# Bridging Methods in Java

In this document I will explain the motivation behind briding method as well as when it will be used.

# The running example

Say we have the following Java program

```
class A<T> {
void fun(T t) {
System.out.println("A::fun");
}
}
```

Here, we have a class `A` with a type parameter `T`. `A` has a method `fun` which takes in `T` as a parameter and prints out "A::fun".

Now let's imagine we are going to extend a parameterized version of A.

```
class B extends A<String> {
void fun(String t) {
System.out.println("B::fun");
}
}
```

Here, we see that `T` has been assigned `String`.

Note that `fun` here has the parameter of type `String`, the same type `T` has been assigned too.

Instinctively, this seems to be an overriding.

Therefore, when we execute following code,

```
A a = new B();
```

```
a.fun("This is a best string you've seen so far");
```

we would expect "B::fun" to be printed, since it should use the implementation of the runtime type.

> If you are lost here, you should revise dynamic binding then come back

# How type erasure ruins things

Consider what `A` would look like after type erasure.

Before:

```
class A<T> {
void fun(T t) {
System.out.println("A::fun");
}
}
```

After:

```
class A {
void fun(Object t) {
System.out.println("A::fun");
}
}
```

> If you are lost here, you should revise type erasure

Now, notice that `A` only has a method of type `fun(Object)`, on top of this `B` **also inherits** this method from `A`.

If we are to run the same code as before

```
A a = new B();

a.fun("This is a best string you've seen so far");
```

Compilation step of dynamic binding would settle on `fun(Object)` since that is the only method that `A` has and that works for the argument type.

Therefore, during the runtime step of dynamic binding it would look for `fun(Object)` which is not implemented in `B` therefore, runs `A`'s implementation printing "A::fun"

Of course this breaks the user expectation that he/she/they have overidden the `A(T)` method.

Especially at the end of the day we would want to hide the fact that we are type erasing from the users perspective.

# How bridging method bridges

To fix this behaviour and to align it with user expectation, the compiler would inject *bridging methods*.

Essentially what it does is that for class `B` it would inject a method specifically for the erased version of `fun`.

Therefore from:

```
class B extends A<String> {

void fun(String t) {

System.out.println("B::fun");

}

}
```

we get

```
class B extends A<String> {

void fun(String t) {

System.out.println("B::fun");

}
```

```
// THIS IS INJECTED BY COMPILER!
void fun(Object t) {
this.fun((String) t);
}
}
```

Notice the cast of `String` to the argument `t`.

So even if we settle on the method `fun(Object)`, during runtime type we do find an implementation in `B` for it and then do another `fun` call, this time it would correctly invoke `B::fun(String)` and print out "B::fun".

# Fixing one problem opens another one

Because of this injected code, we no longer can create a method of the type of the erased version

So using our example, we cannot do this for `B`:

```
class B extends A<String> {
void fun(String t) {
System.out.println("B::fun");
}

void fun(Object o) {
System.out.println("Hi, I'm not allowed. I shouldn't exist :(");
}
}
```

because after injection, it would look like this

```
class B extends A<String> {
```

```
void fun(String t) {
System.out.println("B::fun");
}

void fun(Object o) {
System.out.println("Hi, I'm not allowed. I shouldn't exist :(");
}

// THIS IS INJECTED BY COMPILER!
void fun(Object t) {
this.fun((String) t);
}
}
```

Now if we were to try to invoke `fun(Object)` it would be ambiguous because `B` now contains 2 implementations for `fun(Object)`.

# Congratulations!

You now understand the idea behind bridging methods! ok bye