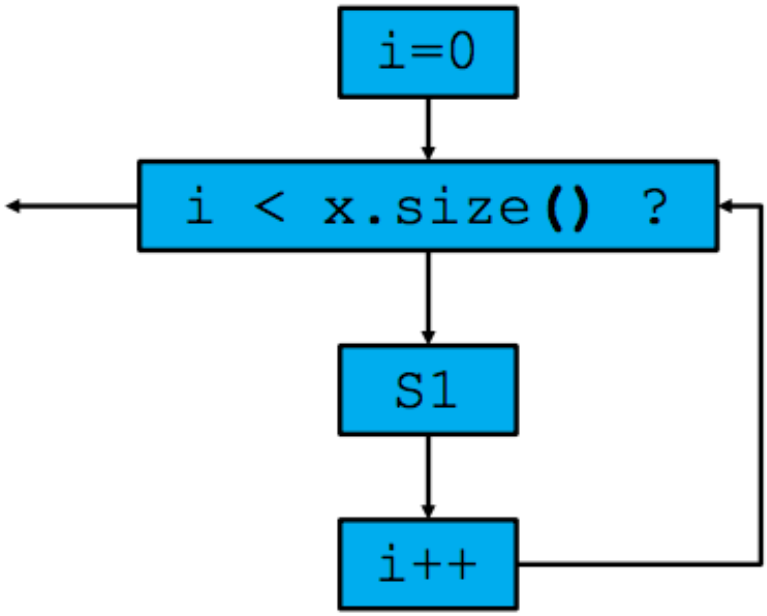
```
@Test public void test3() {
 List<String> list = new ArrayList<>(Arrays.asList("s1"));
 TableRow<String> r = new TableRow<String>(list);
 r.copy(list);
 assertEquals(r.get(1), "s1");
}
```

No, infinite loop

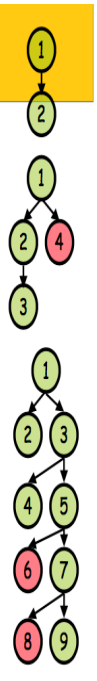
```
class TableRow<T> {
 private List<T> rows;

 public TableRow() { this.rows = new ArrayList<T>(); }

 public TableRow(List<T> rows) { this.rows = rows; }

 public T get(int index) { return rows.get(index); }

/**
  * Copy elements from this TableRow into parameter to
  */
 void copy(List<T> to) {
  for(int i=0; i < rows.size(); i++) {
   to.add(rows.get(i));
}}}
```


S1 ; S2
S1
S2
if(…) { S1 }
S2
H1
S1
S2
if(…) { S1 } else { S2 }
S3
H1
S1
S2
S3
while(…) { S1 }
S2
H1
S1
S2
if(…) { return; }
else { S2 }
S3
H1
S1
S2
S3
if(…) { S1 ; S2 }
S3
H1
S1
S2
S3
if(…) { S1 }
else if(…) { S2 }
else { S3 }
S4
H1
S1
H2
S2
S3
S4

```
for(int i=0; i < x.size(); i++) {
  S1;
}
```




```
class Card {
  private int number, suit;

  public Card(int n, int s) { number = n; suit = s; }

  public boolean equals(Object o) {
    if(o instanceof Card) {
      Card c = (Card) o;
      return c.number == number && c.suit == suit;
    }
  return false;
}

public int compareTo(Card c) {
  if(suit > c.suit) { return -1; }
  else if(suit < c.suit) { return 1; }
  else if(number < c.number) { return -1; }
  else if(number > c.number) { return 1; }
  else { return 0; }
}}
```



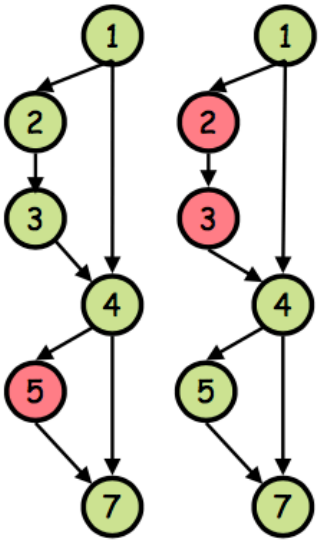
Method Coverage = 3 / 3 = **100%**
Statement Coverage = 12 / 15 = **80%**
Branch Coverage = 2 / 5 = **40%**

Calculating Code Coverage

```
@Test void testEquals() {
  assertTrue(new Card(1,2).equals(new Card(1,2)));
}
@Test void testCompareEquals() {
  assertTrue(new Card(1,2).compareTo(new Card(1,2)) == 0);
}
@Test void testCompareLess() {
  assertTrue(new Card(2,3).compareTo(new Card(2,1)) < 0);
}
@Test void testCompareGreater() {
  assertTrue(new Card(2,1).compareTo(new Card(2,3)) > 0);
}
```

- Based on these, Calculate (as %):
  - Method Coverage
  - Statement Coverage
  - Branch Coverage

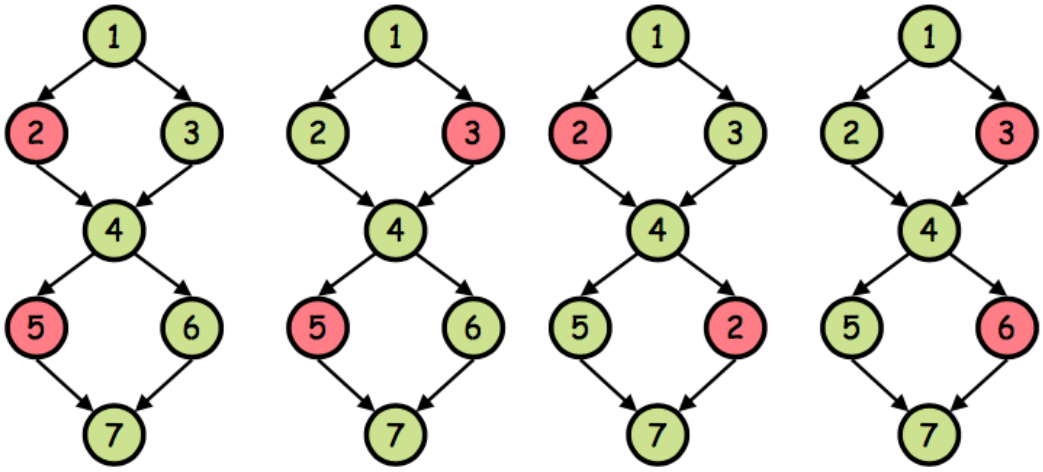
```
class Test {
 static int f(int x, int y) {
  if(x < y && y >= 0) { x = y; y = 0; }
  if(x <= y) { x = x / y; }
  return x;
}}

@Test void tester() {
 assertTrue(Test.f(0,5) == 5);
 assertTrue(Test.f(-4,-2) == 2);
}
```



- Compute (as %):
  - Statement Coverage **= 6/6 => 100%**

Execution Paths

Definition: An **execution path** a path through a method's CFG which corresponds to an execution of that method.



- Here, four distinct paths through CFG
- **100% Path Coverage**: tested all paths through CFG

Infeasible Paths

- Consider this method:

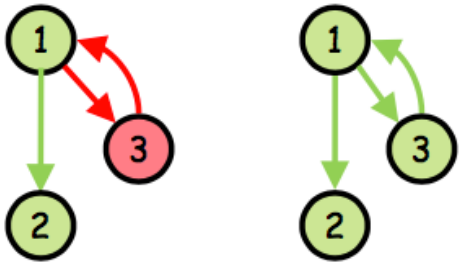
```
class Test {
 static int f(int x, int y) {
  if(x < y) { x = -y;}
  if(x >= y) { x = y; }
  return x;
}}

@Test void tester() {
 assertTrue(Test.f(0,5) == -5);
 assertTrue(Test.f(5,0) == 0);
}
```

- How many execution paths are there here?
- What path coverage is obtained here?

```
class Test {
 static int sum(int x, int y) {
  int s = 0;
  for(int i=x;i<y;++i) {
   s = s + i;
  }
  return s;
}}
```

- Q) How many execution paths are there here?
- A) Undefined

Definition: A **simple execution path** is a path through the method which iterates each loop at most once.




```
class Vec {
 private Object[] elems = new Object[16];
 private int end = 0;
 public void add(Object e) {
  if(end == elems.length) { … }
  elems[end] = e; end=end+1;
 }
 public Object get(int index) {
  if(index >= end) { throw … }
  else return elems[index];
}}
```

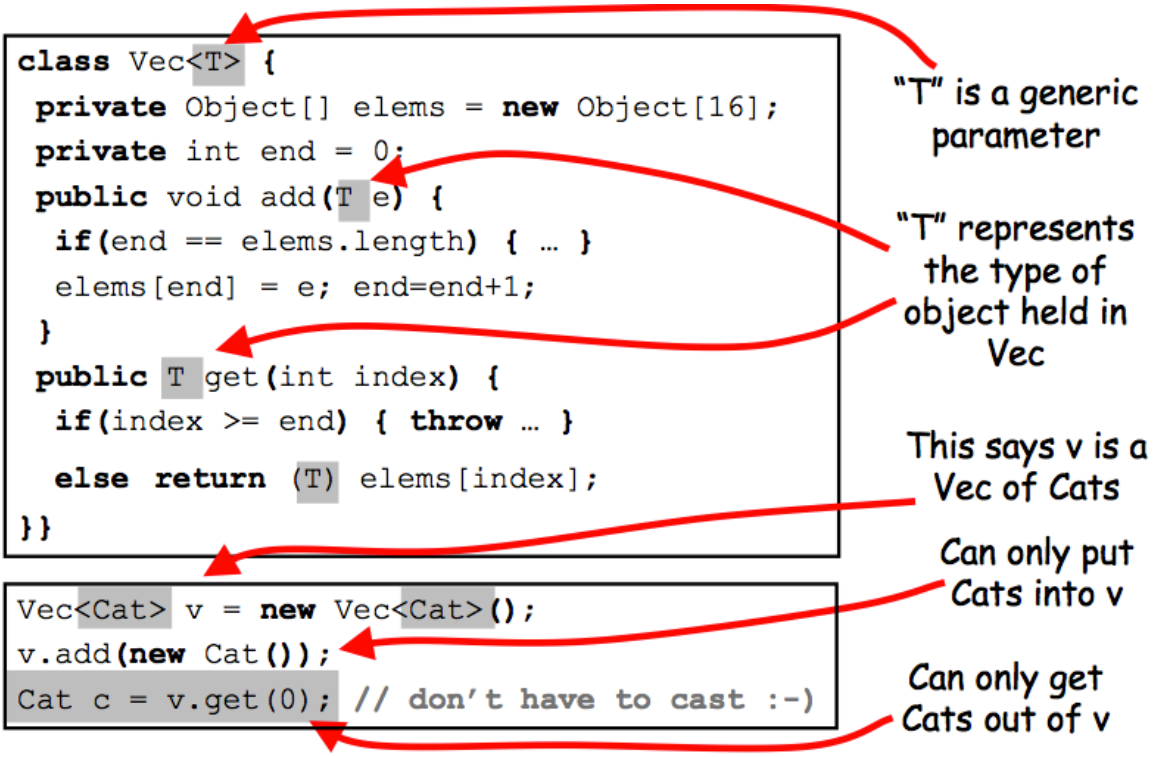
This says v is a Vec of Objects

We know this returns a Cat, but we still have to cast

```
Vec v = new Vec();
v.add(new Cat());
Cat c = (Cat) v.get(0); // have to cast :-(
```

How can we say v is a Vec of Cats?

- Before Java generics:
  - **Can only say things like: 'v' is a Vector of Objects**
  - Then, can put any Object into 'v' without restriction
  - With a Vector of just Cats, have to cast Objects to Cats
- With Java Generics:
  - **Can say things like: 'v' is a Vector of Cats**
  - Then, can only put Cats into 'v'
  - And, can only get Cats out of 'v' – no casting required!


class Vec<T> {
private Object[] elems = new Object[16];
private int end = 0;
public void add(T e) {
if(end == elems.length) { … }
elems[end] = e; end=end+1;
}
public T get(int index) {
if(index >= end) { throw … }
else return (T) elems[index];
}}
Vec<Cat> v = new Vec<Cat>();
v.add(new Cat());
Cat c = v.get(0); // don't have to cast :-)
"T" is a generic parameter
"T" represents the type of object held in Vec
This says v is a Vec of Cats
Can only put Cats into v
Can only get Cats out of v

```
class Pair {
 private Object first;
 private Object second;

 public Pair(Object f, Object s) {
  first = f; second = s;
 }
 public Object first() { return first; }
 public Object second() { return second; }
}
```

```
Pair p1 = new Pair("Cat",1);
Pair p2 = new Pair(10,20);
String c = (String) p1.first();
Integer i = (Integer) p2.first();
```

Pair Example

```
class Pair<FIRST,SECOND> {
 private FIRST first;
 private SECOND second;

 public Pair(FIRST f, SECOND s) {
  first = f; second = s;
 }
 public FIRST first() { return first; }
 public SECOND second() { return second; }
}
```

No need to cast!

```
Pair<String,Integer> p1 = new Pair<String,Integer>("Cat",1);
Pair<Integer,Integer> p2 = new Pair<Integer,Integer>(10,20);
String c = p1.first();
Integer i = p2.first();
```

A

B

C

D

```
interface Shape { void draw(Graphics g); }

class Square implements Shape { … }
class Circle implements Shape { … }

class ShapeGroup implements Shape {
 private List shapes = new ArrayList();

 ...

 public void draw(Graphics g) {
  for(Shape s : shapes) {
   s.draw(g);
}}}
```

- Which part will cause error?
- Which part should be changed?

Shape Example

```
interface Shape { void draw(Graphics g); }

class Square implements Shape { … }
class Circle implements Shape { … }
cl
 p
 .
 public void draw(Graphics g) {
  for(Shape s : shapes) {
   s.draw(g);
}}}
```

```
ShapeGroup.java:7: incompatible types
found   : java.lang.Object
required: Shape
        for(Shape s : shapes) {
                      ^
1 error
```

```
interface Shape { void draw(Graphics g); }

class Square implements Shape { … }
class Circle implements Shape { … }

class ShapeGroup implements Shape {
 private List<Shape> shapes = new ArrayList<Shape>();

 ...

 public void draw(Graphics g) {
  for(Shape s : shapes) {
   s.draw(g);
}}}
```

Group of Square or Circle?

Generic ShapeGroup ?

A

```
interface Shape { void draw(Graphics g); }

class Square implements Shape { … }
class Circle implements Shape { … }
```

B

C

```
class ShapeGroup<T> implements Shape {
 private List<T> shapes = new ArrayList<T>();

 ...

 public void draw(Graphics g) {
  for(T s : shapes) {
   s.draw(g);
}}}
```

D

Are we sure T has a draw() method?

- Q) Now what's wrong?

```
interface Shape { void draw(Graphics g); }

class Square implements Shape { … }
class Circle implements Shape { … }

class ShapeGroup<T extends Shape> implements Shape {
 private List<T> shapes = new ArrayList<T>();
 …
 public void draw(Graphics g) {
  for(T s : shapes) {
   s.draw(g);
}}}
```

```
class Point { int x; int y; }
class ColPoint extends Point { int colour; }

class PointCmp {
 Point min(Point p1, Point p2) {
  if(p1.x < p2.x || (p1.x == p2.x && p1.y < p2.y)) {
      return p1;
  } else { return p2; }
}}

ColPoint c1 = new ColPoint();
ColPoint c2 = new ColPoint();
c1 = (ColPoint) min(c1,c2);
```

Needs cast on the return value!

```
class Point { int x; int y; }
class ColPoint extends Point { int colour; }

class PointCmp {
 <T> T min(T p1, T p2) {
  if(p1.x < p2.x || (p1.x == p2.x && p1.y < p2.y)) {
      return p1;
  } else { return p2; }
}}

ColPoint c1 = new ColPoint();
ColPoint c2 = new ColPoint();
c1 = min(c1,c2);
```

```
class Point { int x; int y; }
class ColPoint extends Point { int colour; }

class PointCmp {
 <T extends Point> T min(T p1, T p2) {
  if(p1.x < p2.x || (p1.x == p2.x && p1.y < p2.y)) {
      return p1;
  } else { return p2; }
}}

ColPoint c1 = new ColPoint();
ColPoint c2 = new ColPoint();
c1 = min(c1,c2);
```

- Java compiler replaces each type parameter with its first bound
- `Object` for unbounded type parameters

```
class Vec {
 private Object[] elems = new Object[16];
 private int end = 0;
 public void add(Object e) {
  if(end == elems.length) { … }
  elems[end] = e; end=end+1;
 }
 public Object get(int index) {
  if(index >= end) { throw … }
  else return elems[index];
}}
```

Type Erasure

- Java compiler replaces each type parameter with its first bound
- `Object` for unbounded type parameters

```
class Vec<T extends Comparable> {
 private Object[] elems = new Object[16];
 private int end = 0;
 public void add(T e) {
  if(end == elems.length) { … }
  elems[end] = e; end=end+1;
 }
 public T get(int index) {
  if(index >= end) { throw … }
  else return (T) elems[index];
}}
```

- Java compiler replaces each type parameter with its first bound
- `Object` for unbounded type parameters

```
class Vec {
 private Object[] elems = new Object[16];
 private int end = 0;
 public void add(Comparable e) {
  if(end == elems.length) { … }
  elems[end] = e; end=end+1;
 }
 public Comparable get(int index) {
  if(index >= end) { throw … }
  else return (Comparable) elems[index];
}}
```

```
int countElements(List<String> v) {
 int count=0;
 for(Object o : v) { count = count+1; }
 return count;
}

ArrayList<String> v = new ArrayList<String>();
…
int total = countElements(v);
```

- Q) Does this work? 

So, `ArrayList<String>` ≤ `List<String>`

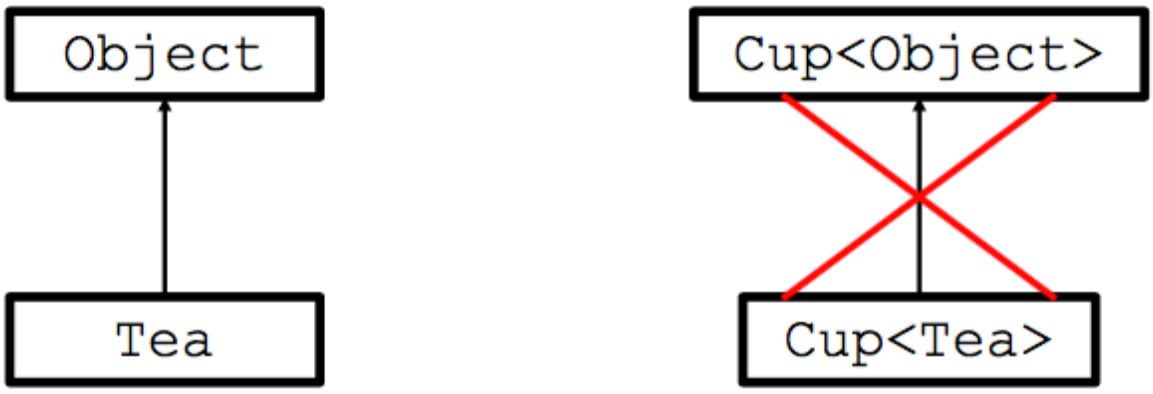
```
void doSomething(List<Object> v) {
 v.add(new Integer(1));
}

List<Object> vo = new ArrayList<Object>();
List<String> vc = new ArrayList<String>();
…
doSomething(vo);
doSomething(vc);
```

- Q) Does this work? ✗

So, `List<String>` ≤ `List<Object>`

Generics + Subtyping

- `MyClass<A>` has **NO** relationship with `MyClass<B>`, no matter whether `A` and `B` are related or not



- Cup<?>: subtype of all Cups

```
void drink(Cup<?> c) {
 System.out.println("Drink a cup of " +
   c.content.toString();
}

Cup<Tea> c1 = new Cup<Tea>(new Tea());
Cup<Coffee> c2 = new Cup<Coffee>(new Coffee());

drink(c1);
drink(c2);
```

Both are OK

```
void drink(Cup<? extends Drinkable> c) {
 c.content.drink();
 System.out.println("Drink a cup of " +
   c.content.toString();
}

Cup<Tea> c1 = new Cup<Tea>(new Tea());
Cup<Coffee> c2 = new Cup<Coffee>(new Coffee());

drink(c1);
drink(c2);
```

```
interface Drinkable { void drink(); }
class Tea implements Drinkable { … }
class Coffee implements Drinkable { … }
```

A

```
void print(List<Object> os) {
 for(Object o : os) { System.out.println(o); }
}
List<String> y = new ArrayList<String>();
print(y);
```

B

```
void print(List<? extends Object> os) {
 for(Object o : os) { System.out.println(o); }
}
List<String> y = new ArrayList<String>();
print(y);
```

- Q) Which are working?

A ✗ B ✓ Both ✗ None ✗

✗

```
void foo(List<?> x) {
 x.set(0, x.get(0));
}
```

This is an `Object`

Cannot confirm what type of Object to set

✓

```
void foo(List<?> x) {
 fooHelper(x);
}
// Helper method created so that the wildcard can be captured
// through type inference.
<T> void fooHelper(List<T> x) {
 x.set(0, x.get(0));
}
```

```
class Point { int x; int y; … }
class ColPoint extends Point { int colour; }

class PointGroup<T extends Point> {
 private List<T> points = …;

 public void write(List<? super Point> out) {
  out.addAll(points);
 }
}
```

- Here, "super" indicates a **lower bound**
  - i.e. **Cannot** be subtype of Point!
  - Why is this useful?

```
public class Point {
 double x, y;

 public boolean equals(Object o) {
  if (this == o) return true;
  if (o == null) return false;
  if (getClass() != o.getClass()) return false;

  Point p = (Point) o;
  if (x != p.x) return false;
  return y == p.y;
 }}
```

```
String s1 = new String("X");
String s2 = new String("Y");
```

A

```
Class<String> c1 = s1.getClass();
```

B

```
Class<? extends String> c2 = s2.getClass();
```

Which can work?

A ✗ B ✓

getClass

```
public final Class<?> getClass()
```

Returns the runtime class of this Object. The returne

The actual result type is `Class<? extends |X|>`

- `.class` syntax to get the class without giving instance

```
System.out.println(String.class == "i".getClass());
System.out.println(String.class.getName());
System.out.println("i".getClass().getName());
```

Output:

```
true
java.lang.String
java.lang.String
```

- **`.class`** syntax can be used for primitive types
- `getClass()` cannot be used for primitive types

```
int a = 1;
Class c1 = a.getClass();   // compile-time error
Class c2 = int.class;      // correct
```

- Can be used for arrays

```
System.out.println(int[].class.getname());
System.out.println(Double[][].class.getname());
```

```
[I
[[Ljava.lang.Double;
```

- If the fully-quantified name is known
- Static method `forName()`

```
// c1: java.lang.String
Class c1 = Class.forName("java.lang.String");
// c2: double[]
Class c2 = Class.forName("[D");
// c3: Integer[][]
Class c3 = Class.forName("[[Ljava.lang.Integer;");
```

```
class SimpleClass {
 public void aSimpleMethod() { … }
 public void anotherSimpleMethod() { … }
 private void privateMethod() { … }
}
```

```
Object o = new SimpleClass();
Class c = o.getClass();
Method[] ms = c.getDeclaredMethods();
for(Method m : ms) {
  System.out.println("o has method: " + m.getName());
}
```

Output:

```
o has method: aSimpleMethod
o has method: anotherSimpleMethod
o has method: privateMethod
```

- `Integer.class = Integer.getClass()`
- `Integer.class != int.class`
- `Integer.TYPE = int.class`

```
Integer a = 1;
Class<? extends Integer> c1 = int.class;
Class<? extends Integer> c2 = a.getClass();
Class<? extends Integer> c3 = Integer.class;
System.out.println(c1 == c2);
System.out.println(c1 == c3);
System.out.println(c1 == Integer.TYPE);
```

A) false ✓
false
true

B) false ✗
true
true

C) true ✗
false
true

```
import java.lang.reflect.*;

class Test {
  public int afield = 1;

  public static void main(String[] args) {
    Test o = new Test();
    Class c = o.getClass();
    try {
      Field f = c.getField("afield");
      System.out.println("GOT: " + f.get(o));
      f.set(o,2);
      System.out.println("NOW: " + o.afield);
    } catch(NoSuchFieldException e) {…}
      catch(IllegalAccessException e) {…}
}}
```

GOT: 1
NOW: 2

```
import java.lang.reflect.*;

class Test {
  private int afield = 1;

  public static void main(String[] args) {
    Test o = new Test();
    Class c = o.getClass();
    try {
      Field f = c.getField("afield");
      System.out.println("GOT: " + f.get(o));
      f.set(o,2);
      System.out.println("NOW: " + o.afield);
    } catch(NoSuchFieldException e) {…}
      catch(IllegalAccessException e) {…}
}}
```

java.lang.NoSuchFieldException: aField

```
import java.lang.reflect.*;

class Test {
  private int afield = 1;

  public static void main(String[] args) {
    Test o = new Test();
    Class c = o.getClass();
    try {
      Field f = c.getDeclaredField("afield");
      System.out.println("GOT: " + f.get(o));
      f.set(o,2);
      System.out.println("NOW: " + o.afield);
    } catch(NoSuchFieldException e) {…}
      catch(IllegalAccessException e) {…}
}}
```

```
GOT: 1
NOW: 2
```

```
public class Shape {
  public class Square {
    private int x, y, width, height;
    public Square(int x, int y, int width, int height) { … }
  }
}
```

```
public class ShapeTest {

  public static void main(String[] args) {
    Shape parent = new Shape();
    Shape.Square square = parent.new Square(1,1,2,3);
  }
}
```

- **Work or not?** **A) Yes** ✓ **B) No** ✗

```
public class Shape {
  private class Square {
    private int x, y, width, height;
    private Square(int x, int y, int width, int height) { … }
  }
}
```

```
public class ShapeTest {

  public static void main(String[] args) {
    Shape parent = new Shape();
    Shape.Square square = parent.new Square(1,1,2,3);
  }
}
```

- **Work or not?** **A) Yes**  **B) No** 

```
public class Shape {
  private class Square {
    private int x, y, width, height;
    private Square(int x, int y, int width, int height) { … }
  }

  public static void main(String[] args) {
    Shape parent = new Shape();
    Shape.Square square = parent.new Square(1,1,2,3);
  }
}
```

- Not static – needs an parent object!

```
public class MyList<T> implemen
  private T[] data;

  public MyList(T[] d) { data = d; }

  public Iterator<T> iterator() { return new EvenIter(); }

  private class EvenIter implements Iterator<T> {
    private int pos = 0;

    public boolean hasNext() { return pos < data.length; }

    public T next() {
      T next = data[pos]; pos
      return next;
    }
}}
```

Enclosing class can construct and return instances of inner class

Other classes cannot construct instances as it's private

```
  public T next() {
    T next = MyList.this.data[pos]; pos += 2;
    return next;
  }
}}
```

Explicit this-scoping

A

B

C

D

```
public class Outer {
  private class Inner { … }

  public static class StaticInner { … }

  public static void main (String[] args) {
    Inner c1 = new Outer.Inner();

    Inner c2 = new Outer().new Inner();

    StaticInner c3 = new Outer.StaticInner();

    StaticInner c4 = new Outer().new StaticInner();
  }
}
```

A ✗ B ✓ C ✓ D ✗

```
public class LocalClassExample {
  public static void outMethod() {
    final int number = 1;
    class Inner {
      public void printNumber() { System.out.println(number); }
    }
    Inner c = new Inner();
    c.printNumber();
  }
  public static void main(String[] args) {
    outMethod();
}}
```

- In Java 8, work or not? ✓

```
public class LocalClassExample {
  public static void outMethod() {
    int number = 1;
    class Inner {
      public void printNumber() { System.out.println(number); }
    }
    Inner c = new Inner();
    c.printNumber();
  }
  public static void main(String[] args) {
    outMethod();
}}
```

- In Java 8, work or not? 

Local Classes

```
public class LocalClassExample {
  public static void outMethod() {
    int number = 1;
    class Inner {
      public void printNumber() { System.out.println(number); }
    }
    number = 2;
    Inner c = new Inner();
    c.printNumber();
  }
  public static void main(String[] args) {
    outMethod();
}}
```

- In Java 8, work or not? 

- Cannot have static methods (same as Inner Classes)
- Must be non-static, so cannot declare interfaces as local classes
- Cannot have static member, unless it's final primitive (constant)

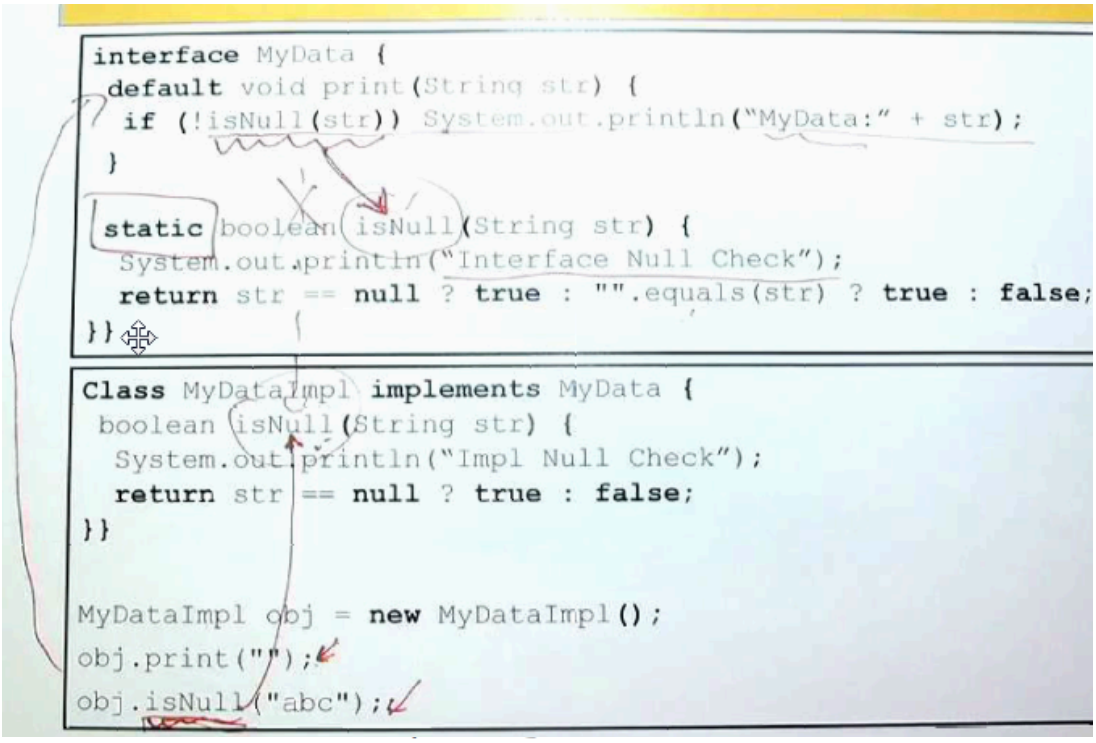
```
public void greetInEnglish(String name) {
   interface Greeting { public void greet(); }
   class EnglishGreeting implements Greeting {
     public static String prefix;
     public static final String HELLO = "Hello! ";
     public void greet() {
       System.out.println(HELLO + name);
}}}
```

- Which **default** `log(String str)` to choose from?
- `Interface1` or `Interface2`?

- The common default method(s) must be overridden

Default Methods

```
class MyClass implements Interface1, Interface2 {
 @Override void method1(String str) { … }
 @Override void method2(){ … }
 // must override common default method(s)
 @Override default void log(String str) {
  System.out.println("MyClass logging:" + str);
 }
}
```



```
interface MyData {
 default void print(String str) {
  if (!isNull(str)) System.out.println("MyData:" + str);
 }

 static boolean isNull(String str) {
  System.out.println("Interface Null Check");
  return str == null ? true : "".equals(str) ? true : false;
}}

Class MyDataImpl implements MyData {
 boolean isNull(String str) {
  System.out.println("Impl Null Check");
  return str == null ? true : false;
}}

MyDataImpl obj = new MyDataImpl();
obj.print("");
obj.isNull("abc");
```

```
Collections.sort(ls, new Comparator<String>(){
  public int compare(String s1, String s2) {
    return s1.compareToIgnoreCase(s2);
}});
```

- Use Lambdas to make it more compact

```
Collections.sort(ls, (s1,s2) -> s1.compareToIgnoreCase(s2));
```

Parameters — Method body

sort(List<T> list, Comparator<? super T> c)

Sorts the specified list according to the order induced by the specified comparator.

swap(List<?> list, int i, int j)

- Normal

```
JButton b = new JButton("Press Me");
b.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent e) {
  System.out.println("Button pressed");
}});
```

- Lambda

```
JButton b = new JButton("Press Me");
b.addActionListener(
  e -> System.out.println("Button pressed"));
```

- A function takes `(T t)` and returns `(R r)`

```
interface Function<T,R> {
 R apply(T t);

 // identity function always returns the input
 static <T> Function<T,T> identity() { return t -> t; }

 // first apply "before", then apply this function
 default <V> Function<V,R>
    compose(Function<? super V,? extends T> before){
  return (V v) -> apply(before.apply(v));
 }

 // first apply this function, then apply "after"
 default <V> Function<T,V>
    andThen(Function<? super R,? extends V> after){
  return (T t) -> after.apply(apply(t));
}}
```

- What does the following output?

```
Function<Integer,Integer> mul2 = x -> x*2;
Function<Integer,Integer> add2 = x -> x+2;

System.out.println(
   mul2.andThen(add2).apply(1));
System.out.println(
   add2.andThen(mul2).apply(1));
System.out.println(
   add2.compose(mul2).apply(1));
```

```
4              // (1*2)+2
6              // (1+2)*2
4              // (1*2)+2
```

```
interface Predicate<T> {
 boolean test(T t);

 // a predicate to test if two arguments are equal
 static <T> Predicate<T> isEqual(Object targetRef) { … }

 default Predicate<T> negate() { return (t) -> !test(t); }

 default Predicate<T> or(Predicate<? super T> other) {
  return (t) -> test(t) || other.test(t);
 }

 default Predicate<T> and(Predicate<? super T> other) {
  return (t) -> test(t) && other.test(t);
}}
```

```
Predicate<Person> males = p -> p.getGender() == "male";
Predicate<Person> young = p -> p.getAge() < 18;

Predicate<Person> youngFemales = [???]
```

```
Predicate<Person> youngFemales = young.and(males.negate());
```

```
class Person {
 public String name;
 public Person(String name){ this.name = name; }
}

Optional<Person> op1 = Optional.ofNullable(null);
Optional<Person> op2 = Optional.of(new Person("Eva"));
Person p1 = op1.orElse(new Person("Allen"));
Person p2 = op2.orElse(new Person("Allen"));

if(op1.isPresent()) System.out.println(p1.name);
if(op2.isPresent()) System.out.println(p2.name);
```

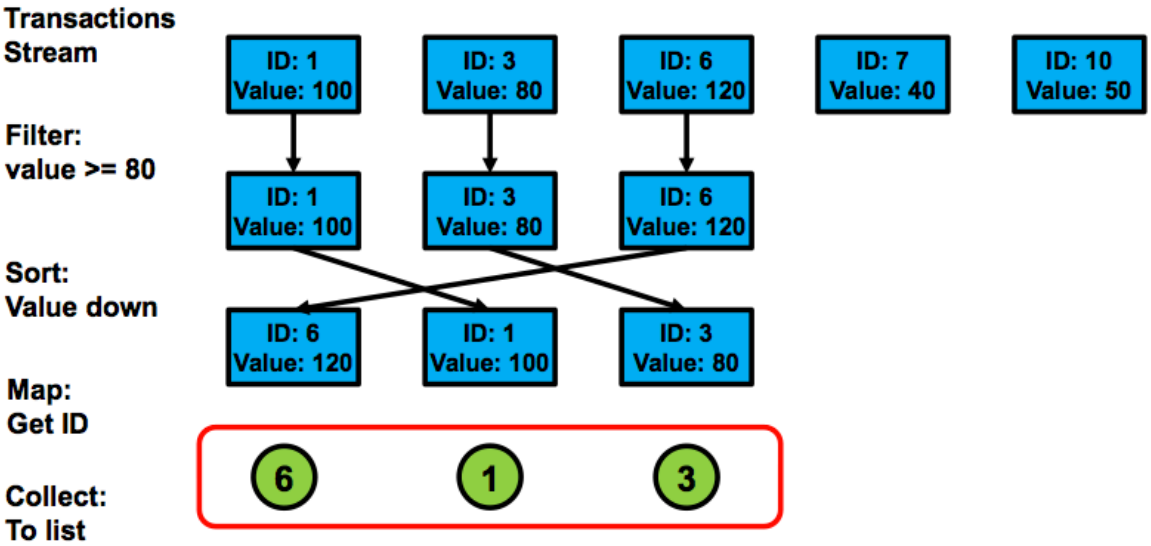
A) Eva Allen

B) Eva Eva

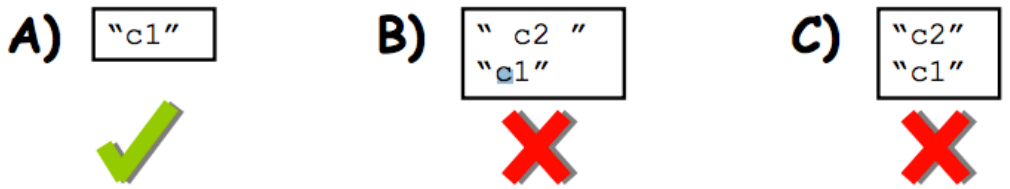
C) Eva

D) Cannot compile

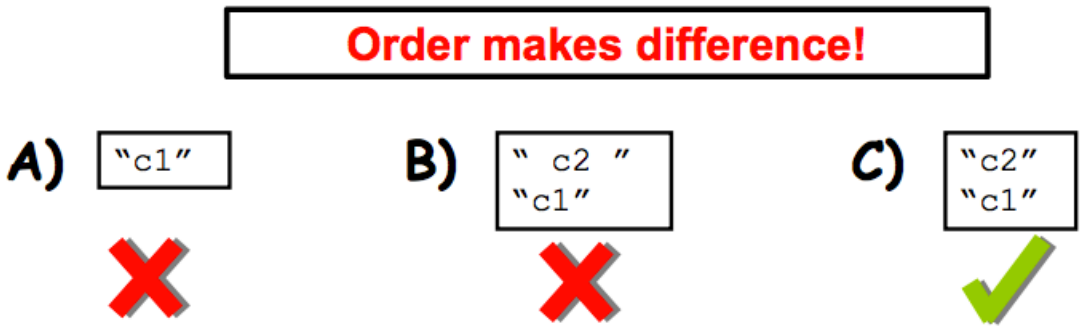
- Not `InputStream/OutputStream`
- Rich library to query and process collections



```
List<String> myList =
    Arrays.asList("a1 ", " a2 ", "b1", " c2 ", "c1");

myList = myList.stream()
    .filter(s -> s.startsWith("c"))
    .map(s -> s.trim())
    .collect(Collectors.toList());

for (String s : myList) System.out.println(s);
```

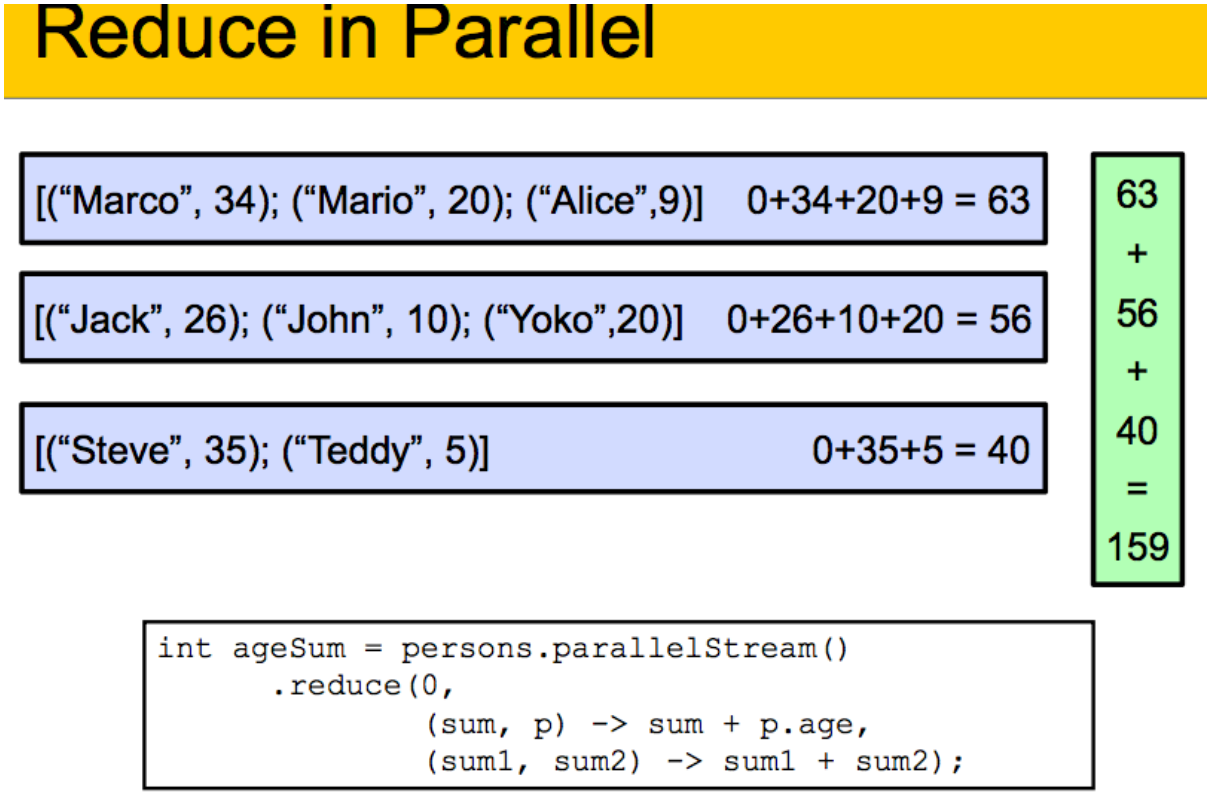



```
List<String> myList =
    Arrays.asList("a1 ", " a2 ", "b1", " c2 ", "c1");

myList = myList.stream()
    .map(s -> s.trim())
    .filter(s -> s.startsWith("c"))
    .collect(Collectors.toList());

for (String s : myList) System.out.println(s);
```



This is equivalent to:

```
T result = identity;

for (T element : this stream)

    result = accumulator.apply(result, element)
return result;
```



优先级由高到低依次为：this.show(O)、super.show(O)、this.show((super)O)、super.show((super)O)

比如④，a2.show(b)，a2是一个引用变量，类型为A，则this为a2，b是B的一个实例，于是它到类A里面找show(B obj)方法，没有找到，于是到A的super(超类)找，而A没有超类，因此转到第三优先级this.show((super)O)，this仍然是a2，这里O为B，(super)O即(super)B即A，因此它到类A里面找show(A obj)的方法，类A有这个方法，但是由于a2引用的是类B的一个对象，B覆盖了A的show(A obj)方法，因此最终锁定到类B的show(A obj)，输出为"B and A”。

（一）相关类

```
1.      class A ...{
2.          public String show(D obj)...{
3.              return ("A and D");
4.          }
5.          public String show(A obj)...{
6.              return ("A and A");
7.          }
8.      }
9.      class B extends A...{
10.         public String show(B obj)...{
11.             return ("B and B");
12.         }
13.         public String show(A obj)...{
14.             return ("B and A");
15.         }
16.     }
```

```
17.     class C extends B...{}
18.     class D extends B...{}
```

（二）问题：以下输出结果是什么？

```
1.      A a1 = new A();
2.          A a2 = new B();
3.          B b = new B();
4.          C c = new C();
5.          D d = new D();
6.          System.out.println(a1.show(b));  ①
7.          System.out.println(a1.show(c));  ②
8.          System.out.println(a1.show(d));  ③
9.          System.out.println(a2.show(b));  ④
10.         System.out.println(a2.show(c));  ⑤
11.         System.out.println(a2.show(d));  ⑥
12.         System.out.println(b.show(b));   ⑦
13.         System.out.println(b.show(c));   ⑧
14.         System.out.println(b.show(d));   ⑨
```

（三）答案

① A and A
② A and A
③ A and D
④ B and A
⑤ B and A
⑥ A and D
⑦ B and B
⑧ B and B
⑨ A and D