```
public static <T extends Comparable<? super T>> void sort( @NotNull List<T> list)
```

any list of type T

our type T must be Comparable

```
<T> void sort2(List<T> list)
```

use it

declare

```
static class Person implements Comparable<Person>
```

```
<T extends Comparable<T>> void sort1(List<T> list)
```

=

T must implement Comparable<T>
T can be compared
T can be sorted

Comparator<A>

compare(a1, a2) → int

Comparable<A>

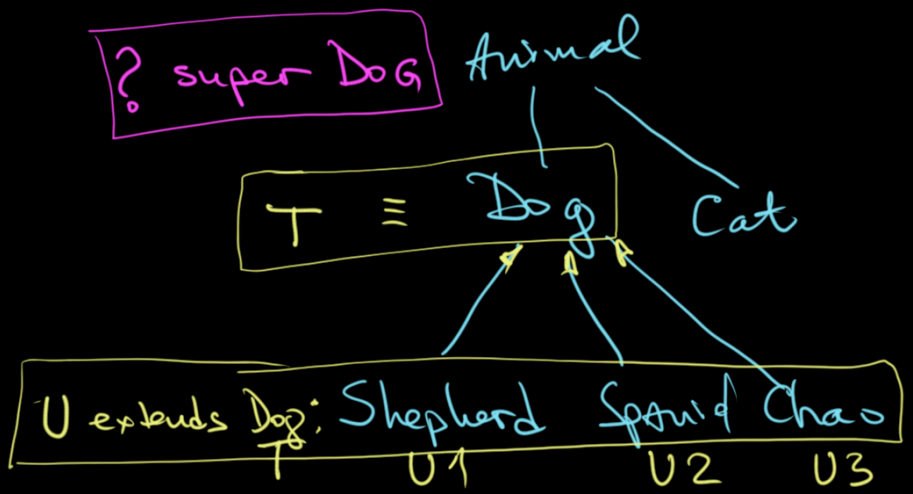
compareTo(that) → int

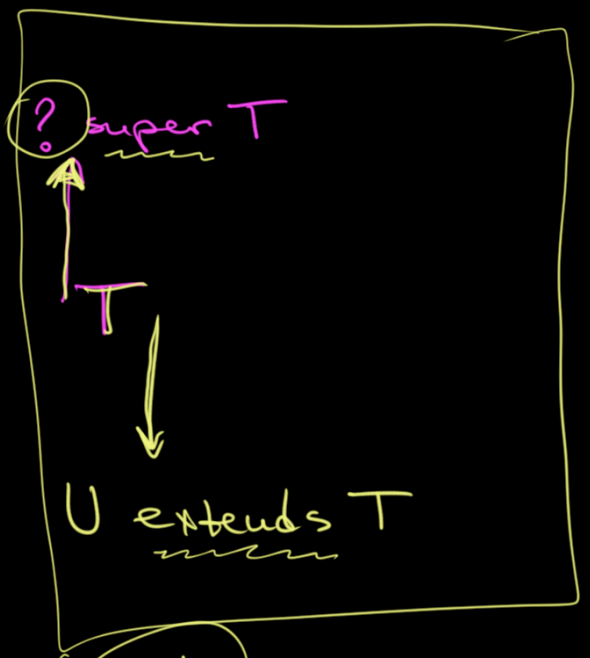
the first object is "this" from our context

```
<T extends Comparable<? super T>>
```

<T extends Dog> T makeDog(...)

Dog makeDog(name) {
return new Shepherd
return new Chao

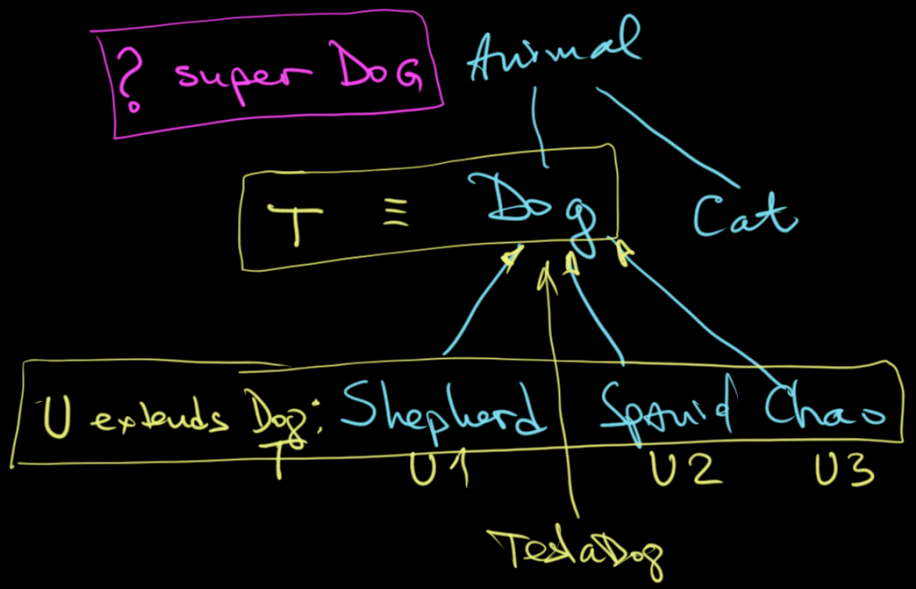




<A,B> A extends Something (B) super B => A

(T) U V W ...

A B C D E ...



Dog < Shepherd

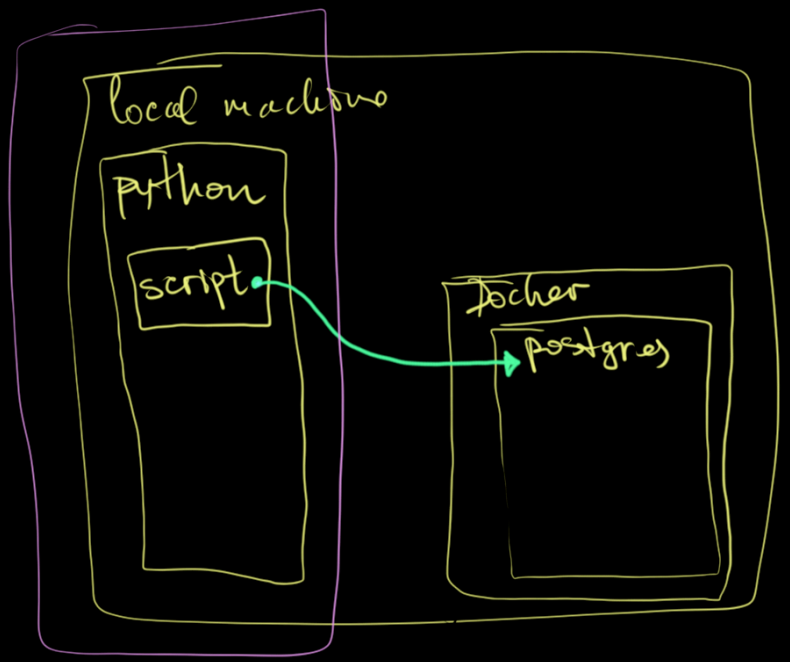
heal (Dog d)

do I know how to heal the Shepherd — No X

do I know how to heal the Animal — Yes V

Animal has less properties than Dog

We have



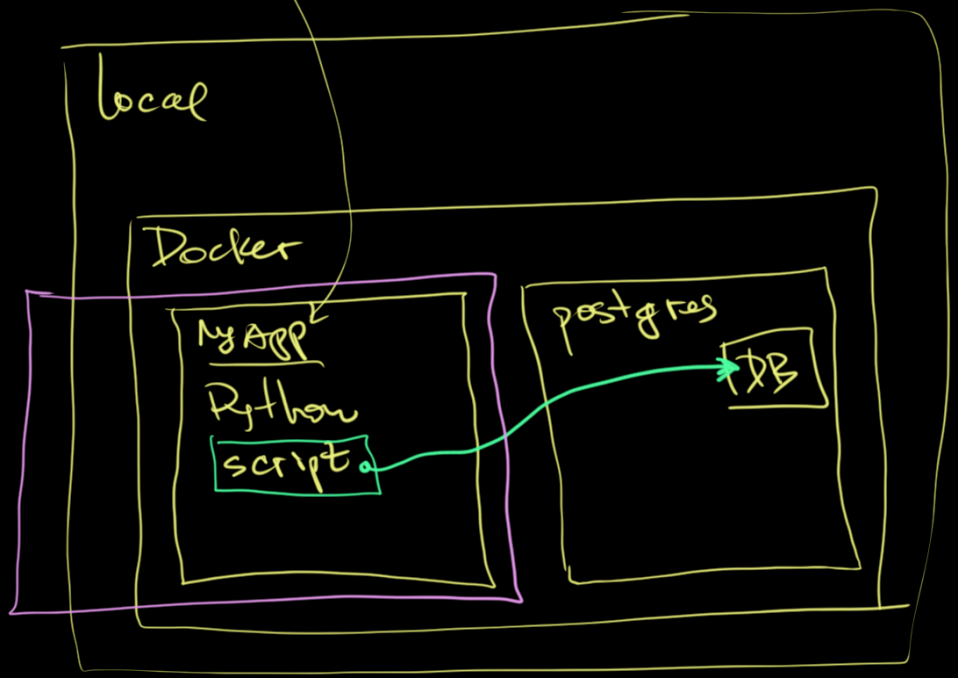
myApp ← docker container

1

Dockerfile

docker build

2



python + script

should not be on your machine

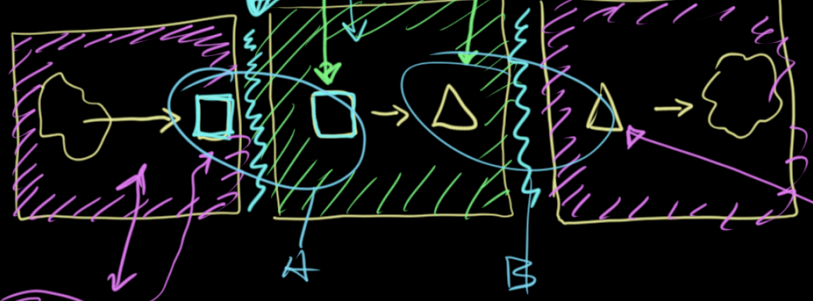
should be in the docker also

you need to share docker-compose.yml

I can run it

• to convert all the strings to UpperCase

```
public String convert (String origin) {
  return origin.toUpperCase();
}
```




```
public void app() {
  String line = read();
  String converted = convert(line);
  write(converted);
}
```

if we want read/write from different source => we need to rewrite

```
public String read() {
  return "test line";
}
```

```
public void write(String line) {
  // write to db
}
```

```
static public void app1(
    Supplier<String> howToRead,
    Consumer<String> howToWrite) {
  String line = howToRead.get();
  String converted = convert(line);
  howToWrite.accept(converted);
}

public static void main(String[] args) {
  app1(Convertor::read, Convertor::write);
}
```

it's still wired / limited
you can't compose it in a different way

we decoupled our app
any read / write function can be passed

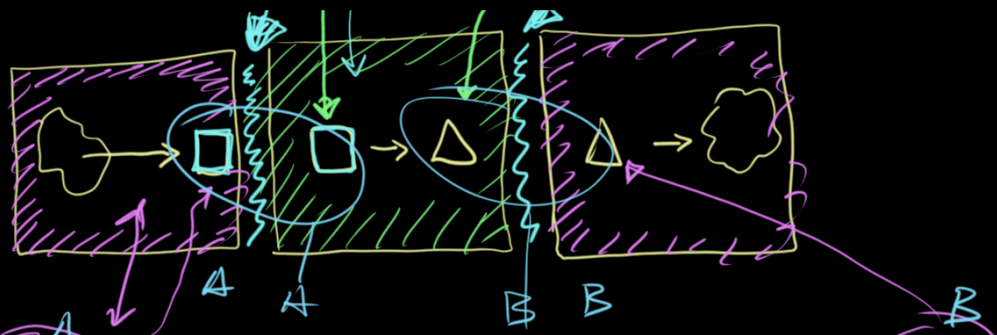
```
static public void app2(
    Supplier<String> howToRead,
    Function<String, String> core,
    Consumer<String> howToWrite) {
  String line = howToRead.get();
  String converted = core.apply(line);
  howToWrite.accept(converted);
}

public static void main1(String[] args) {
  app2(Convertor::read, Convertor::convert, Convertor::write);
}
```

it's still composed
the sequence (composition) is guaranteed

it's decoupled

you can pass different impl. here

```
public static void main2(String[] args) {
  app2(Convertor::read2, Convertor::convert, Convertor::write);
}
```




```java
public String read() {
  return "test line";
}
```

```java
public void write(String line) {
  // write to db
}
```

```java
static public <A, B> void app3(
    Supplier<A> howToRead,
    Function<A, B> core,
    Consumer<B> howToWrite) {
  A line = howToRead.get();
  B converted = core.apply(line);
  howToWrite.accept(converted);
}
```

we almost impossible to compose in a wrong way

```java
public static void main3(String[] args) {
  app3(Convertor::read2, Convertor::convert, Convertor::write);
}
```

```java
static public <A, B> void app3(
    Supplier<A> howToRead,
    Function<A, B> core,
    Consumer<B> howToWrite) {
  A line = howToRead.get();
  B converted = core.apply(line);
  howToWrite.accept(converted);
}
```

A = String

```java
static public Integer convertToInt (String origin) {
  return Integer.parseInt(origin);
}

public static void main4(String[] args) {
  app3(Convertor::read2, Convertor::convertToInt, Convertor::write);
}
```

B

A

String → Int

() → String

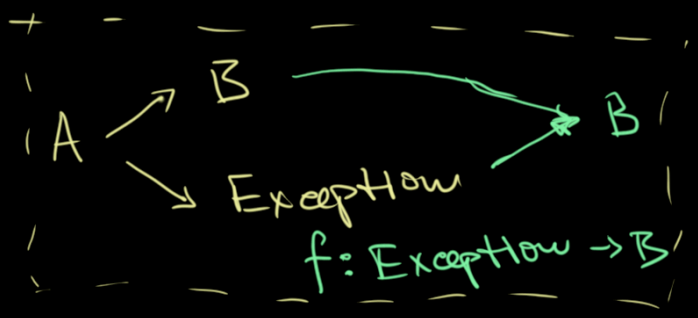
```java
static public Integer convertToInt (String origin) {
  return Integer.parseInt(origin);
}

static public void writeInt(Integer line) {
  // write to db
}

public static void main4(String[] args) {
  app3(Convertor::read2, Convertor::convertToInt, Convertor::writeInt);
}
```

$\frac{5}{0}$

function unsafe A => B | Exception

recover : Exception -> B

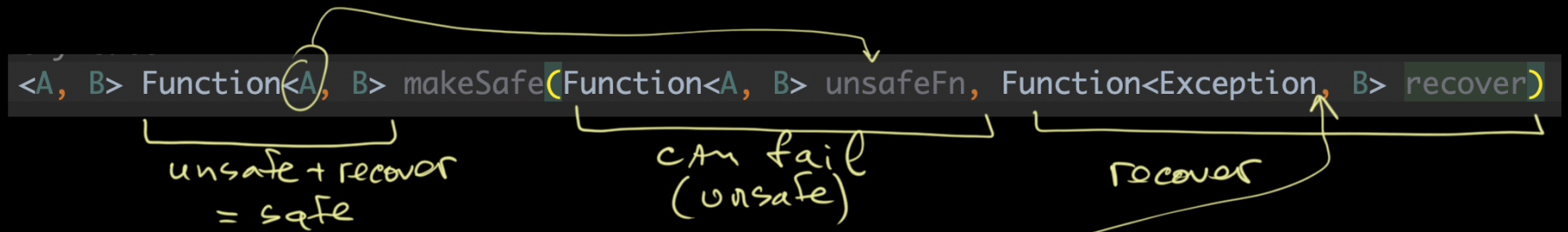
A => B safe



A -> C safe

ok: A -> B B -> C

error: A -> Ex Ex -> C




```
return (A a) -> {
  try {
    return unsafeFn.apply(a);
  } catch (Exception x) {
    return recover.apply(x);
  }
};
```