Object-Oriented Programming

CSCI-UA 0470-001
Class 17
Instructor: Randy Shepherd

Method Overriding In Java

Translator

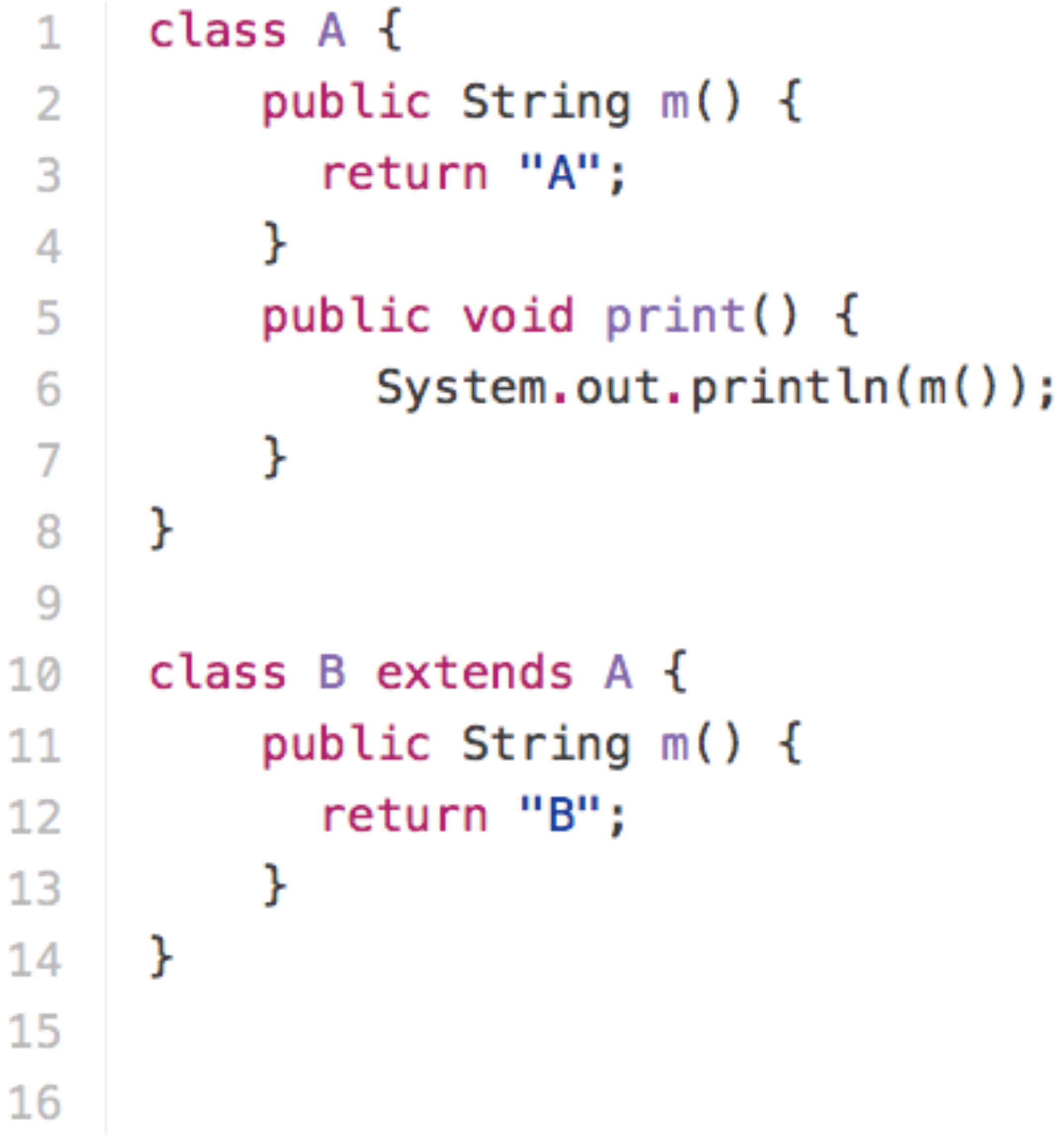
- In our translator we have supported method overriding via dynamic dispatch.
- However, there is some fine print with overriding we have yet to explore.

Method Overriding Review

- An instance method in a subclass with the *same signature and return type* as an instance method in the superclass ‘overrides’ the superclass's method.
- This language feature allows a class to inherit from a superclass whose behavior is "close enough" and then to modify behavior as needed.
- Seems simple enough, right?

Simple Example

- Is the method defined on line 11 legal Java?



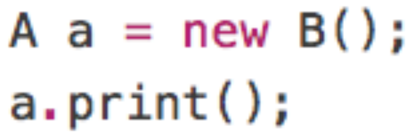
```
1   class A {
2       public String m() {
3         return "A";
4       }
5       public void print() {
6           System.out.println(m());
7       }
8   }
9
10  class B extends A {
11      public String m() {
12        return "B";
13      }
14  }
15
16
```

Simple Example

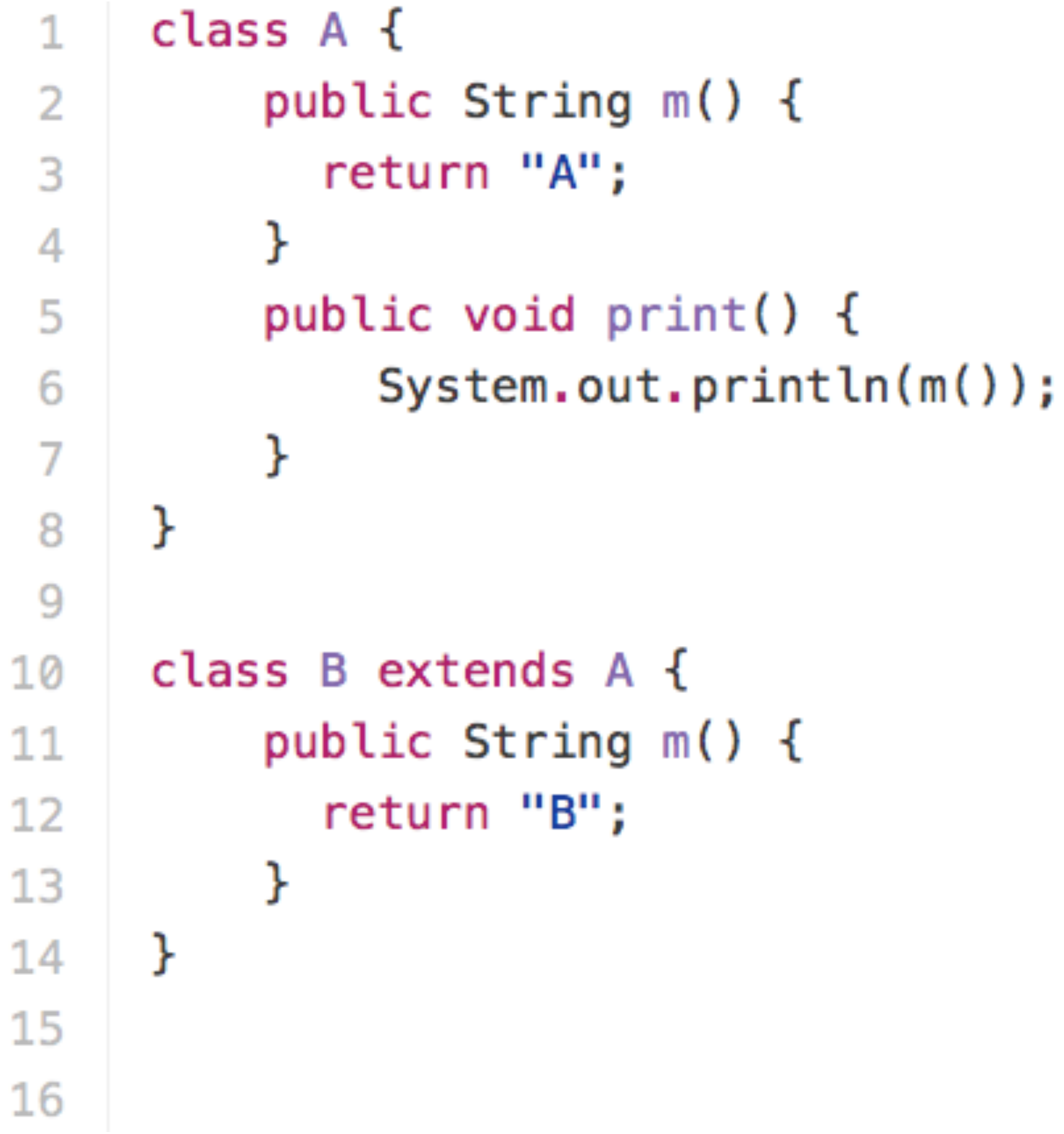
- Is the method defined on line 11 legal Java?

- Yes, this is a basic override.

- Ok, what is printed by this code?



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



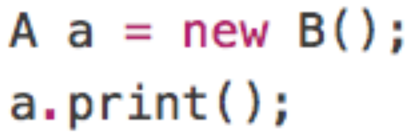
```
 1  class A {
 2      public String m() {
 3        return "A";
 4      }
 5      public void print() {
 6          System.out.println(m());
 7      }
 8  }
 9
10  class B extends A {
11      public String m() {
12        return "B";
13      }
14  }
15
16
```

Simple Example

- Is the method defined on line 11 legal Java?

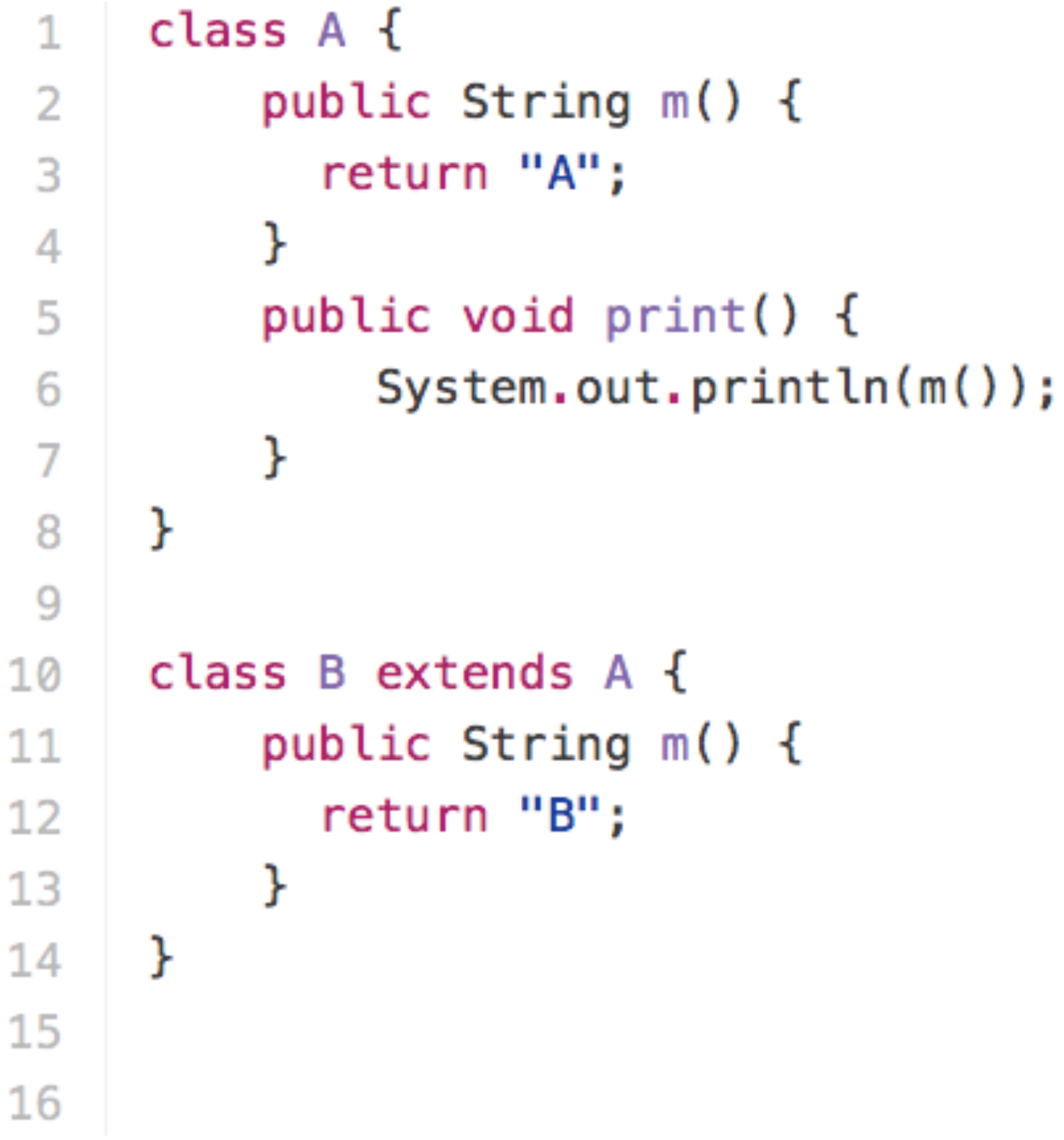
- Yes, this is a basic override.

- Ok, what is printed by this code?



```java
A a = new B();
a.print();
```

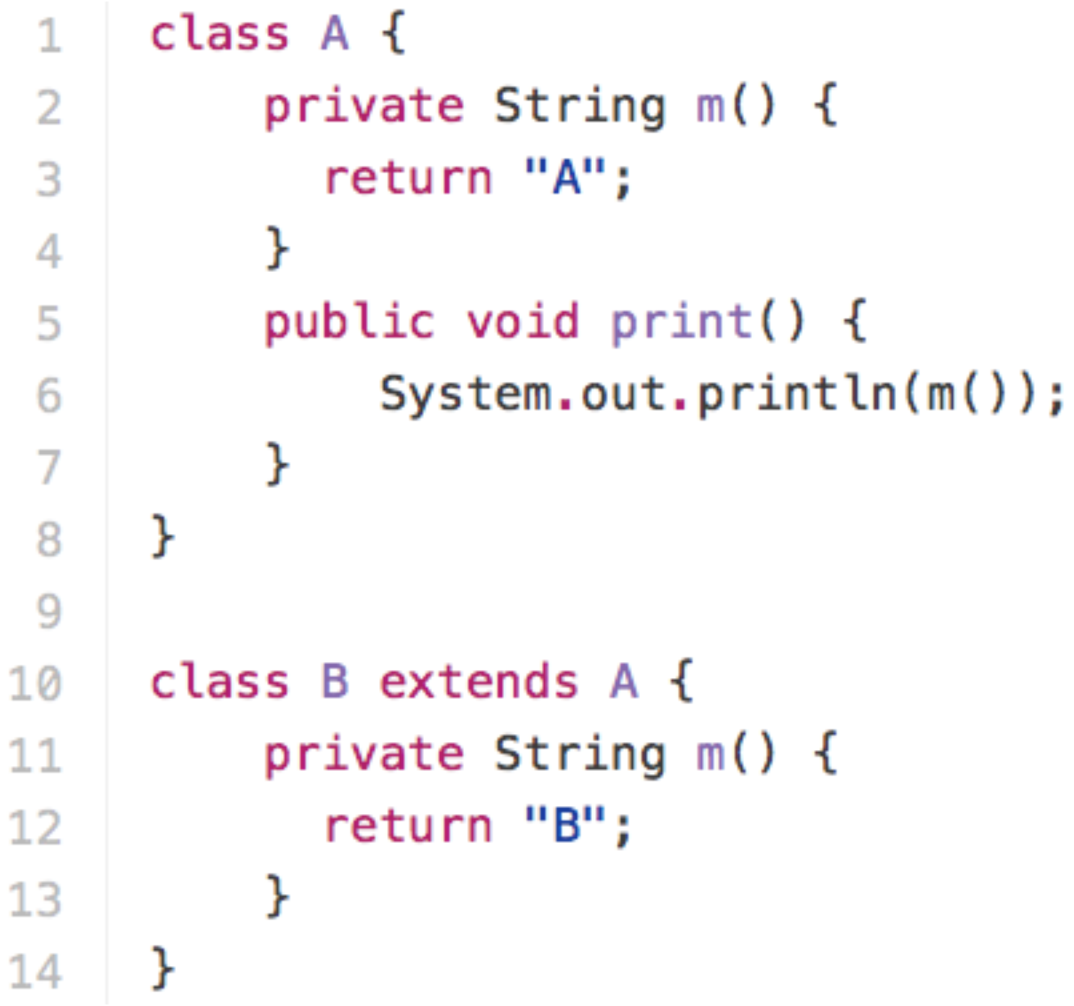
- “B” is printed. Dynamic dispatch!



```java
1   class A {
2       public String m() {
3         return "A";
4       }
5       public void print() {
6           System.out.println(m());
7       }
8   }
9
10  class B extends A {
11      public String m() {
12        return "B";
13      }
14  }
15
16
```

Private Methods

- Ok how about now. Is the method defined on line 11 legal Java?

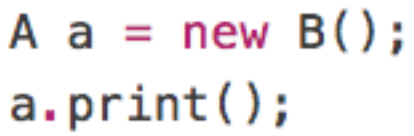


```
 1  class A {
 2      private String m() {
 3        return "A";
 4      }
 5      public void print() {
 6          System.out.println(m());
 7      }
 8  }
 9
10  class B extends A {
11      private String m() {
12        return "B";
13      }
14  }
```

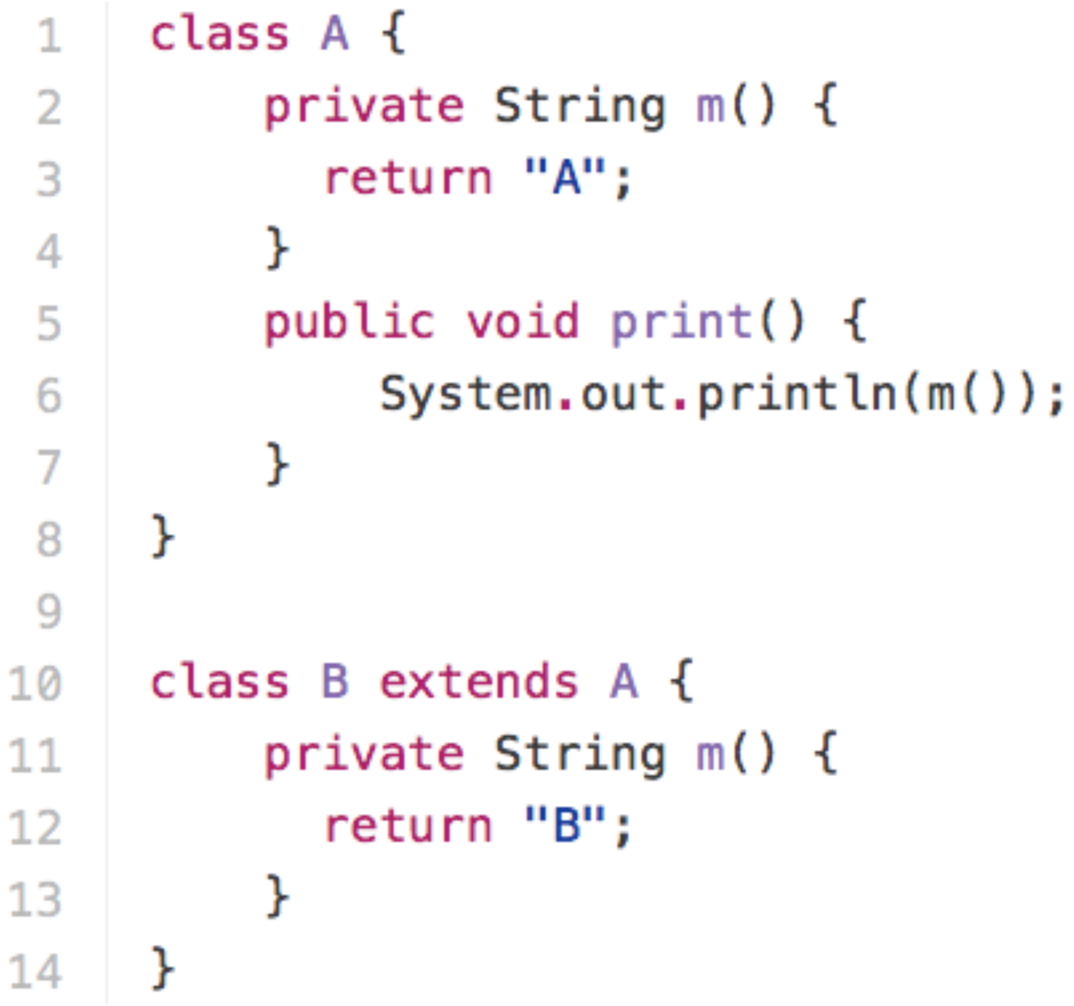
Private Methods

- Is the method defined on line 11 legal Java?

- Yes, what is the result of the following code?



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

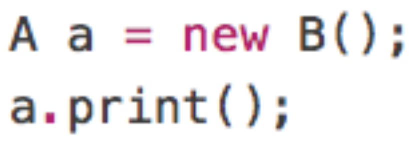


```
1   class A {
2       private String m() {
3         return "A";
4       }
5       public void print() {
6           System.out.println(m());
7       }
8   }
9
10  class B extends A {
11      private String m() {
12        return "B";
13      }
14  }
```

Private Methods

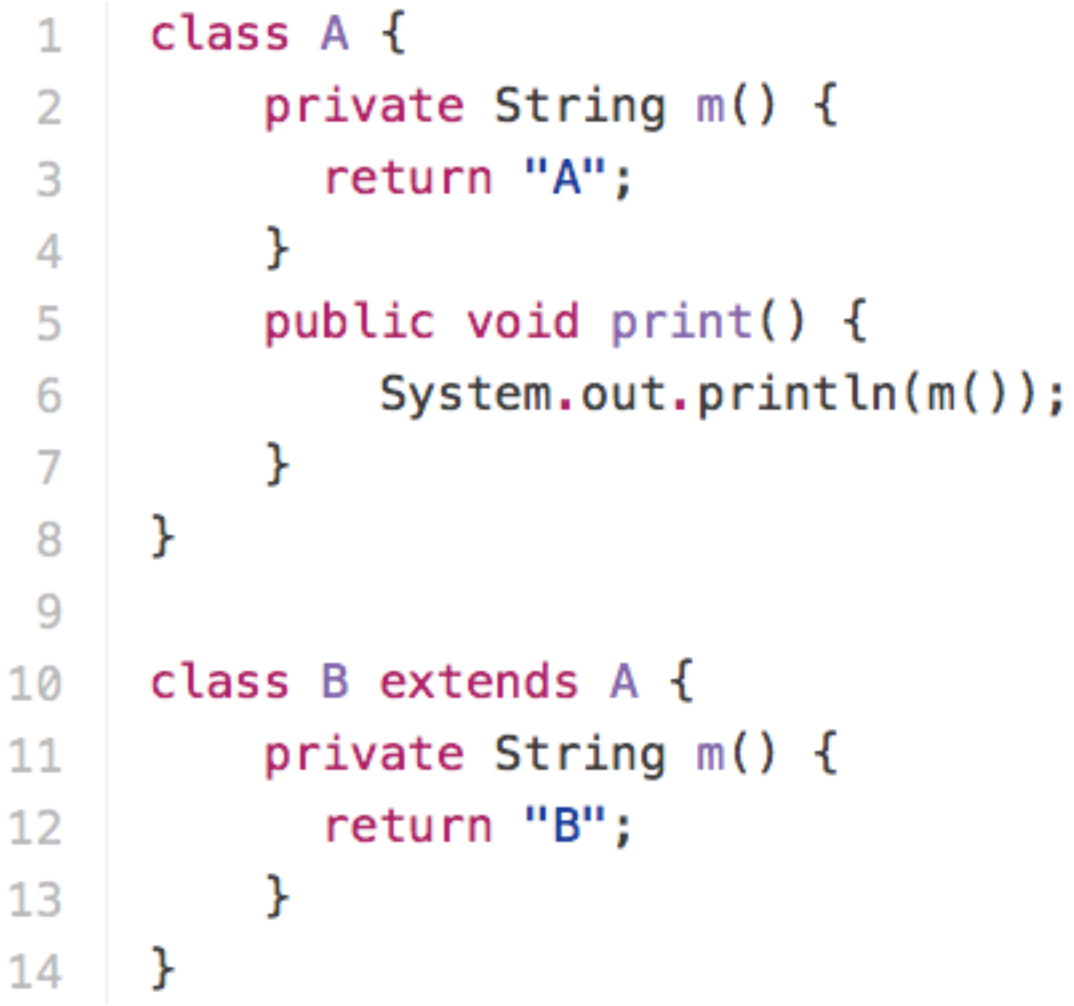
- Is the method defined on line 11 legal Java?

- Yes, what is the result of the following code?



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

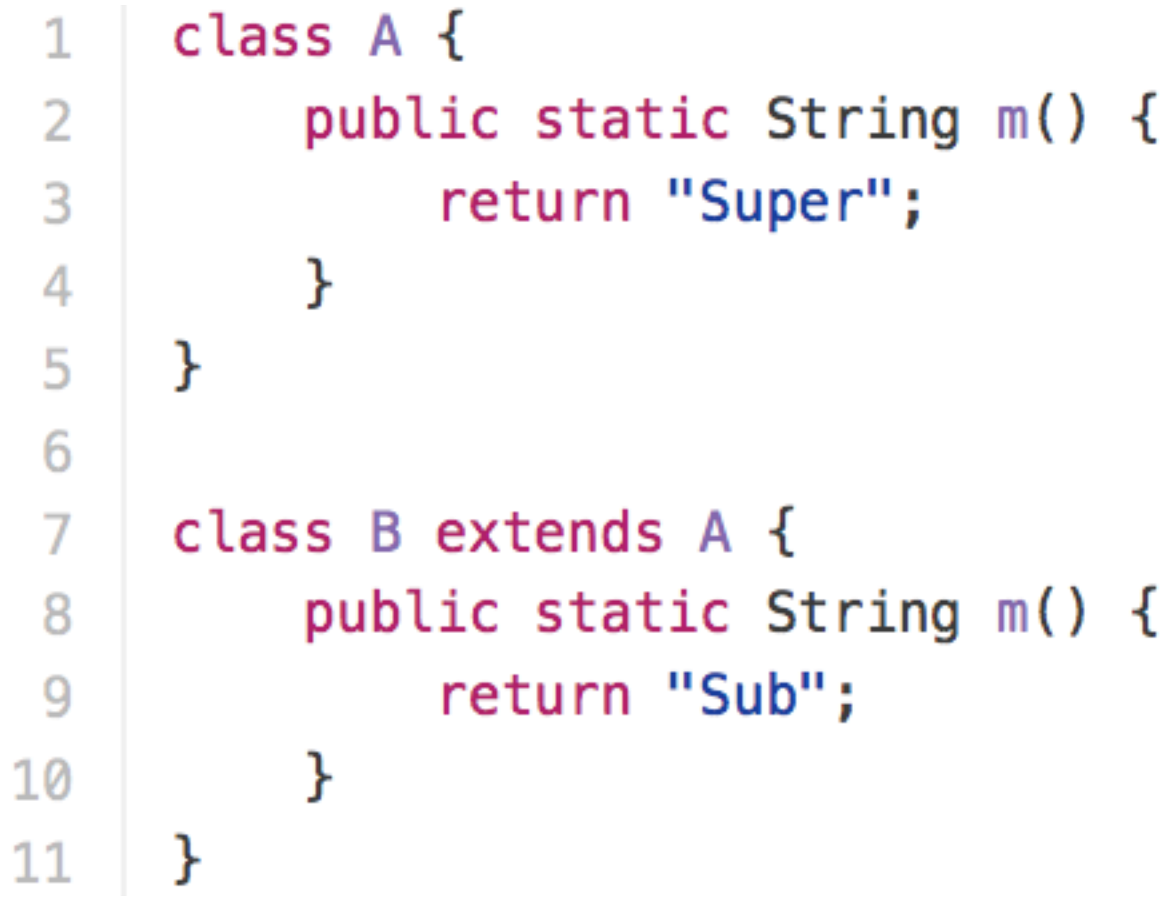
- “A” is printed. Private methods are not in the vtable, no dynamic dispatch.



```
1   class A {
2       private String m() {
3         return "A";
4       }
5       public void print() {
6           System.out.println(m());
7       }
8   }
9
10  class B extends A {
11      private String m() {
12        return "B";
13      }
14  }
```

Static Methods

- Is the method defined on line 8 legal Java?

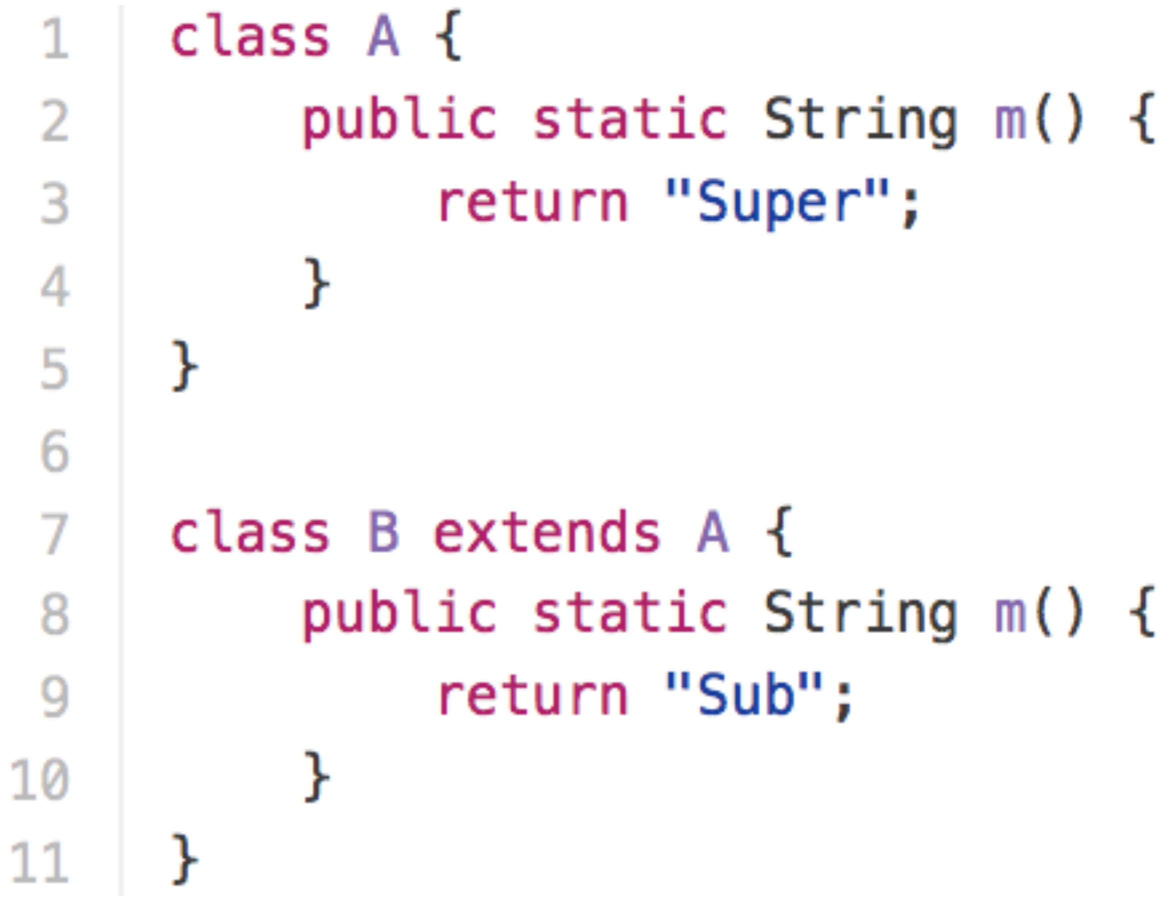


```
1   class A {
2       public static String m() {
3           return "Super";
4       }
5   }
6
7   class B extends A {
8       public static String m() {
9           return "Sub";
10      }
11  }
```

Static Methods

- Is the method defined on line 8 legal Java?

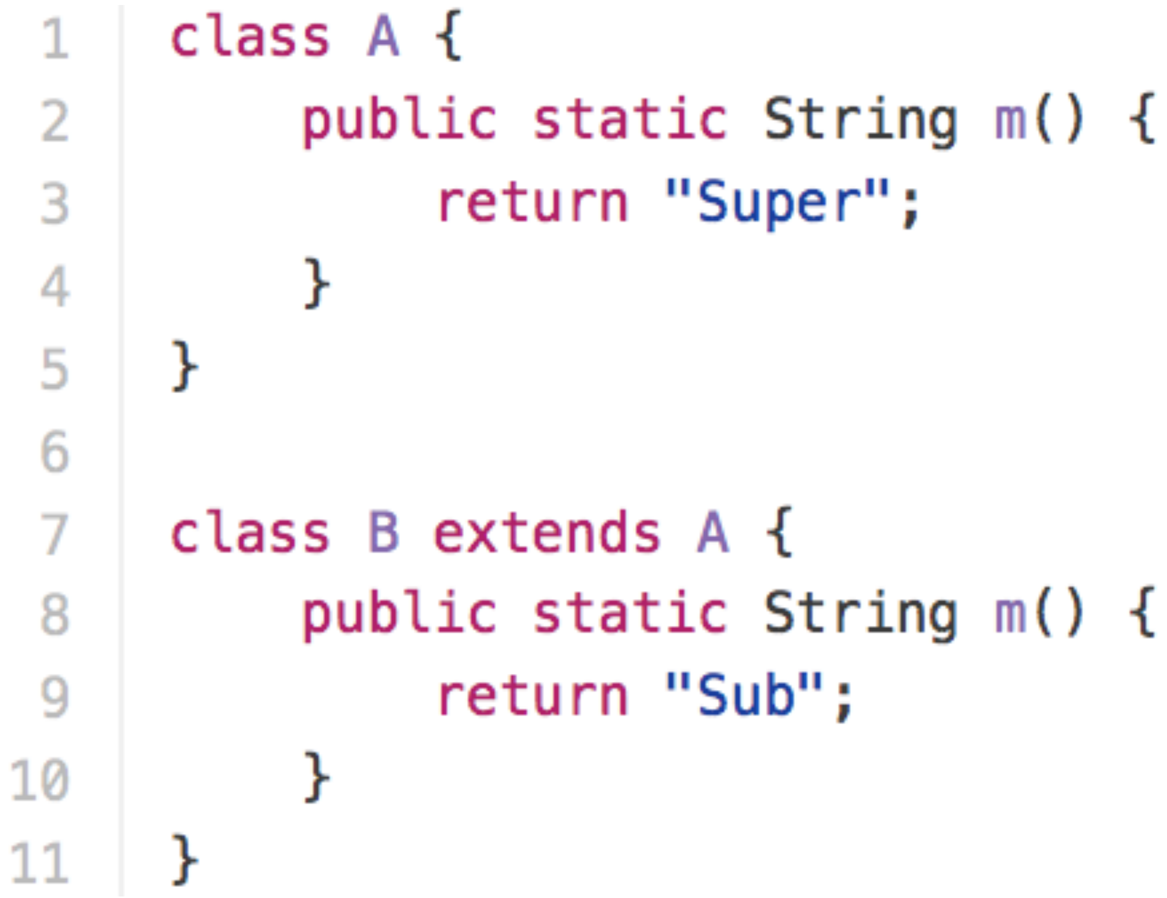
- Yes. Is it an override?



```
1   class A {
2       public static String m() {
3           return "Super";
4       }
5   }
6
7   class B extends A {
8       public static String m() {
9           return "Sub";
10      }
11  }
```

Static Methods

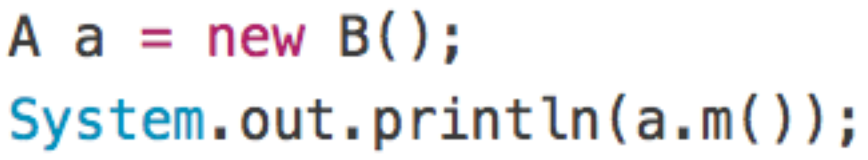
- Is the method defined on line 8 legal Java?
- Yes. Is it an override?
- No. This is whats known as ‘method hiding’. This is specific to static methods.
- If invoked on an instance, what version gets called depends on static type of instance.



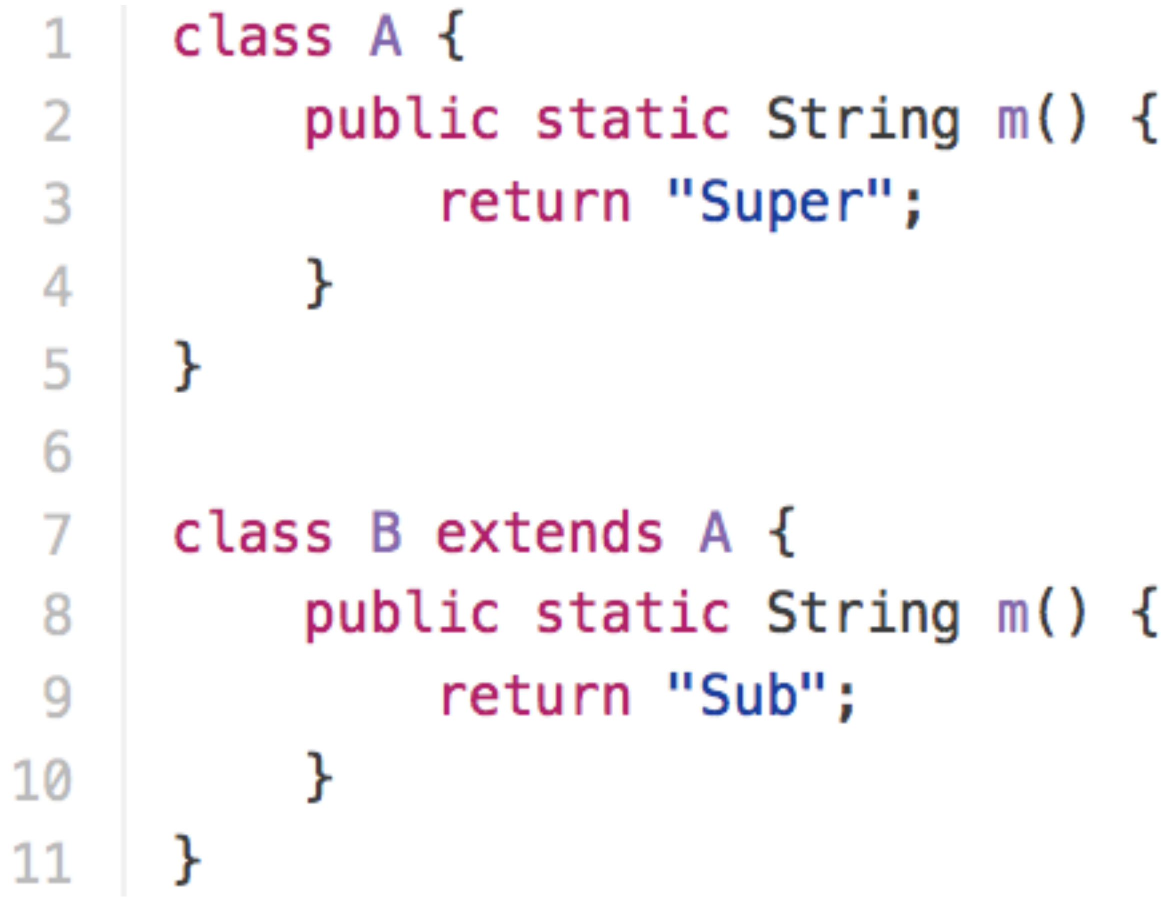
```
 1  class A {
 2      public static String m() {
 3          return "Super";
 4      }
 5  }
 6
 7  class B extends A {
 8      public static String m() {
 9          return "Sub";
10      }
11  }
```

Static Methods

- Ok then, what is the result of this code?



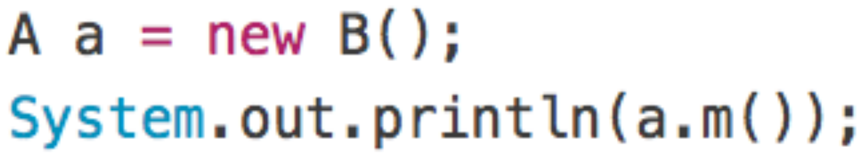
```
A a = new B();
System.out.println(a.m());
```



```
class A {
    public static String m() {
        return "Super";
    }
}

class B extends A {
    public static String m() {
        return "Sub";
    }
}
```

Static Methods

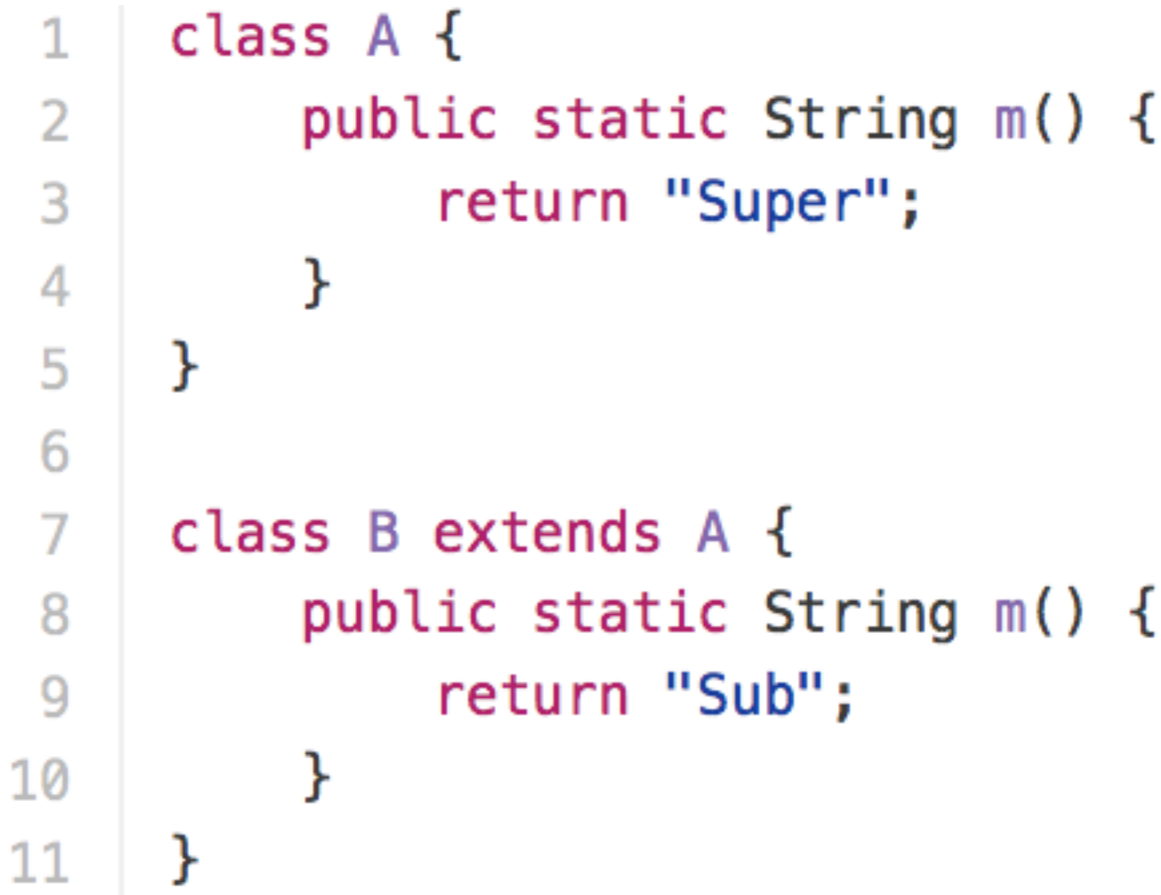
- Ok then, what is the result of this code?



```
A a = new B();
System.out.println(a.m());
```

- “Super” is printed.

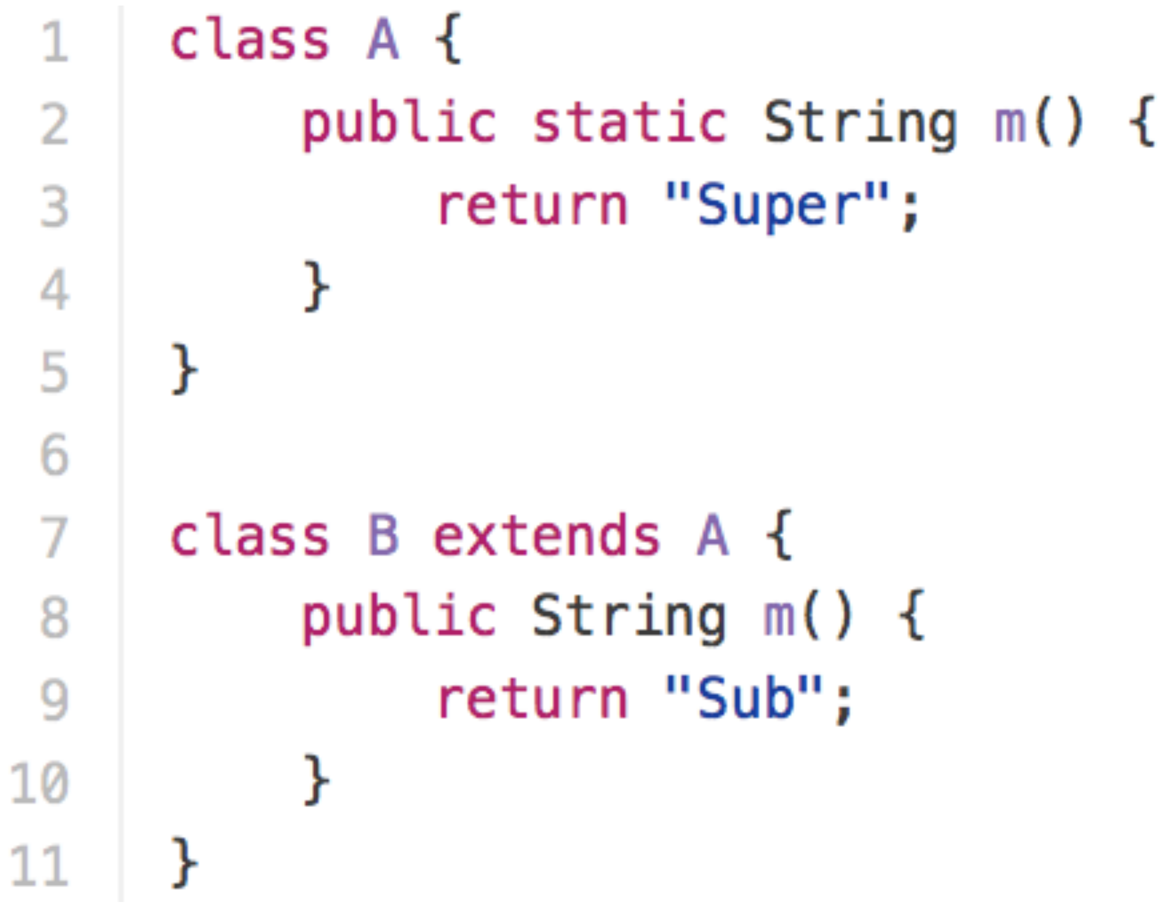
- No dynamic dispatch!



```
class A {
    public static String m() {
        return "Super";
    }
}

class B extends A {
    public static String m() {
        return "Sub";
    }
}
```

Static Methods

- Is the method defined on line 8 legal Java?



```
 1  class A {
 2      public static String m() {
 3          return "Super";
 4      }
 5  }
 6
 7  class B extends A {
 8      public String m() {
 9          return "Sub";
10      }
11  }
```

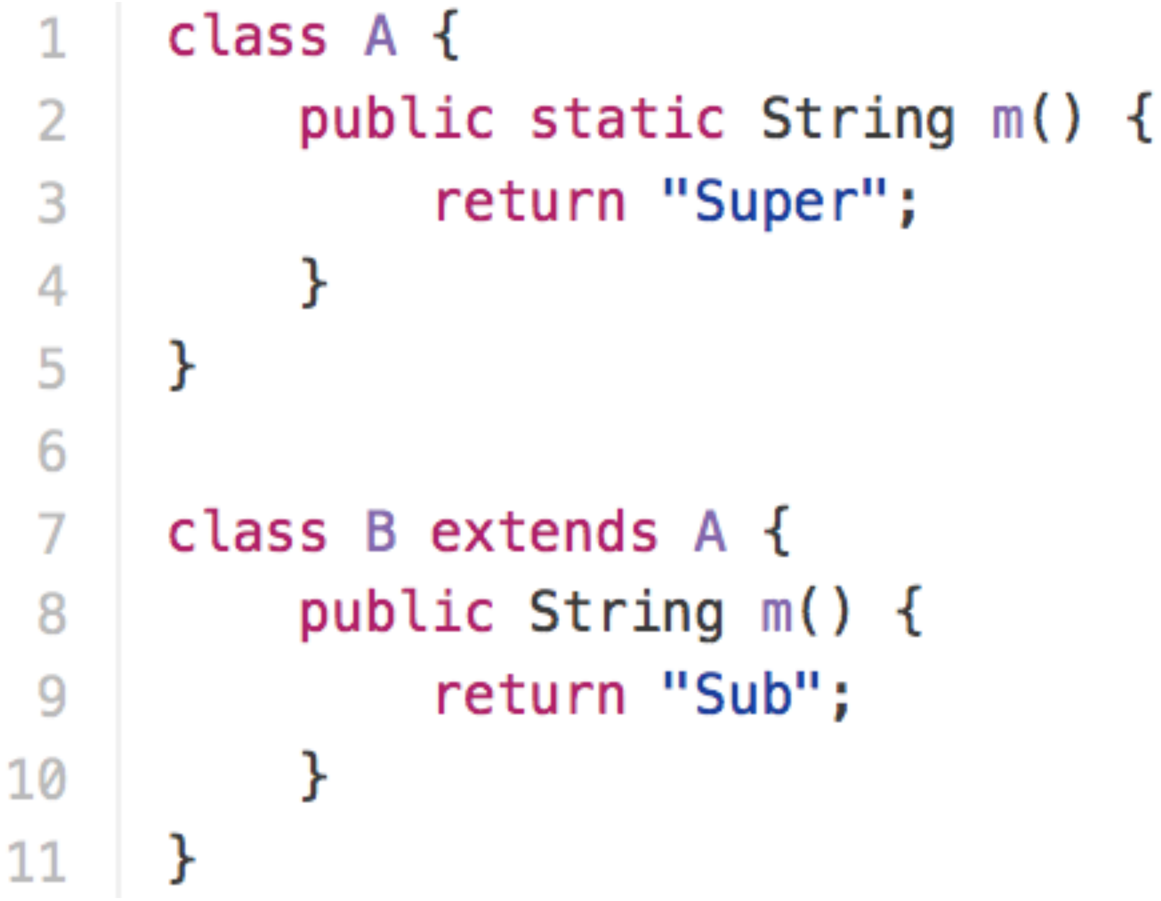
Static Methods

- Is the method defined on line 8 legal Java?

- No. This is a compile error.

- An instance method cannot override or hide a static method.

- What if A.m() was an instance method and B.m() was static?



```
 1  class A {
 2      public static String m() {
 3          return "Super";
 4      }
 5  }
 6
 7  class B extends A {
 8      public String m() {
 9          return "Sub";
10      }
11  }
```

Static Methods

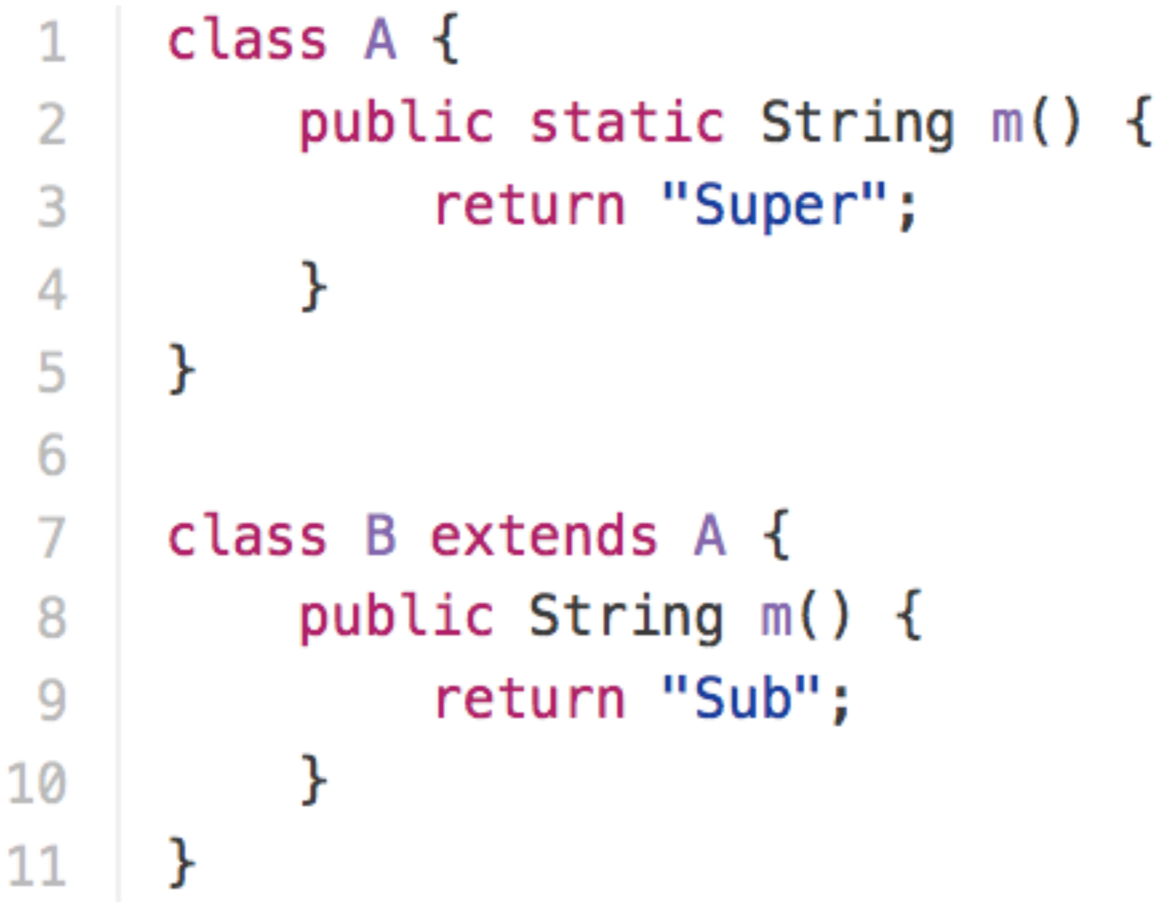
- Is the method defined on line 8 legal Java?

- No. This is a compile error.

- An instance method cannot override or hide a static method.

- What if A.m() was an instance method and B.m() was static?

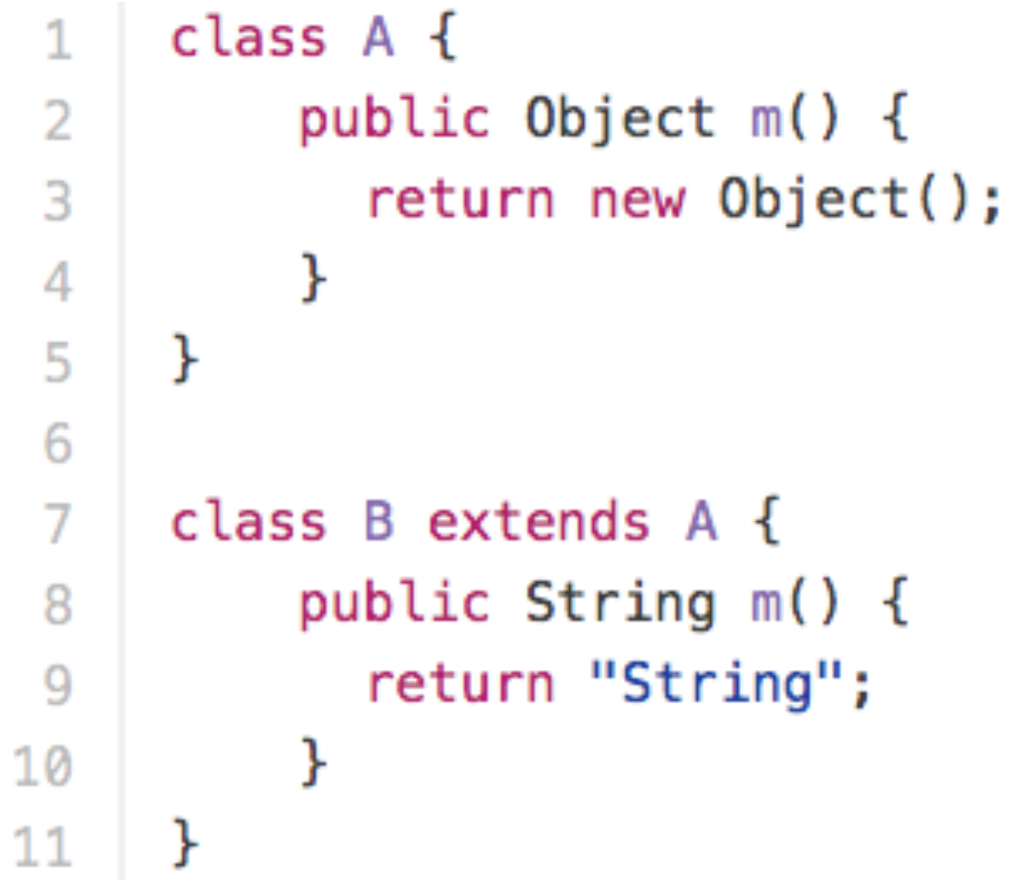
- Also a compile error.



```
1   class A {
2       public static String m() {
3           return "Super";
4       }
5   }
6
7   class B extends A {
8       public String m() {
9           return "Sub";
10      }
11  }
```

Return Types

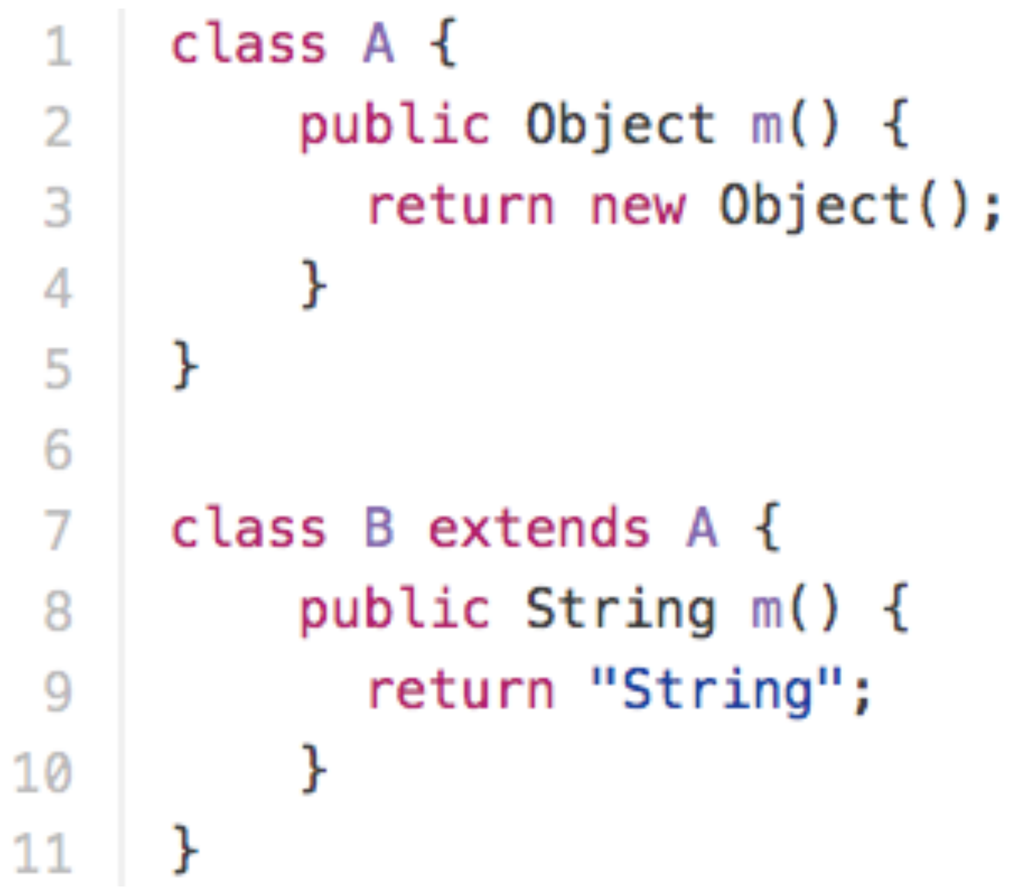
- Is the method defined on line 8 legal Java?



```
1   class A {
2       public Object m() {
3         return new Object();
4       }
5   }
6
7   class B extends A {
8       public String m() {
9         return "String";
10      }
11  }
```

Return Types

- Is the method defined on line 8 legal Java?
- Yes, but is it an override?



```
1  class A {
2      public Object m() {
3        return new Object();
4      }
5  }
6
7  class B extends A {
8      public String m() {
9        return "String";
10     }
11 }
```

Return Types

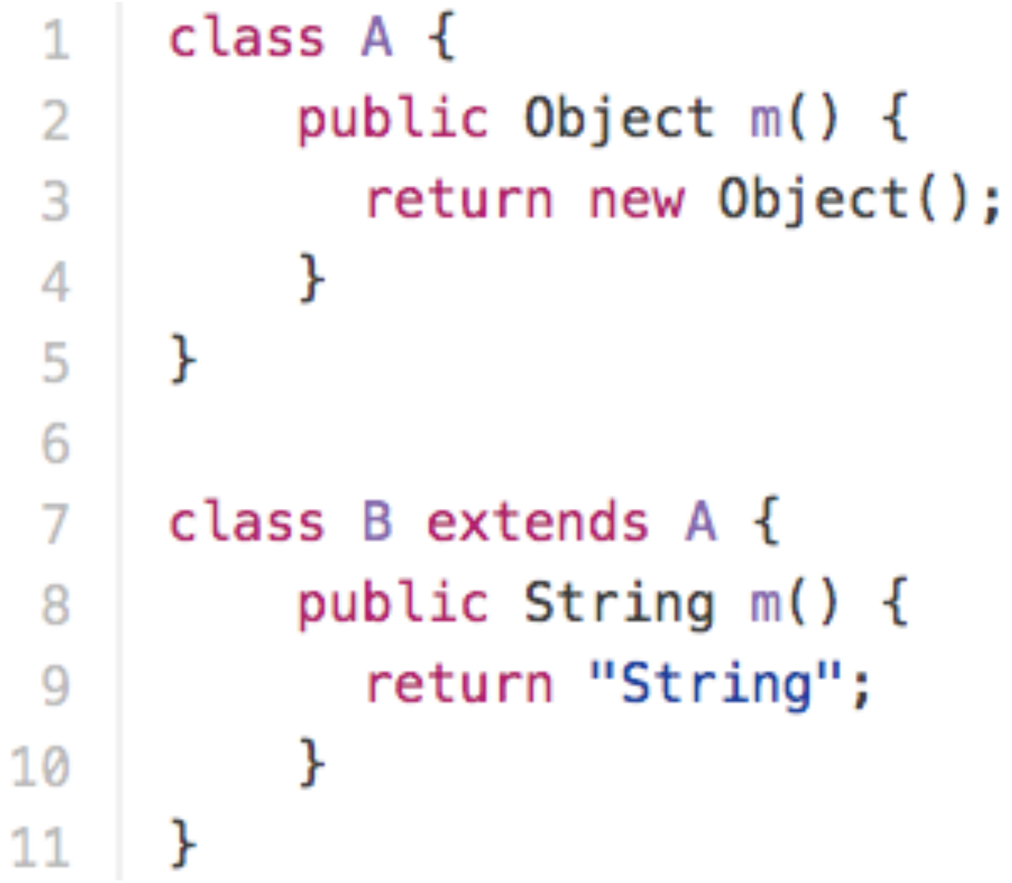
- Is the method defined on line 8 legal Java?

- Yes, but is it an override?

- Yes, an overriding method can also return a subtype of the type returned by the overridden method.

- This subtype is called a *covariant* return type.

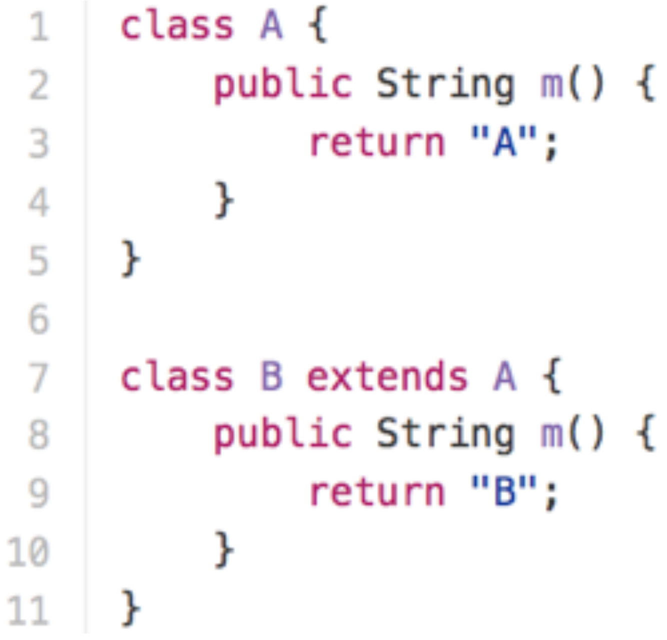
- Does this affect your translator?



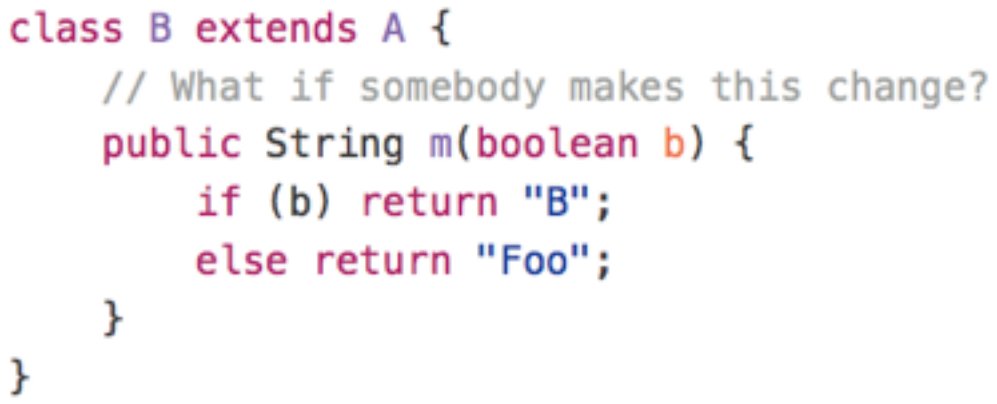
```
 1  class A {
 2      public Object m() {
 3          return new Object();
 4      }
 5  }
 6
 7  class B extends A {
 8      public String m() {
 9          return "String";
10      }
11  }
```

Override Annotation

- Lets say you have a class hierarchy like the on in lines 1-11
- Then later some junior developer who didn’t take NYU OOP comes along and changes B as we see below.
- m() no longer an override.
- Now B’s return “A” when m() is called!! Would be nice if this was a compile error!



```
1   class A {
2       public String m() {
3           return "A";
4       }
5   }
6
7   class B extends A {
8       public String m() {
9           return "B";
10      }
11  }
```



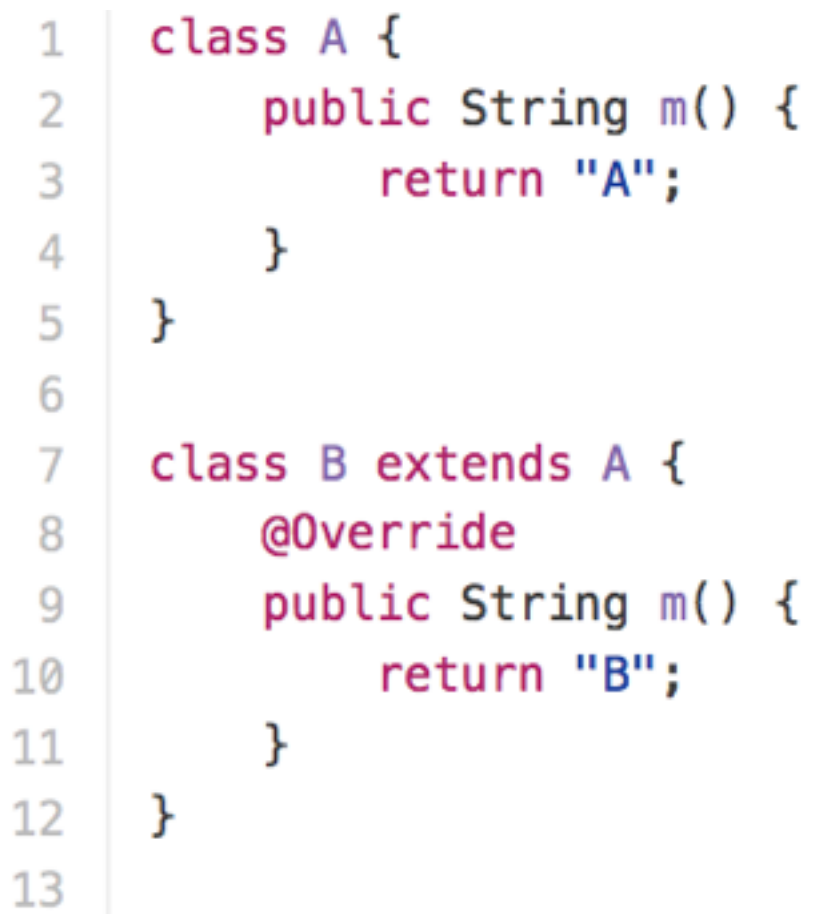
```
class B extends A {
    // What if somebody makes this change?
    public String m(boolean b) {
        if (b) return "B";
        else return "Foo";
    }
}
```

Override Annotation

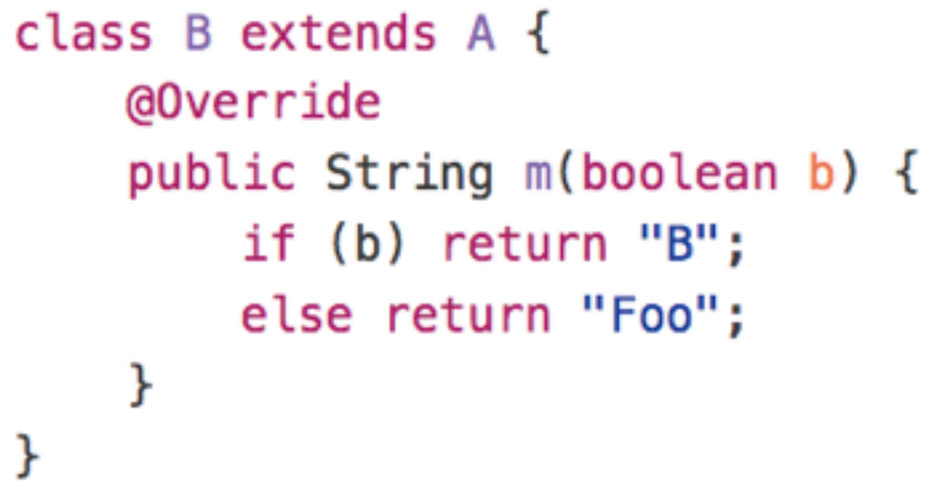
- See line 8

- Now if the code were changed as below, the code would not compile.

- The @Override annotation instructs the compiler that the method is being override if the following methods does not, it is a compiler error.



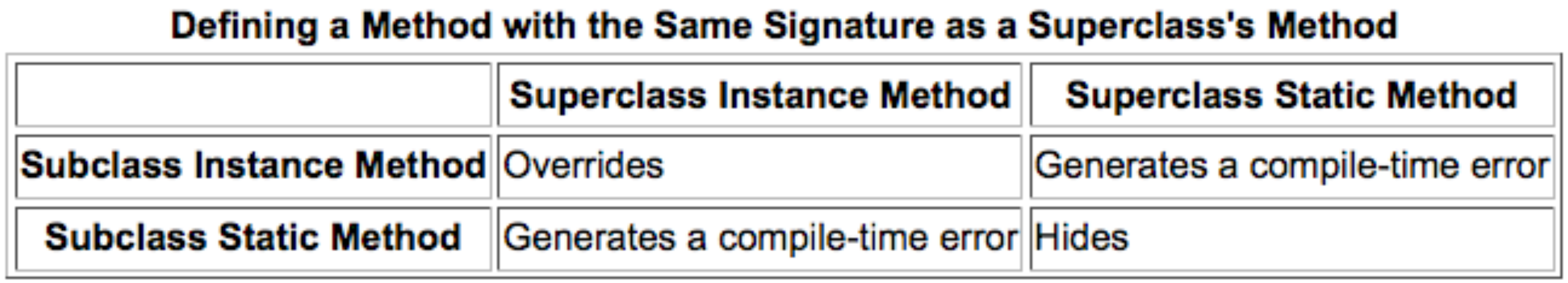
```
 1  class A {
 2      public String m() {
 3          return "A";
 4      }
 5  }
 6
 7  class B extends A {
 8      @Override
 9      public String m() {
10          return "B";
11      }
12  }
13
```



```
class B extends A {
    @Override
    public String m(boolean b) {
        if (b) return "B";
        else return "Foo";
    }
}
```

Take-aways

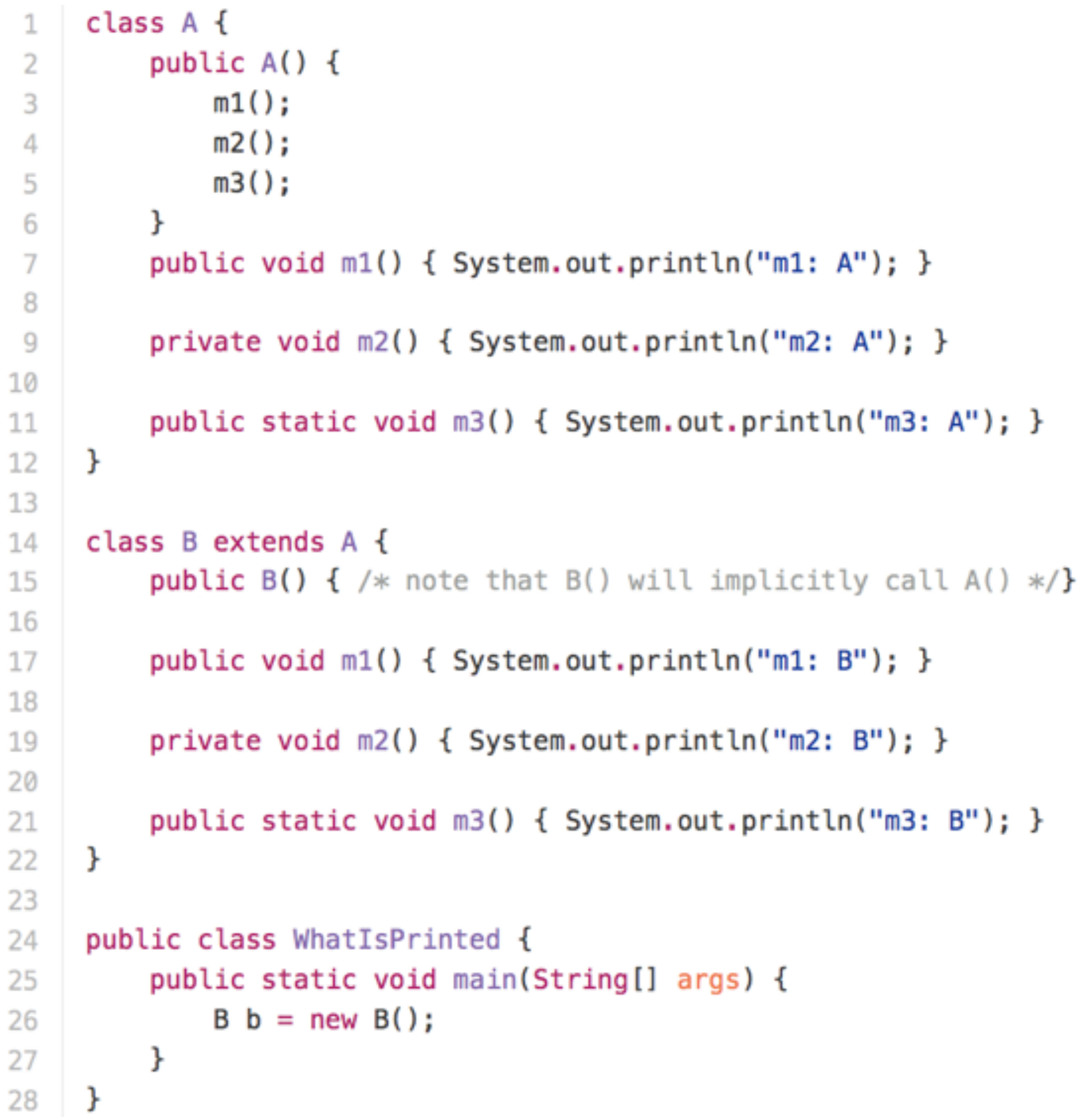
- Return types are covariant w.r.t. overrides.
- The override annotation expresses the intent to override explicitly and future-proof your code.
- Private and static methods cannot be overridden. Static methods can be *hidden*.
- When you define a method with the same signature as a method in a superclass….



Defining a Method with the Same Signature as a Superclass's Method

	Superclass Instance Method	Superclass Static Method
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

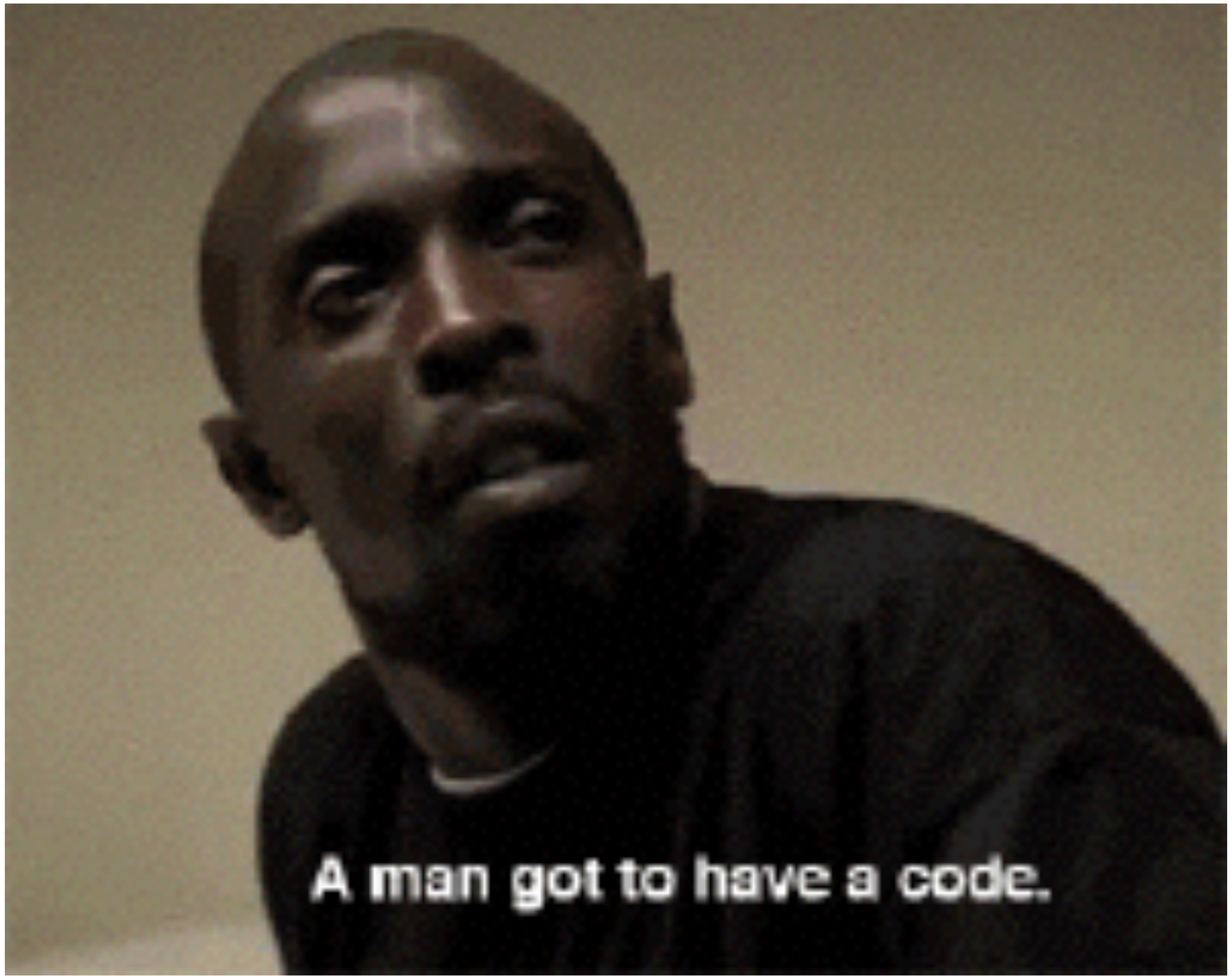
Interview Question Time



```java
class A {
    public A() {
        m1();
        m2();
        m3();
    }
    public void m1() { System.out.println("m1: A"); }

    private void m2() { System.out.println("m2: A"); }

    public static void m3() { System.out.println("m3: A"); }
}

class B extends A {
    public B() { /* note that B() will implicitly call A() */}

    public void m1() { System.out.println("m1: B"); }

    private void m2() { System.out.println("m2: B"); }

    public static void m3() { System.out.println("m3: B"); }
}

public class WhatIsPrinted {
    public static void main(String[] args) {
        B b = new B();
    }
}
```

As usual…

- Code: https://github.com/nyu-oop/method-overriding-java



Symbol Table

Symbol Table

- In a compiler, after ASTs have been constructed, a compiler must check whether the input program is type-correct.

- A compiler checks whether the use of names (such as variables, functions, type names) is consistent with their definition in the program.

- For example, if a variable x has been defined to be of type int, then x+1 is correct since it adds two integers while x[1] is wrong.

Symbol Table

- Consequently, it is necessary to remember declarations so that inconsistencies and misuses can be detected during type checking.

- This is the task of a symbol table.

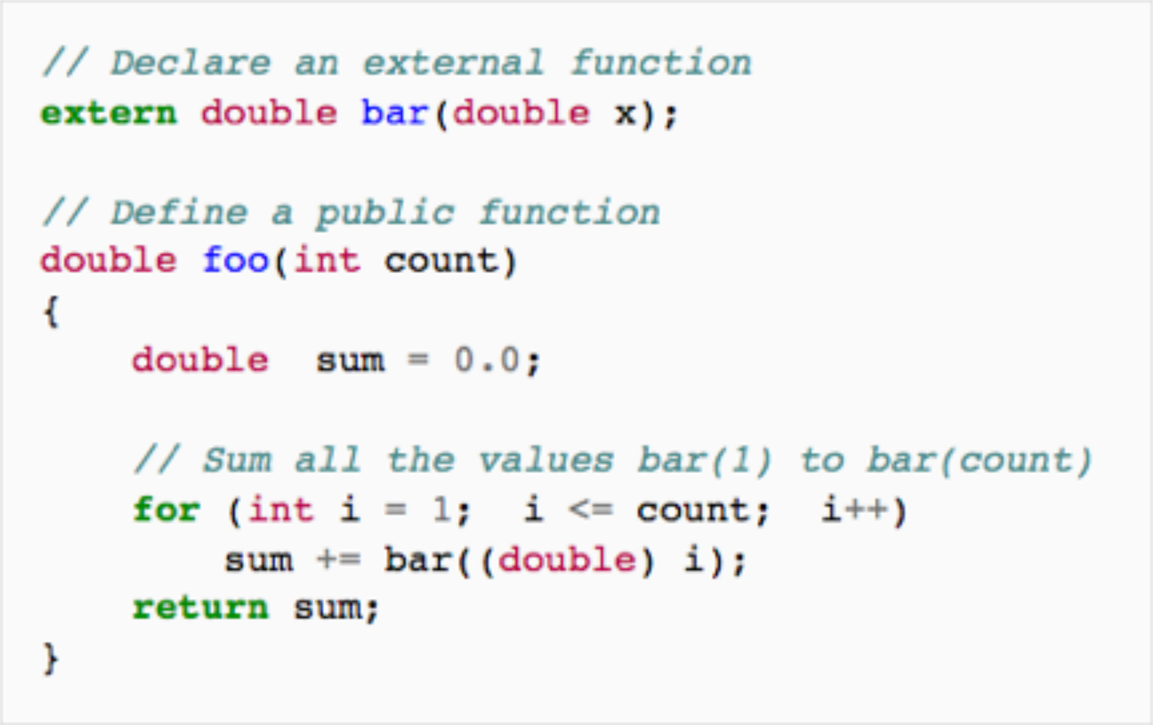
Symbol Table

- More specifically, a symbol table stores:
  - for each variable name, its type. If the variable is an array, it also stores dimension information. It may also store storage class, offset in activation record etc.
  - for each constant name, its type and value.
  - for each function and procedure, its parameter list and its return type. Each formal parameter has name and type.

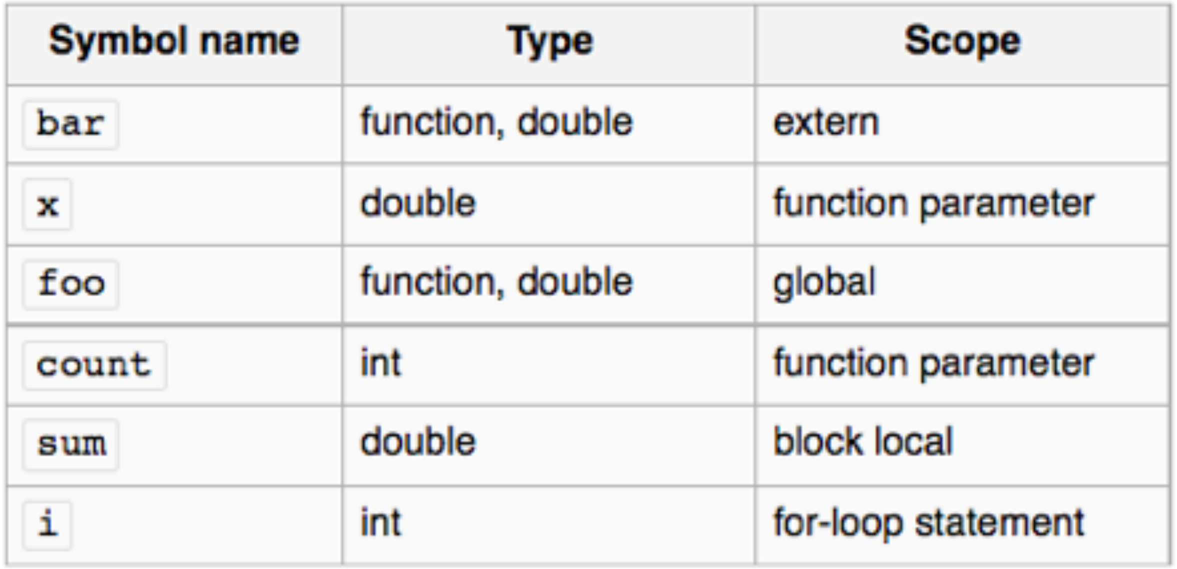
Symbol Table

- Moreover, a symbol table is a data structure used by a language translator or compiler where each identifier in a program's source code is associated with information relating to its type and scope.

- For example…



```
// Declare an external function
extern double bar(double x);

// Define a public function
double foo(int count)
{
    double  sum = 0.0;

    // Sum all the values bar(1) to bar(count)
    for (int i = 1;  i <= count;  i++)
        sum += bar((double) i);
    return sum;
}
```



Symbol name	Type	Scope
`bar`	function, double	extern
`x`	double	function parameter
`foo`	function, double	global
`count`	int	function parameter
`sum`	double	block local
`i`	int	for-loop statement

Method Overloading in Java

Translator

- After array translation, then next feature we will support in our translator is method overloading.

- Which means we need to resolve a method call site to a definition when we may have more than one method definition with the same name.

Method Overloading Review

- Java supports overloading methods, and Java can distinguish between methods with he same name but different signatures.

- This means that methods within a class can have the same name if they have different parameter lists.

Type & Arity

- Overloaded methods are disambiguated by parameter list *type* and *arity*
  - *type* is a classification identifying one of various types of data, such as real, integer or boolean.
  - *arity* is the number of arguments or operands a function or operation takes.

Overloading Vs Overriding

- When overloading a method, you are really just making a number of different methods that happen to have the same name. It is *resolved statically at compile time* which of these methods are used.
- This should not be confused with overriding where the correct method is chosen at *runtime*, e.g. through virtual functions.

Example

- Suppose that you have a class that can use calligraphy to draw various types of data (strings, integers, and so on)

- It contains a method for drawing each data type.

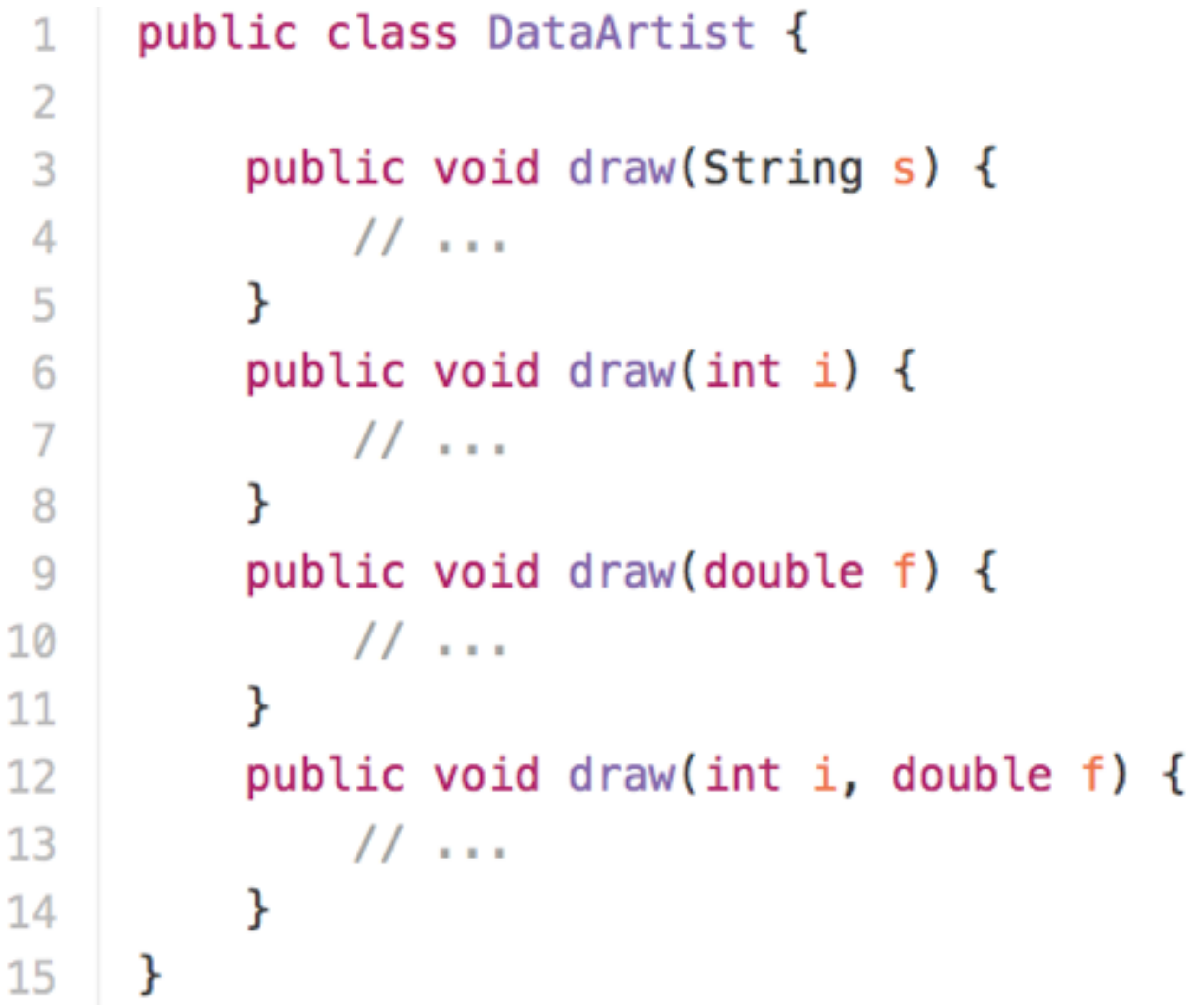
- It is cumbersome to use a new name for each method—for example, drawString, drawInteger, drawFloat, etc.

- You can instead use the same name for all the drawing methods but pass a different argument list to each method.

Example

- The class might declare four methods named draw, each of which has a different parameter list.

- Moreover, draw(String s) and draw(int i) are distinct and unique methods because they require different argument types.



```java
public class DataArtist {

    public void draw(String s) {
        // ...
    }
    public void draw(int i) {
        // ...
    }
    public void draw(double f) {
        // ...
    }
    public void draw(int i, double f) {
        // ...
    }
}
```

Caveats

- You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

- The compiler does *not* consider return type when differentiating methods, so you cannot declare two methods with the same signature but with different return types.

- Overuse can make code less readable.

Implementation Details

- Overloaded methods have separate slots in the vtable because the compiler treats them as totally different, unrelated methods.

- The sameness of their names is a convenience for us as programmers, and does not change the execution model of our language.

Let's look at some code…

https://github.com/nyu-oop/method-overloading-java

Rules for Overload Resolution

- From the Java language specification: first determine the class then find methods that are applicable and accessible.

- Now we need a symbol table to track statically declared variable types so that we can decide which methods are applicable.

- xtc.util.SymbolTable already exists, we’ll look at how to use it very soon.

Rules for Overload Resolution

- We also need to make sure that we only use accessible methods.
- Moreover, do not attempt to access private methods from a subclass or a protected method from outside the hierarchy or a package private method from outside the package.

Rules for Overload Resolution

- If more than one method declaration is both accessible and applicable, choose the most specific method.

- One method declaration is more specific than another if the *types* are more specific.

- So to return to our example from Overloading.java, neither m(A, B) or m(B, A) is more specific than the other. This is why we encountered an ambiguity when attempting to call m(b, b)

Rules for Overload Resolution

- Finally, we need to make sure that the chosen method is appropriate: for example, calling an instance method in a static context will not compile

Implications for the Translator

- If there are overloaded methods, we need to do some name mangling for our vtable because the method name is no longer unique.

- If we overload a method in a subclass, the vtable for the superclass may also have a mangled name, so we need some convention that we obey.

- However, we do have the ability to read the entirety of the code before beginning translation -- highly useful. (I.e we can do a pass and transmute names)

Implications for the Translator

- Name mangling is necessary since members of the vtable are function pointers. Function pointers can't be distinguished by types of parameters like methods can be.

- For example, in Overloaded we have 14 methods named m, but in our vtable there is no concept of "slot overloading" so each of these 14 methods need a different name, just like you can't declare an int n and a double n in the same scope.

Implications for the Translator

- A little tricky, but *very* testable. A perfect thing to have unit test coverage on.

Conceptual Recap

- From a language design perspective, we want to keep concepts like static, private, overloading orthogonal.

- From a programmer's perspective, we want overloading to achieve greater conciseness, but don't want to change runtime behavior of our code.

Fair Warning

- Phases 1-5 need to be completed asap.
- More inputs coming soon.
- Arrays will be your next task.
- Then method overloading.
- And we will introduce memory management before the end of the semester too.
- Still lots to do!