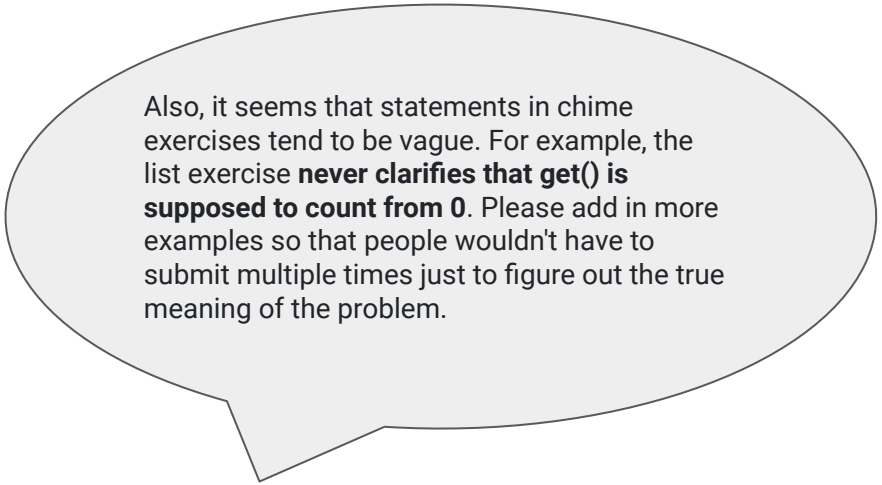
Object Oriented Programming Examples class

November 2021
Professor Andrew Rice

These slides are on the course website if you want to follow along.
(Some of the code is necessarily smaller than ideal for presenting.)

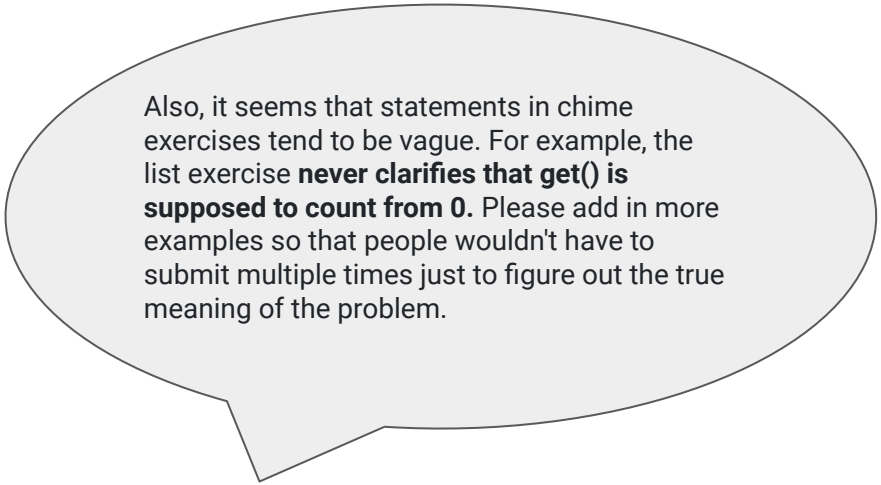
Course suggestions



Course suggestions

Clarified specification of get() for the list exercise

I will make any other **specific** suggestions people have....



Syntax

Could we go over some Java syntax? E.g. when the 'this' keyword is needed in classes, the difference between private and public, why some types are capitalised and some aren't, and which brackets to use when.

Please can we go through java syntax for beginners. e.g. Classes, Private vs Public variables, what brackets to use when, when to use new, semicolons?, I understand the concepts but it takes a long time for java to understand what I want.

package declaration (has to be first)

```
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)",
                        first,
                        second);
 }
}
```

P.S. there's a long standing debate about whether Pair is a good thing....I think it is not.
Java 16: `record NameAndAge(String name, int age) {}`

class declaration. Every file must define at least one class (or class-like thing). If the class is public then the file name must match.

By convention class names are in UpperCamelCase.

```
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)",
                        first,
                        second);
 }
}
```

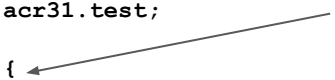
class declaration. Every file must define at least one class (or class-like thing). If the class is public then the file name must match.

By convention class names are in UpperCamelCase.

```
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)",
                        first,
                        second);
 }
}
```

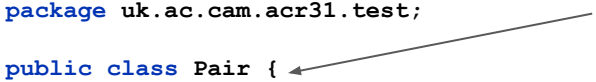
type names in lower case are probably primitive types: long, double, float, etc.

Curly braces denote *blocks*. Things like the body of a method, class, for-loop etc.



```
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)",
                        first,
                        second);
 }
}
```

Curly braces denote *blocks*. Things like the body of a method, class, for-loop etc.



```
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)",
                        first,
                        second);
 }
}
```

Curly braces also denote array bodies e.g.

int[] i = new int[] {1,2};

Curly braces denote *blocks*. Things like the body of a method, class, for-loop etc.

```
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)",
                        first,
                        second);
 }
}
```

Curly braces also denote array bodies e.g.

int[] i = new int[] {1,2};

Square brackets are for array indexing: e.g. i[0] and array types e.g. int[]

Curly braces denote *blocks*. Things like the body of a method, class, for-loop etc.

```
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)",
                        first,
                        second);
 }
}
```

Round brackets delimit argument (or parameter) lists

Curly braces also denote array bodies e.g.

int[] i = new int[] {1,2};

Square brackets are for array indexing: e.g. i[0] and array types e.g. int[]

Angle brackets are for generic types: e.g. LinkedList<String>

Curly braces denote *blocks*. Things like the body of a method, class, for-loop etc.

```
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)",
                        first,
                        second);
 }
}
```

Curly braces also denote array bodies e.g.

int[] i = new int[] {1,2};

Round brackets delimit argument (or parameter) lists

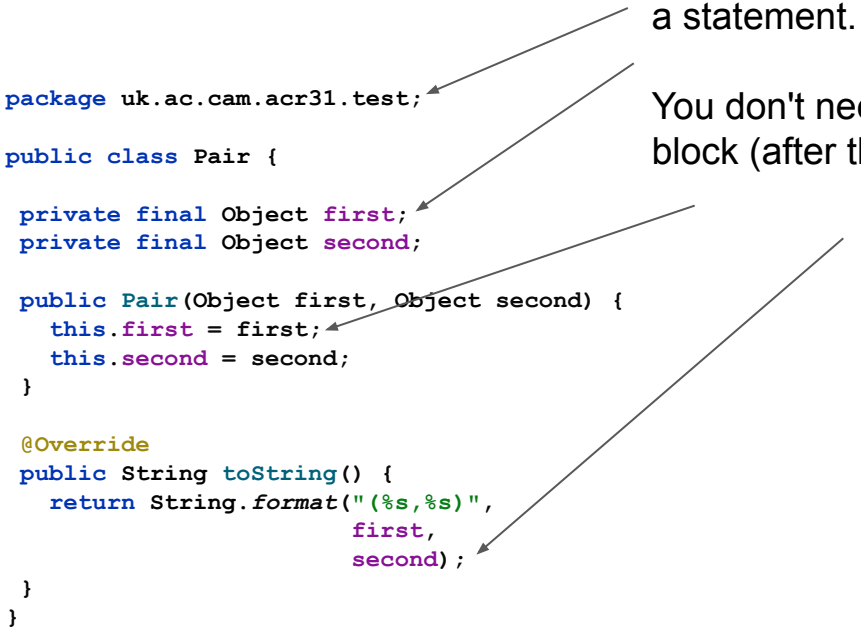
Square brackets are for array indexing: e.g. i[0] and array types e.g. int[]

Access modifiers determine who can 'see' the 'thing'. Private fields are only accessible to members of the same **class**.


```
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)",
                        first,
                        second);
 }
}
```

Semi-colons indicate the end of a statement.

You don't need them to end a block (after the curly brace)



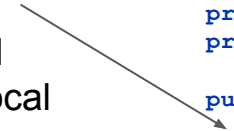
```
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)",
                        first,
                        second);
 }
}
```

Constructors are named with the name of the class

```
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)",
                        first,
                        second);
 }
}
```

'this' is a *reference* to the current object. In this case 'this' has type 'Pair'.

One use of 'this' is to disambiguate when a field name is shadowed by a local variable



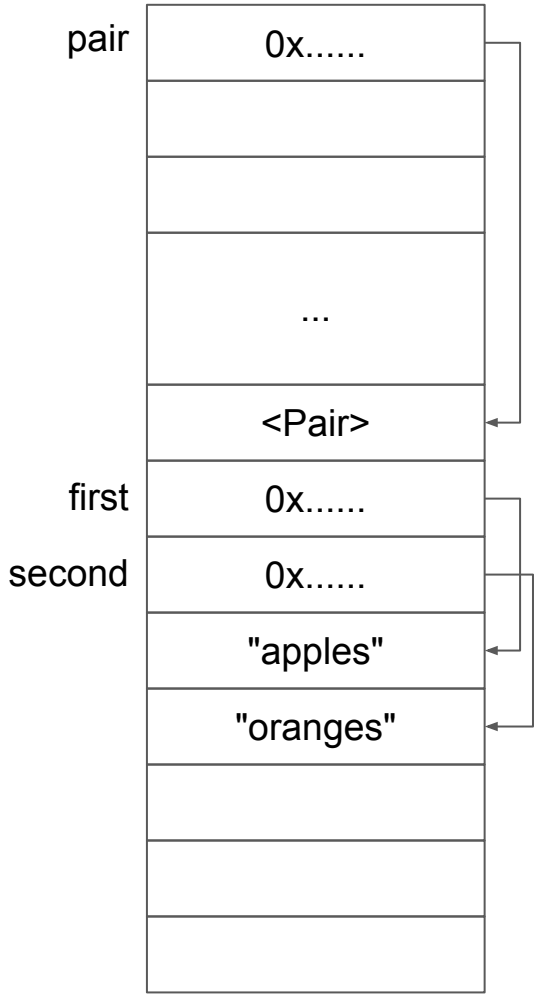
```
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)",
                        first,
                        second);
 }
}
```

```
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)",
                        first,
                        second);
 }
}
```

@Override is an *annotation* that tells the compiler to do some extra stuff. In this case it says please check that this method overrides one from the superclass

new is used to create a new instance of an object

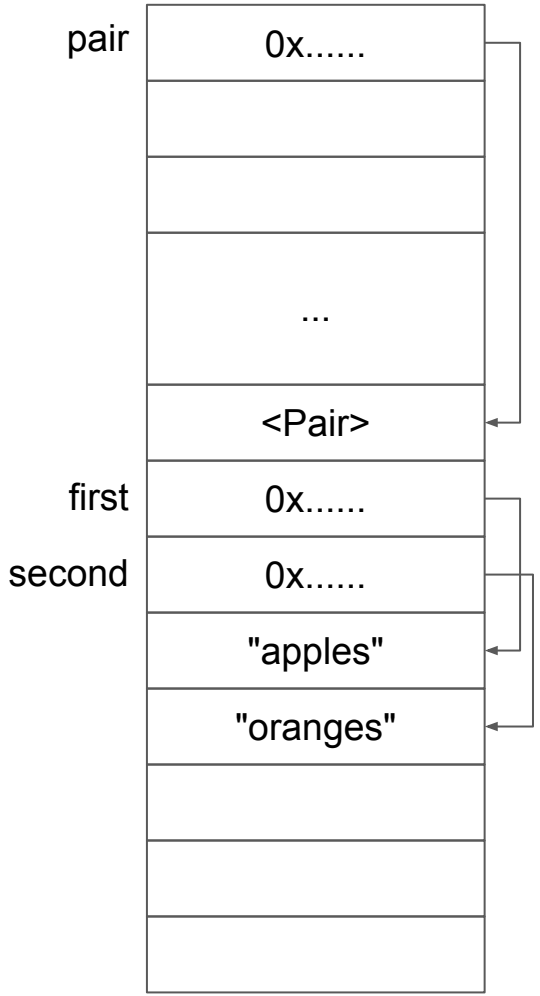
```
Pair pair = new Pair("apples","oranges");
int i = 4;
double[] d = new double[] { 1.0, 2.0 };
```



new is used to create a new instance of an object

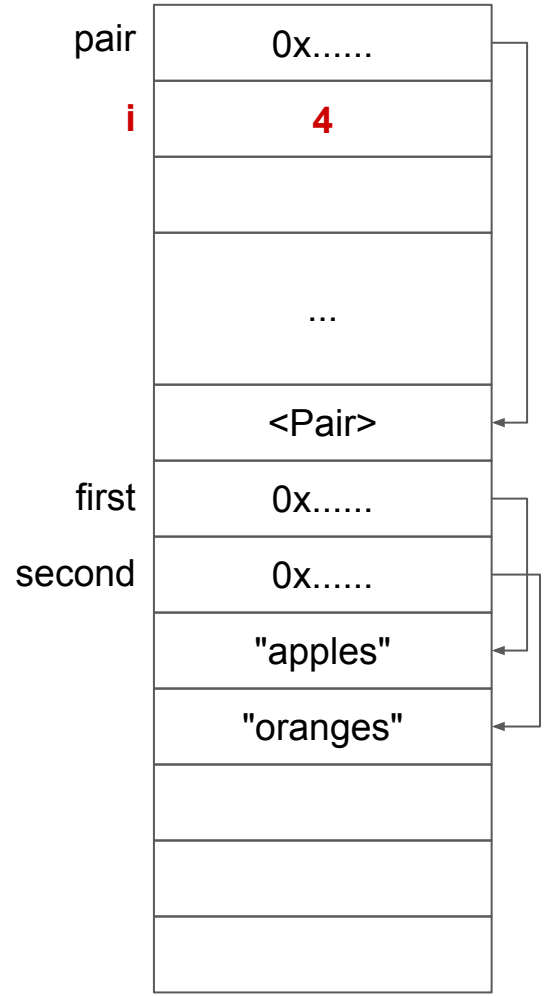
Special case: you don't need to use new with Strings. Even though they are objects!

```
Pair pair = new Pair("apples","oranges");
int i = 4;
double[] d = new double[] { 1.0, 2.0 };
```



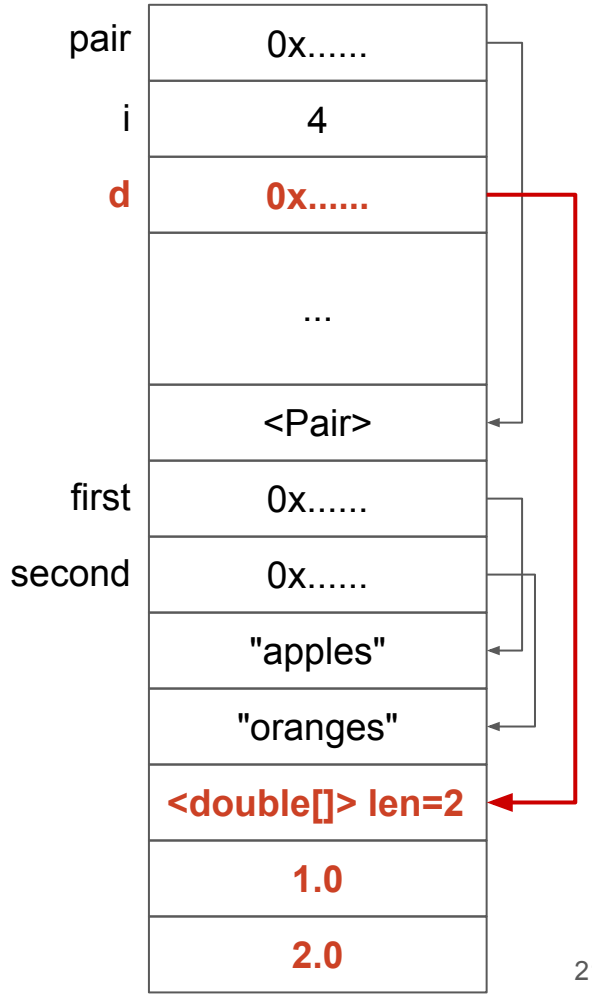
you don't use new
with primitive types
(they are not objects)

```
Pair pair = new Pair("apples","oranges");
int i = 4;
double[] d = new double[] { 1.0, 2.0 };
```



```
Pair pair = new Pair("apples","oranges");
int i = 4;
double[] d = new double[] { 1.0, 2.0 };
```

You do use new with arrays....because they are objects.



Immutability

https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html lays out a strategy for converting mutable classes into immutable classes. Most are fairly intuitive, but **what is the purpose of preventing classes from being extended** or the methods from being overridden? If one overrides a method in a subclass, why would it have an effect on the mutability or immutability of objects of the superclass?

```java
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)", first, second);
 }
}
```

```java
public class Main {

 public static void main(String[] args) {
   Pair pair = new Pair("apples","oranges");
   System.out.println(pair); // prints (apples.oranges)
   // Nothing I can do to change pair
   System.out.println(pair); // prints (apples.oranges)
 }
}
```

```java
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)", first, second);
 }
}
```

```java
public class Main {

 public static void main(String[] args) {
   Pair pair = new Pair("apples","oranges");
   System.out.println(pair); // prints (apples.oranges)
   // Nothing I can do to change pair
   System.out.println(pair); // prints (apples.oranges)
 }
}
```

```java
package uk.ac.cam.acr31.test;

public class MutablePair extends Pair {

 private Object first;
 private Object second;

 public MutablePair(Object first, Object second) {
   super(first, second);
   this.first = first;
   this.second = second;
 }

 public void first(Object o) {
   first = o;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)", first, second);
 }
}
```

MutablePair is a subtype of Pair. i.e. everywhere I can use Pair I can use a MutablePair

```
package uk.ac.cam.acr31.test;

public class MutablePair extends Pair {

 private Object first;
 private Object second;

 public MutablePair(Object first, Object second) {
   super(first, second);
   this.first = first;
   this.second = second;
 }

 public void first(Object o) {
   first = o;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)", first, second);
 }
}
```

It has its own copy of the first and second values.
They *shadow* the fields of the superclass.

```
package uk.ac.cam.acr31.test;

public class MutablePair extends Pair {

 private Object first;
 private Object second;

 public MutablePair(Object first, Object second) {
   super(first, second);
   this.first = first;
   this.second = second;
 }

 public void first(Object o) {
   first = o;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)", first, second);
 }
}
```

super(first, second) specifies how to construct the superclass when creating an instance.

```
package uk.ac.cam.acr31.test;

public class MutablePair extends Pair {

 private Object first;
 private Object second;

 public MutablePair(Object first, Object second) {
   super(first, second);
   this.first = first;
   this.second = second;
 }

 public void first(Object o) {
   first = o;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)", first, second);
 }
}
```

Change the value stored in the first position of this object.

```
package uk.ac.cam.acr31.test;

public class MutablePair extends Pair {

 private Object first;
 private Object second;

 public MutablePair(Object first, Object second) {
   super(first, second);
   this.first = first;
   this.second = second;
 }

 public void first(Object o) {
   first = o;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)", first, second);
 }
}
```

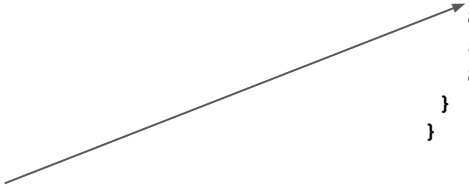
Change the value stored in the first position of this object. WHAAAAAT?

```
package uk.ac.cam.acr31.test;

public class MutablePair extends Pair {

 private Object first;
 private Object second;

 public MutablePair(Object first, Object second) {
   super(first, second);
   this.first = first;
   this.second = second;
 }

 public void first(Object o) {
   first = o;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)", first, second);
 }
}
```

```java
package uk.ac.cam.acr31.test;


public class Main {

 public static void main(String[] args) {
   Pair pair = new Pair("apples","oranges");
   System.out.println(pair); // prints (apples.oranges)
   // Nothing I can do to change pair
   System.out.println(pair); // prints (apples.oranges)
 }
}
```

```java
package uk.ac.cam.acr31.test;

public class Main {

 public static void main(String[] args) {
   MutablePair sneaky = new MutablePair("apples","oranges");
   Pair pair = sneaky;
   System.out.println(pair);  // prints (apples.oranges)
   sneaky.first("CHANGED!");
   System.out.println(pair);  // prints (CHANGED!,oranges)
 }
}
```

```
package uk.ac.cam.acr31.test;

public class Main {

 public static void main(String[] args) {
   MutablePair sneaky = new MutablePair("apples","oranges");
   Pair pair = sneaky;
   System.out.println(pair);  // prints (apples.oranges)
   sneaky.first("CHANGED!");
   System.out.println(pair);  // prints (CHANGED!,oranges)
 }
}
```



This is the important bit. Here the programmer thinks that they have a Pair which is immutable.

```
package uk.ac.cam.acr31.test;

public final class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 @Override
 public String toString() {
   return String.format("(%s,%s)", first, second);
 }
}
```

'final' here means that you may not extend this class

Generics

Demo of how Java generics is used. It would be best if it is something similar to what we have to do for the exercise Classic collection part 2.

How exactly does the compiler handle generics? Specifically, in what order does it execute type checks and type erasure? Does it do a type check first before type erasure? Otherwise how will it know that you can't put an Integer into List with only the raw type List?

More on generic class & type please.

```
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 public Object first() {
   return first;
 }

 public Object second() {
   return second;
 }
}


public class Main {

 public static void main(String[] args) {
   Pair pair = new Pair("apples", 1);
   String first = (String) pair.first();
 }
}
```

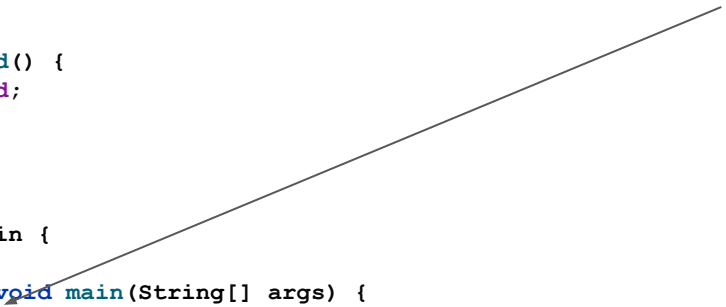
```java
package uk.ac.cam.acr31.test;

public class Pair<F, S> {

 private final F first;
 private final S second;

 public Pair(F first, S second) {
   this.first = first;
   this.second = second;
 }

 public F first() {
   return first;
 }

 public S second() {
   return second;
 }
}


public class Main {

 public static void main(String[] args) {
   Pair<String, Integer> pair = new Pair<>("apples", 1);
   String first = pair.first();
 }
}
```

```java
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 public Object first() {
   return first;
 }

 public Object second() {
   return second;
 }
}


public class Main {

 public static void main(String[] args) {
   Pair pair = new Pair("apples", 1);
   String first = (String) pair.first();
 }
}
```

```
package uk.ac.cam.acr31.test;

public class Pair<F, S> {

 private final F first;
 private final S second;

 public Pair(F first, S second) {
   this.first = first;
   this.second = second;
 }

 public F first() {
   return first;
 }

 public S second() {
   return second;
 }
}


public class Main {

 public static void main(String[] args) {
   Pair<String, Integer> pair = new Pair<>("apples", 1);
   String first = pair.first();
 }
}
```

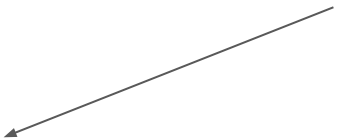
Type variables: whenever you use F and S in instance members of the class they refer to some concrete type provided at compile time

```
package uk.ac.cam.acr31.test;

public class Pair<F, S> {

 private final F first;
 private final S second;

 public Pair(F first, S second) {
   this.first = first;
   this.second = second;
 }

 public F first() {
   return first;
 }

 public S second() {
   return second;
 }
}


public class Main {

 public static void main(String[] args) {
   Pair<String, Integer> pair = new Pair<>("apples", 1);
   String first = pair.first();
 }
}
```



Specify the types you want to use when you declare the variable

```
package uk.ac.cam.acr31.test;

public class Pair<F, S> {

 private final F first;
 private final S second;

 public Pair(F first, S second) {
   this.first = first;
   this.second = second;
 }

 public F first() {
   return first;
 }

 public S second() {
   return second;
 }
}


public class Main {

 public static void main(String[] args) {
   Pair<String, Integer> pair = new Pair<>("apples", 1);
   String first = pair.first();
 }
}
```



Empty angle brackets means: please infer the types for me. You could write <String,Integer> here if you like.

```
package uk.ac.cam.acr31.test;

public class Pair<F, S> {

 private final F first;
 private final S second;

 public Pair(F first, S second) {
   this.first = first;
   this.second = second;
 }

 public F first() {
   return first;
 }

 public S second() {
   return second;
 }
}
```

Compiled separately

```
public class Main {

 public static void main(String[] args) {
   Pair<String, Integer> pair = new Pair<>("apples", 1);
   String first = pair.first();
 }
}
```

```java
package uk.ac.cam.acr31.test;

public class Pair<F, S> {

 private final F first;
 private final S second;

 public Pair(F first, S second) {
   this.first = first;
   this.second = second;
 }

 public F first() {
   return first;
 }

 public S second() {
   return second;
 }
}


public class Main {

 public static void main(String[] args) {
   Pair<String, Integer> pair = new Pair<>("apples", 1);
   String first = pair.first();
 }
}
```

```java
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 public Object first() {
   return first;
 }

 public Object second() {
   return second;
 }
}


public class Main {

 public static void main(String[] args) {
   Pair pair = new Pair("apples", 1);
   String first = (String) pair.first();
 }
}
```

```java
package uk.ac.cam.acr31.test;

public class Pair<F, S> {

 private final F first;
 private final S second;

 public Pair(F first, S second) {
   this.first = first;
   this.second = second;
 }

 public F first() {
   return first;
 }

 public S second() {
   return second;
 }
}
```

```java
public class Main {

 public static void main(String[] args) {
   Pair<String, Integer> pair = new Pair<>("apples", 1);
   String first = pair.first();
 }
}
```

```java
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 public Object first() {
   return first;
 }

 public Object second() {
   return second;
 }
}
```

```java
public class Main {

 public static void main(String[] args) {
   Pair pair = new Pair("apples", 1);
   String first = (String) pair.first();
 }
}
```

```java
package uk.ac.cam.acr31.test;

public class Pair<F, S> {

 private final F first;
 private final S second;

 public Pair(F first, S second) {
   this.first = first;
   this.second = second;
 }

 public F first() {
   return first;
 }

 public S second() {
   return second;
 }
}
```

```java
public class Main {

 public static void main(String[] args) {
   Pair<String, Integer> pair = new Pair<>("apples", 1);
   String first = pair.first();
 }
}
```

```java
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 public Object first() {
   return first;
 }

 public Object second() {
   return second;
 }
}
```

```java
public class Main {

 public static void main(String[] args) {
   Pair pair = new Pair("apples", 1);
   String first = (String) pair.first();
 }
}
```

Type Erasure!

```
package uk.ac.cam.acr31.test;

public class Pair<F, S> {

 private final F first;
 private final S second;

 public Pair(F first, S second) {
   this.first = first;
   this.second = second;
 }

 public F first() {
   return first;
 }

 public S second() {
   return second;
 }
}
```

```
public class Main {

 public static void main(String[] args) {
   Pair<String, Integer> pair = new Pair<>("apples", 1);
   String first = pair.first();
 }
}
```

```
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 public Object first() {
   return first;
 }

 public Object second() {
   return second;
 }
}
```

```
public class Main {

 public static void main(String[] args) {
   Pair pair = new Pair("apples", 1);
   String first = (String) pair.first();
 }
}
```

```
package uk.ac.cam.acr31.test;

public class Pair<F, S> {

 private final F first;
 private final S second;

 public Pair(F first, S second) {
   this.first = first;
   this.second = second;
 }

 public F first() {
   return first;
 }

 public S second() {
   return second;
 }
}
```

```
package uk.ac.cam.acr31.test;

public class Pair {

 private final Object first;
 private final Object second;

 public Pair(Object first, Object second) {
   this.first = first;
   this.second = second;
 }

 public Object first() {
   return first;
 }

 public Object second() {
   return second;
 }
}
```

```
public class Main {

 public static void main(String[] args) {
   Pair<String, Integer> pair = new Pair<>("apples", 1);
   String first = pair.first();
 }
}
```

```
public class Main {

 public static void main(String[] args) {
   Pair pair = new Pair("apples", 1);
   String first = (String) pair.first();
 }
}
```

Wildcards

List

List<Number>

List<?>

List<? extends Number>

List<? super Number>

Wildcards

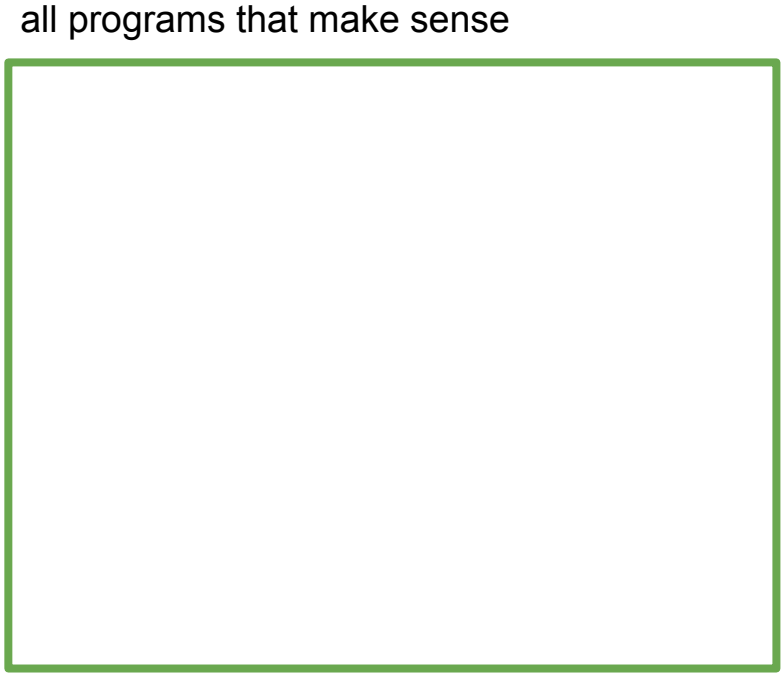
List

List<Number>

List<?>

List<? extends Number>

List<? super Number>



Wildcards

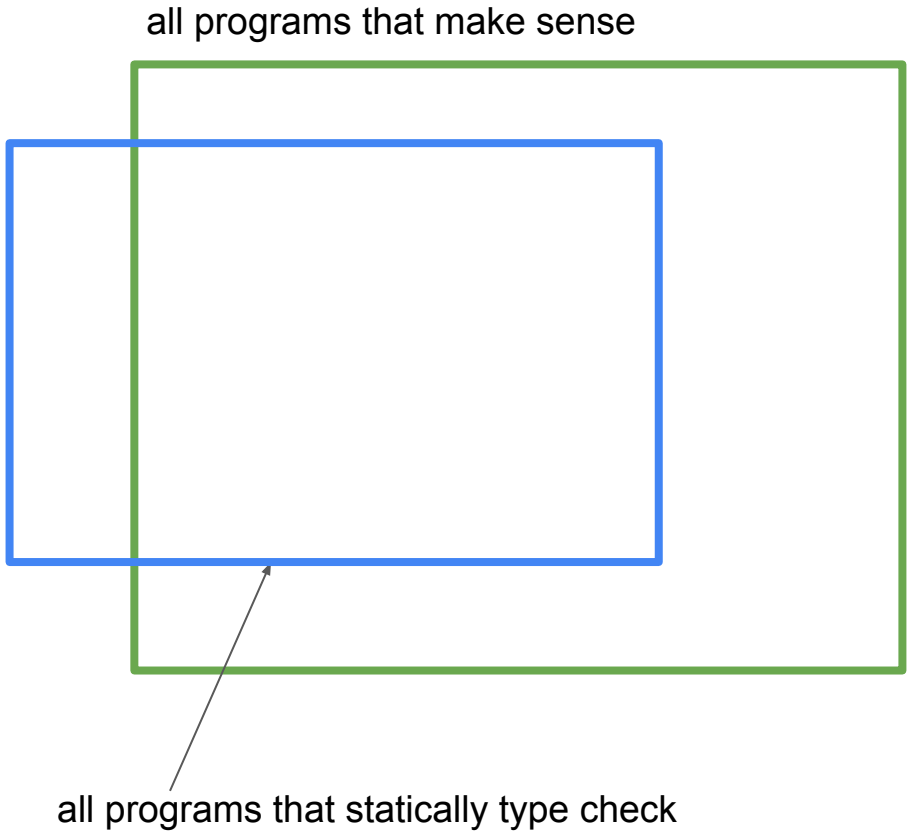
List

List<Number>

List<?>

List<? extends Number>

List<? super Number>



Wildcards

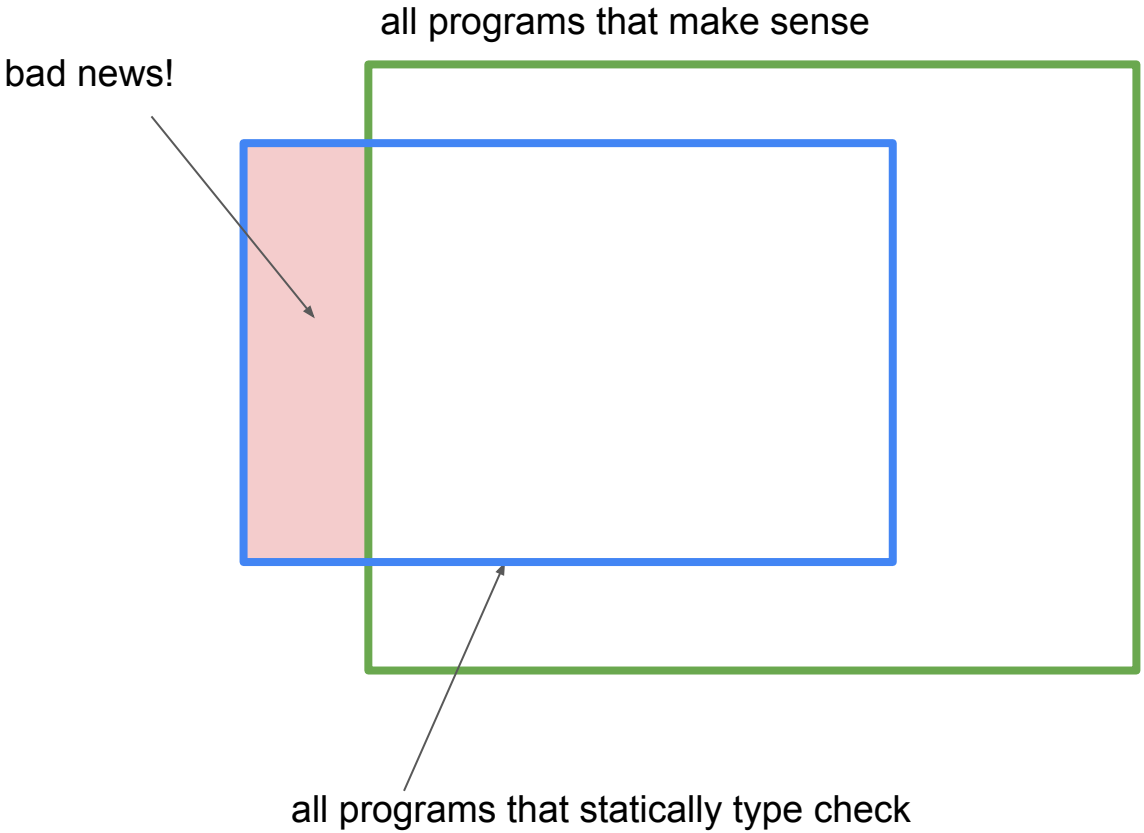
List

List<Number>

List<?>

List<? extends Number>

List<? super Number>



Wildcards

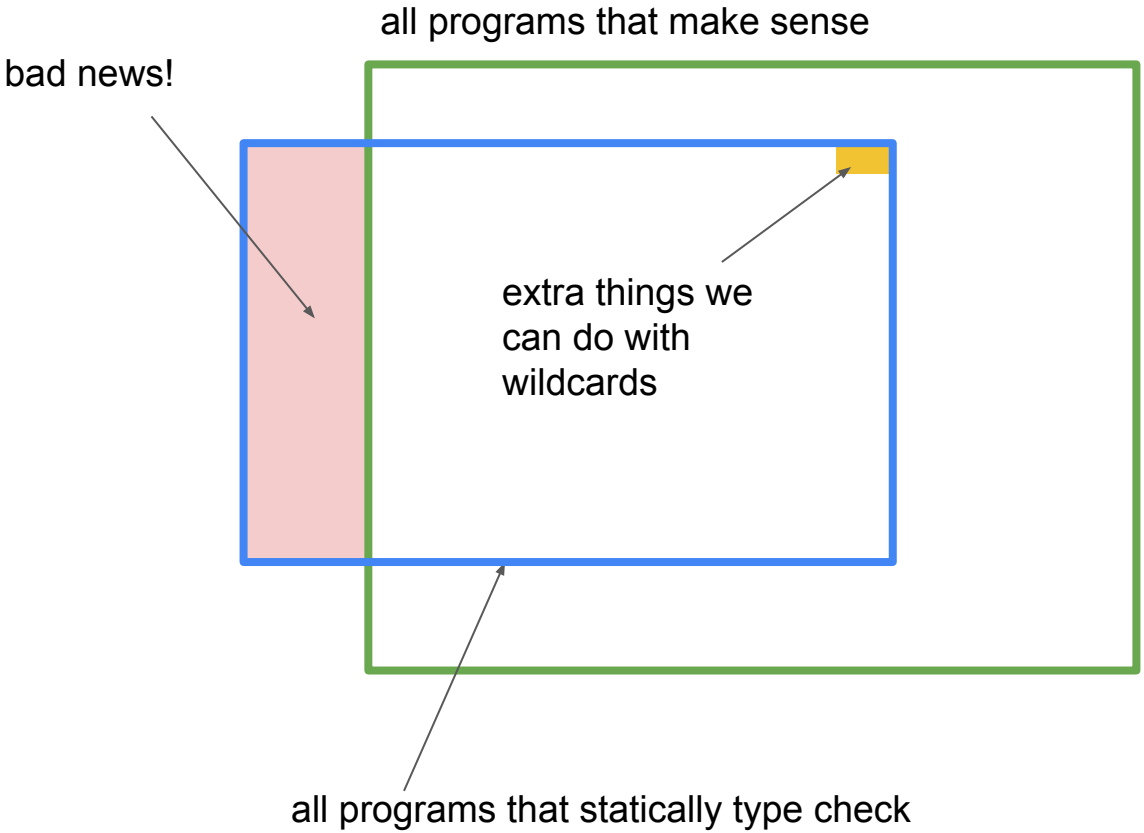
List

List<Number>

List<?>

List<? extends Number>

List<? super Number>



```
static void processElements(List<Object> elements){
 for(Object o : elements){
   System.out.println(o);
 }
}

List<Integer> l = List.of(1,2,3,4);
processElements(l);
```

Does not compile!

```
static void processElements(List<Object> elements){
 for(Object o : elements){
   System.out.println(o);
 }
}

List<Integer> l = List.of(1,2,3,4);
processElements(l);
```

List<Integer> is not a subtype of List<Object> so you can't pass it to the method

Wildcard '?' means any type

```
static void processElements(List<?> elements){
 for(Object o : elements){
   System.out.println(o);
 }
}

List<Integer> l = List.of(1,2,3,4);
processElements(l);
```

When we read instances of 'any type' from the list we can assign them to Object since all generic parameters must be Objects

List<Integer> is assignable to List<?> so this works

Does not compile!

Although we can read Objects from the list we can't write them back - its a list of 'any type' - could be Strings or Integers or Ducks

```
static void processElements(List<?> elements){
 for(Object o : elements){
   System.out.println(o);
 }
 elements.set(0,5);
}

List<Integer> l = List.of(1,2,3,4);
processElements(l);
```

Compile error:
expected capture of ?
got int

```java
class Shape {
 String getName() {
   return "Shape";
 }
}

class Square extends Shape {
 @Override
 String getName() {
   return "Square";
 }
}

class Triangle extends Shape {
 @Override
 String getName() {
   return "Triangle";
 }
}

public class Main {

 static void processElements(List<Shape> elements) {
   for (Shape o : elements) {
     System.out.println(o.getName());
   }
 }

 public static void main(String[] args) {
   List<Square> l = List.of(new Square(), new Square());
   processElements(l);
 }
}
```

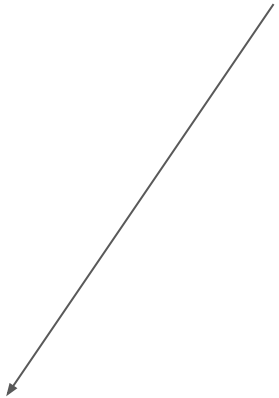
```
class Shape {
 String getName() {
   return "Shape";
 }
}

class Square extends Shape {
 @Override
 String getName() {
   return "Square";
 }
}

class Triangle extends Shape {
 @Override
 String getName() {
   return "Triangle";
 }
}

public class Main {

 static void processElements(List<Shape> elements) {
   for (Shape o : elements) {
     System.out.println(o.getName());
   }
 }

 public static void main(String[] args) {
   List<Square> l = List.of(new Square(), new Square());
   processElements(l);
 }
}
```

Does not compile!

List<Square> is not a subtype of List<Shape>

```
class Shape {
 String getName() {
   return "Shape";
 }
}

class Square extends Shape {
 @Override
 String getName() {
   return "Square";
 }
}

class Triangle extends Shape {
 @Override
 String getName() {
   return "Triangle";
 }
}

public class Main {

 static void processElements(List<? extends Shape> elements) {
   for (Shape o : elements) {
     System.out.println(o.getName());
   }
 }

 public static void main(String[] args) {
   List<Square> l = List.of(new Square(), new Square());
   processElements(l);
 }
}
```

Upper-bounded wildcard



Alternatively.....

```
class Shape {
 String getName() {
   return "Shape";
 }
}

class Square extends Shape {
 @Override
 String getName() {
   return "Square";
 }
}

class Triangle extends Shape {
 @Override
 String getName() {
   return "Triangle";
 }
}

public class Main {

 static void processElements(List<Shape> elements) {
   for (Shape o : elements) {
     System.out.println(o.getName());
   }
 }

 public static void main(String[] args) {
   List<Shape> l = List.<Shape>of(new Square(), new Square());
   processElements(l);
 }
}
```

Can work round it by having a List<Shape> filled with Square objects

Note the odd syntax for generic type on a method

```
static void processElements(List<Number> elements){
 elements.add(1);
 elements.add(1.0);
 elements.add(1L);
}

List<Object> l = new ArrayList<Object>();
processElements(l);
```

Does not compile!

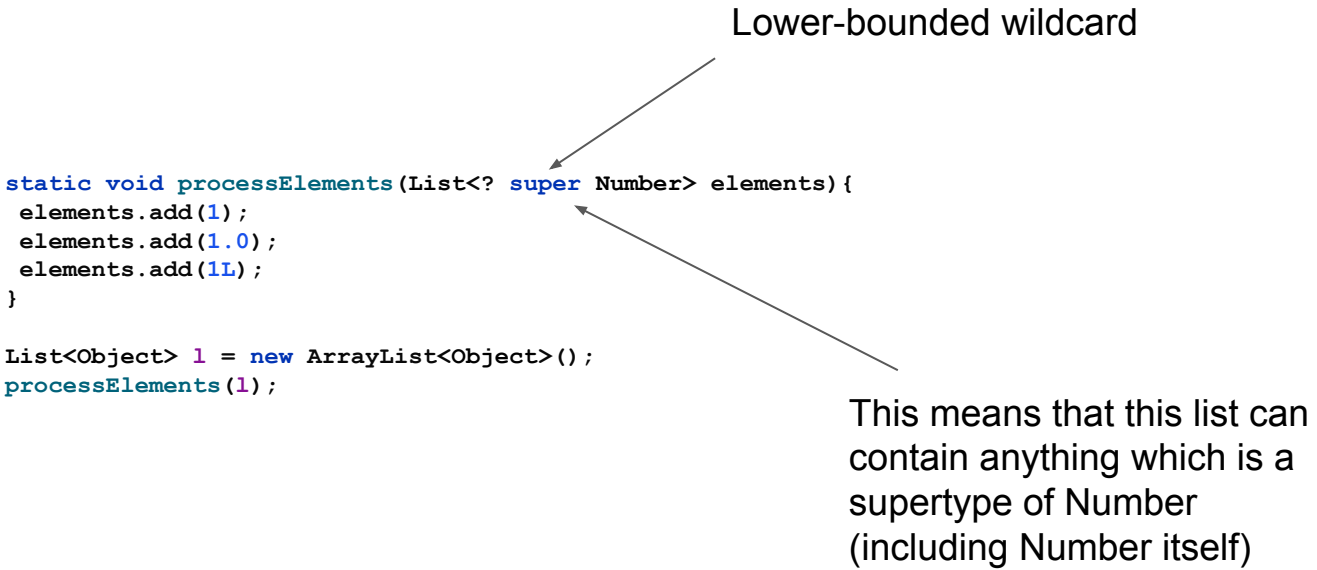
```
static void processElements(List<Number> elements){
 elements.add(1);
 elements.add(1.0);
 elements.add(1L);
}

List<Object> l = new ArrayList<Object>();
processElements(l);
```

List<Object> is not a subtype of List<Number> ...duh!

```
static void processElements(List<Number> elements){
 elements.add(1);
 elements.add(1.0);
 elements.add(1L);
}

List<Number> l = new ArrayList<Number>();
processElements(l);
```

These would both be fine

```
static void processElements(List<Object> elements){
 elements.add(1);
 elements.add(1.0);
 elements.add(1L);
}

List<Object> l = new ArrayList<Object>();
processElements(l);
```


Lower-bounded wildcard
static void processElements(List<? super Number> elements){
elements.add(1);
elements.add(1.0);
elements.add(1L);
}
List<Object> l = new ArrayList<Object>();
processElements(l);
This means that this list can contain anything which is a supertype of Number (including Number itself)

Bounds can be used when declaring a generic class too

```
public class SortingPair<F extends Comparable<F>> {

 private final F first;
 private final F second;

 public SortingPair(F first, F second) {
   int compare = first.compareTo(second);
   this.first = compare > 0 ? first : second;
   this.second = compare > 0 ? second : first;
 }

 public F first() {
   return first;
 }

 public F second() {
   return second;
 }
}
```

We know F extends Comparable<F> so we know that it provides a compareTo method

Summary thoughts on generics

Keyword: 'type erasure'

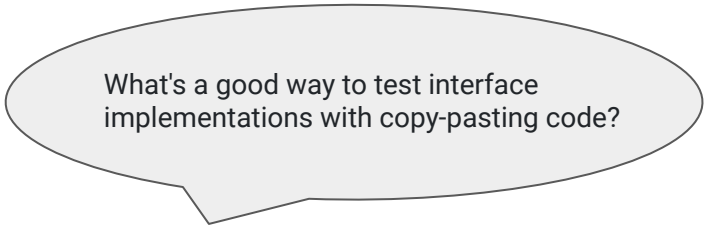
If you can translate the generic one to erased one then you can spot problems

Wildcards and bounds allow you to be more expressive with the type system

Can also use bounds to make assumptions on the class you are instantiated with

Further thinking: an array is like a generic container. T[] is like Array<T>. Watch out for similarities and differences between them...

Testing


What's a good way to test interface implementations with copy-pasting code?

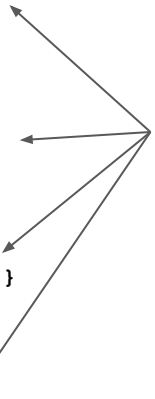
```
package uk.ac.cam.acr31.test;

interface Doubler {
 int apply(int x);
}

class MultDoubler implements Doubler {
 @Override
 public int apply(int x) { return x * 2; }
}

class AddDoubler implements Doubler {
 @Override
 public int apply(int x) { return x + x; }
}

class ShiftDoubler implements Doubler {
 @Override
 public int apply(int x) { return x << 1; }
}

class NotADoubler implements Doubler {
 @Override
 public int apply(int x) { return 0; }
}
```

```
package uk.ac.cam.acr31.test;

interface Doubler {
 int apply(int x);
}

class MultDoubler implements Doubler {
 @Override
 public int apply(int x) { return x * 2; }
}

class AddDoubler implements Doubler {
 @Override
 public int apply(int x) { return x + x; }
}

class ShiftDoubler implements Doubler {
 @Override
 public int apply(int x) { return x << 1; }
}

class NotADoubler implements Doubler {
 @Override
 public int apply(int x) { return 0; }
}
```

An interface called Doubler which requires only a single method

(In future you'll learn that this is a 'functional interface')

```
package uk.ac.cam.acr31.test;

interface Doubler {
 int apply(int x);
}

class MultDoubler implements Doubler {
 @Override
 public int apply(int x) { return x * 2; }
}

class AddDoubler implements Doubler {
 @Override
 public int apply(int x) { return x + x; }
}

class ShiftDoubler implements Doubler {
 @Override
 public int apply(int x) { return x << 1; }
}

class NotADoubler implements Doubler {
 @Override
 public int apply(int x) { return 0; }
}
```



Various implementations which double in different ways

Not examinable

```
package uk.ac.cam.acr31.test;

import ...

@RunWith(Parameterized.class)
public class DoublerTest {

 @Parameters(name="{0}")
 public static Collection<Object[]> data() {
   return List.of(
   new Object[] {"MultDoubler", new MultDoubler()},
   new Object[] {"AddDoubler", new AddDoubler()},
   new Object[] {"ShiftDoubler", new ShiftDoubler()},
   new Object[] {"NotADoubler", new NotADoubler()});
 }

 private final Doubler doubler;

 public DoublerTest(String testName, Doubler doubler) {
   this.doubler = doubler;
 }

 @Test
 public void doubler_doublesInput() {
   // ARRANGE
   // ACT
   int result = doubler.apply(2);

   // ASSERT
   assertThat(result).isEqualTo(4);
 }
}
```

Not examinable

```java
package uk.ac.cam.acr31.test;

import ...

@RunWith(Parameterized.class)
public class DoublerTest {

 @Parameters(name="{0}")
 public static Collection<Object[]> data() {
   return List.of(
   new Object[] {"MultDoubler", new MultDoubler()},
   new Object[] {"AddDoubler", new AddDoubler()},
   new Object[] {"ShiftDoubler", new ShiftDoubler()},
   new Object[] {"NotADoubler", new NotADoubler()});
 }

 private final Doubler doubler;

 public DoublerTest(String testName, Doubler doubler) {
   this.doubler = doubler;
 }

 @Test
 public void doubler_doublesInput() {
   // ARRANGE
   // ACT
   int result = doubler.apply(2);

   // ASSERT
   assertThat(result).isEqualTo(4);
 }
}
```

Specify a strategy for running the tests in this class - we need 'Parameterized'

Not examinable

```
package uk.ac.cam.acr31.test;

import ...

@RunWith(Parameterized.class)
public class DoublerTest {

 @Parameters(name="{0}")
 public static Collection<Object[]> data() {
   return List.of(
   new Object[] {"MultDoubler", new MultDoubler()},
   new Object[] {"AddDoubler", new AddDoubler()},
   new Object[] {"ShiftDoubler", new ShiftDoubler()},
   new Object[] {"NotADoubler", new NotADoubler()});
 }

 private final Doubler doubler;

 public DoublerTest(String testName, Doubler doubler) {
   this.doubler = doubler;
 }

 @Test
 public void doubler_doublesInput() {
   // ARRANGE
   // ACT
   int result = doubler.apply(2);

   // ASSERT
   assertThat(result).isEqualTo(4);
 }
}
```

You need a special method (annotated with Parameters) to tell JUnit what parameters to use. It returns a collection of Object arrays.

Not examinable

```
package uk.ac.cam.acr31.test;

import ...

@RunWith(Parameterized.class)
public class DoublerTest {

 @Parameters(name="{0}")
 public static Collection<Object[]> data() {
   return List.of(
   new Object[] {"MultDoubler", new MultDoubler()},
   new Object[] {"AddDoubler", new AddDoubler()},
   new Object[] {"ShiftDoubler", new ShiftDoubler()},
   new Object[] {"NotADoubler", new NotADoubler()});
 }

 private final Doubler doubler;

 public DoublerTest(String testName, Doubler doubler) {
   this.doubler = doubler;
 }

 @Test
 public void doubler_doublesInput() {
   // ARRANGE
   // ACT
   int result = doubler.apply(2);

   // ASSERT
   assertThat(result).isEqualTo(4);
 }
}
```

You need a special method (annotated with Parameters) to tell JUnit what parameters to use. It returns a collection of Object arrays.

Each object array is unpacked and passed to the constructor for each instance of the test

Not examinable

You'll see other syntax for building Collection<Object[]> online ... blah blah blah

```
package uk.ac.cam.acr31.test;

import ...

@RunWith(Parameterized.class)
public class DoublerTest {

 @Parameters(name="{0}")
 public static Collection<Object[]> data() {
   return List.of(
    new Object[] {"MultDoubler", new MultDoubler()},
    new Object[] {"AddDoubler", new AddDoubler()},
    new Object[] {"ShiftDoubler", new ShiftDoubler()},
    new Object[] {"NotADoubler", new NotADoubler()});
 }

 private final Doubler doubler;

 public DoublerTest(String testName, Doubler doubler) {
   this.doubler = doubler;
 }

 @Test
 public void doubler_doublesInput() {
   // ARRANGE
   // ACT
   int result = doubler.apply(2);

   // ASSERT
   assertThat(result).isEqualTo(4);
 }
}
```

You need a special method (annotated with Parameters) to tell JUnit what parameters to use. It returns a collection of Object arrays.

Each object array is unpacked and passed to the constructor for each instance of the test

Not examinable

```
package uk.ac.cam.acr31.test;

import ...

@RunWith(Parameterized.class)
public class DoublerTest {

 @Parameters(name="{0}")
 public static Collection<Object[]> data() {
   return List.of(
   new Object[] {"MultDoubler", new MultDoubler()},
   new Object[] {"AddDoubler", new AddDoubler()},
   new Object[] {"ShiftDoubler", new ShiftDoubler()},
   new Object[] {"NotADoubler", new NotADoubler()});
 }

 private final Doubler doubler;

 public DoublerTest(String testName, Doubler doubler) {
   this.doubler = doubler;
 }

 @Test
 public void doubler_doublesInput() {
   // ARRANGE
   // ACT
   int result = doubler.apply(2);

   // ASSERT
   assertThat(result).isEqualTo(4);
 }
}
```

Little bit of magic that says: please name the test with the value of the first argument

End

Drop in session tomorrow - email me today if you want a slot

The department has increased the VM size that chime is running on - hopefully its a bit faster

Please submit questions for the final class on Wednesday 1st December.