Object-Oriented Programming

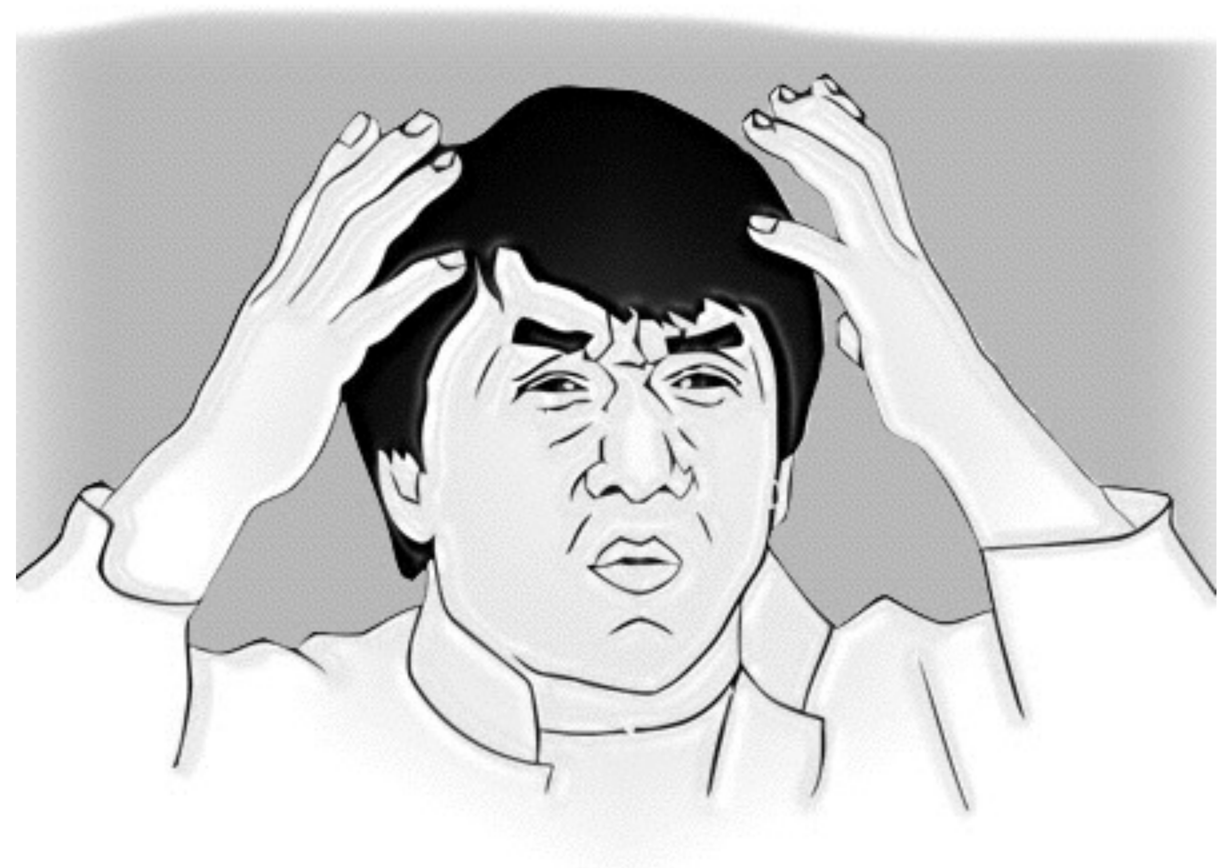
CSCI-UA 0470-001
Class 8
Instructor: Randy Shepherd

Last Class

- We talked about…
  - inheritance and subclassing
  - method overloading
  - virtual methods
  - dynamic and static types
  - dynamic dispatch

Project Goal

- Translate those concepts in our Java subset to our C++ subset.
- However we **cannot** use C++ inheritance or virtual methods in the translator's target language.



Therefore..

- It will be necessary to leverage techniques similar to what the Java compiler might when it compiles inheritance.

- In order to do that, we first have to a discuss a couple of important concepts:

    - Object data layout in memory

    - Virtual method tables

Agenda

- To implement virtual method dispatch efficiently we need to think about the data layout of objects in memory.
- Towards that end, we will first understand inheritance and virtual methods by looking at the data layout of objects and vtables.
- Then we will review C++ code that implements java.lang.Object, java.lang.String and java.lang.Class so that you can see what these structures look like in C++ code without using inheritance and virtual methods.

Object Memory Layout

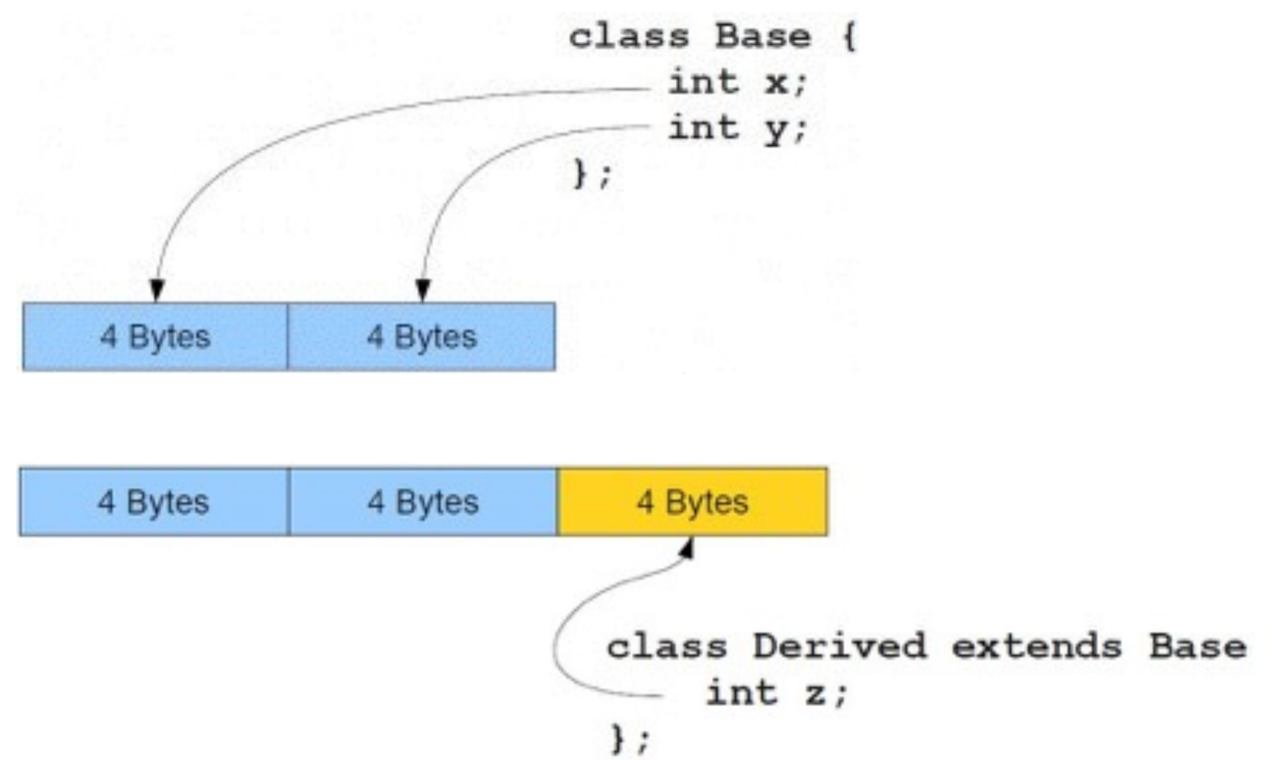
(images from http://www.programcreek.com/2011/11/what-do-java-objects-look-like-in-memory/)

Objects in Memory

- For any given object, how is the data organized in memory?
- How does the runtime find a particular property?
- How do these things work when dealing with inheritance hierarchies?

Object Data

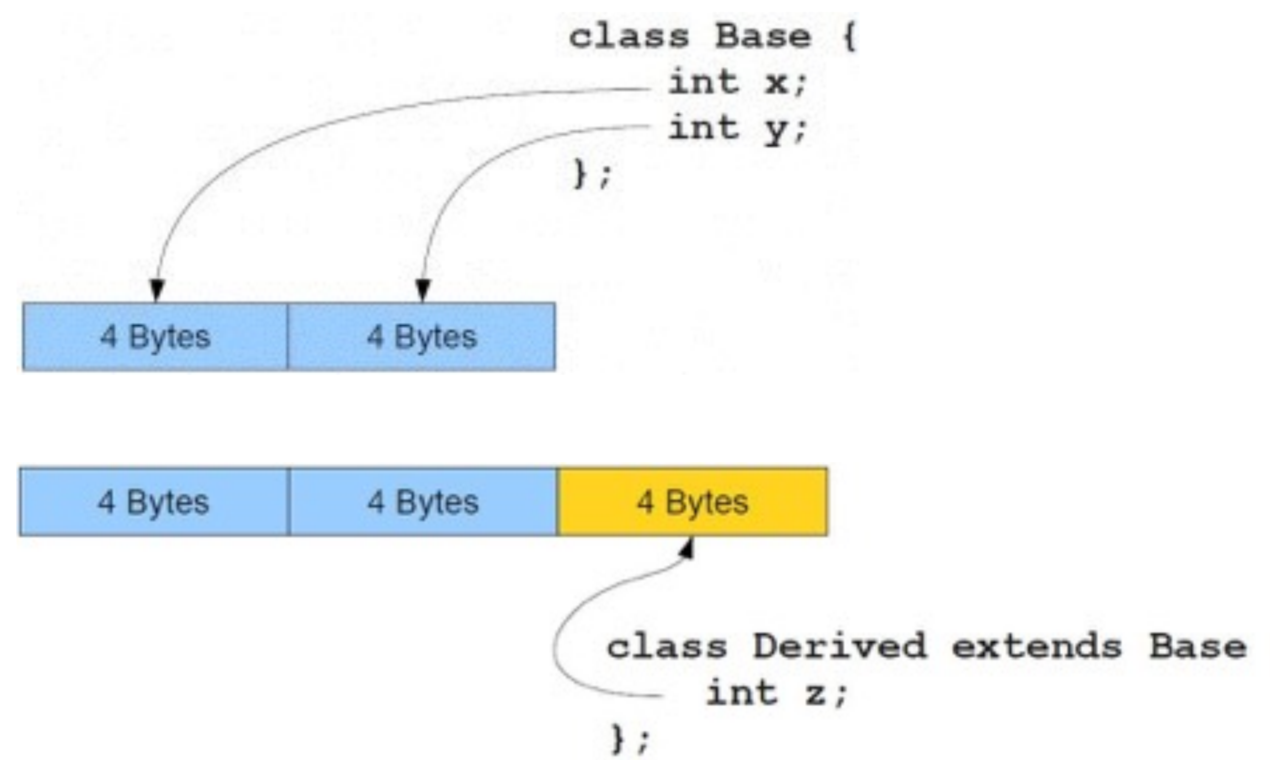
- Data members of a class are laid out contiguously in memory for each instance.
- Can be accessed via an offset.
- Child objects have the same memory layout as parent objects with additional space for subclass members.



Object Data

- Objects of type *Derived* can be polymorphically operated on as type *Base*, since the offsets are the same.

- No need for the runtime to check the dynamic type of instances of *Base*.

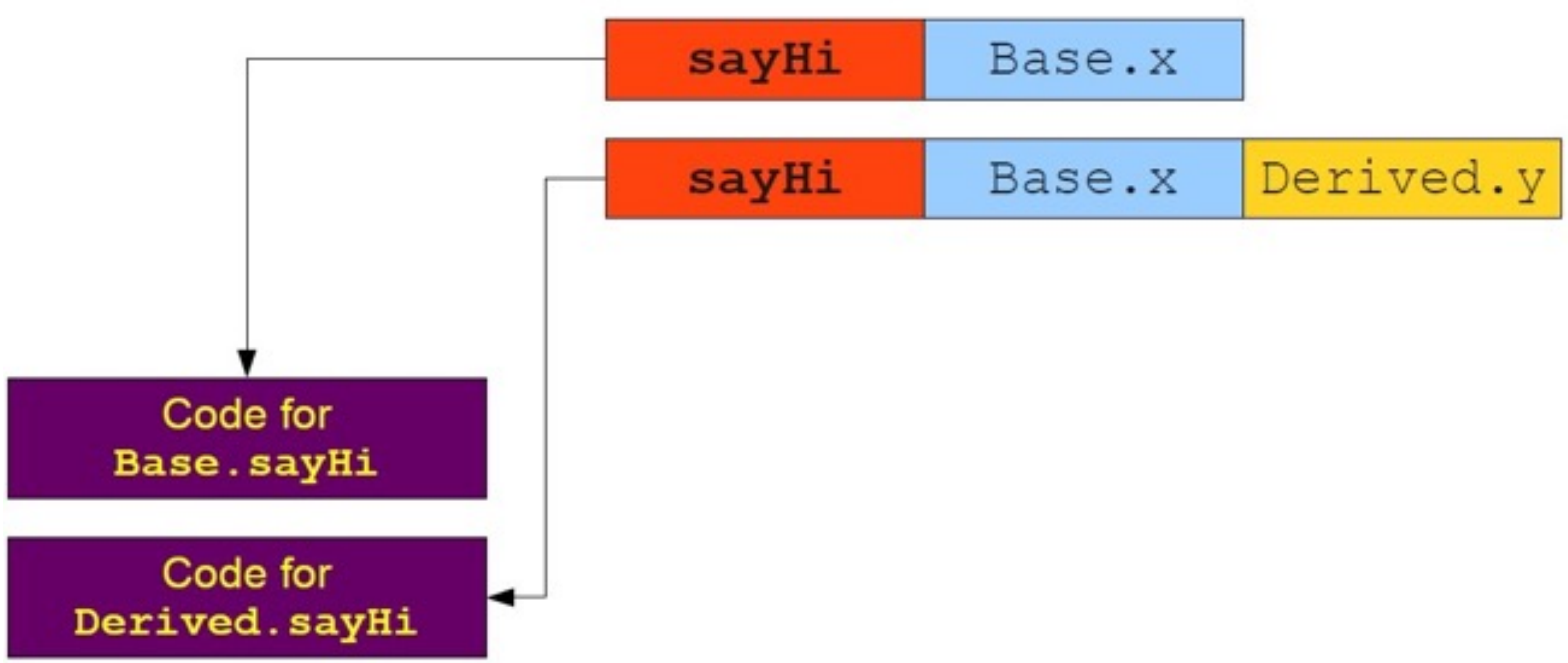


Object Methods

- So could we take this approach with methods?

```
class Base {
    int x;
    void sayHi() {
        Print("Base");
    }
}
```

```
class Derived extends Base {
    int y;
    void sayHi() {
        Print("Derived");
    }
}
```



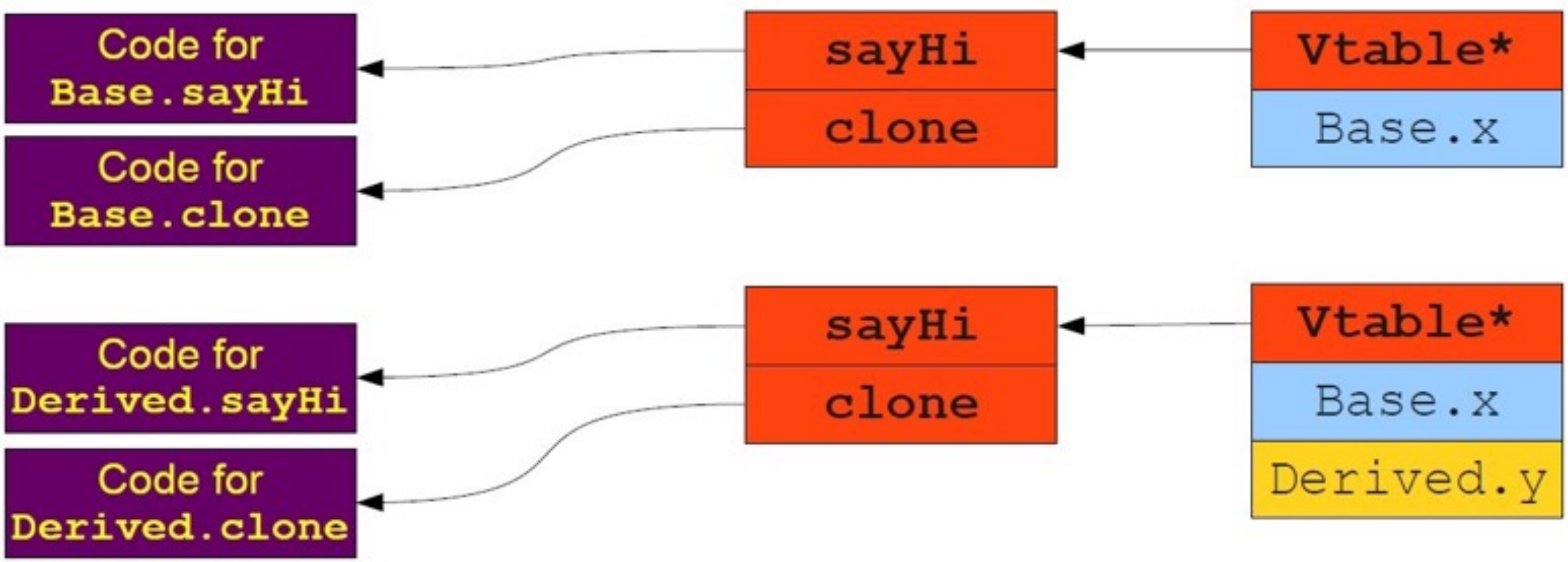
Problems with this approach

- For every method added in a subclass the size of each instance grows the size of one pointer
- Object creation slower
- Memory consumption higer

A better approach…

```
class Base {
    int x;
    void sayHi() {
        Print("Base");
    }
    Base clone() {
        return new Base;
    }
}
```

```
class Derived extends Base {
    int y;
    void sayHi() {
        Print("Derived");
    }
    Derived clone() {
        return new Derived;
    }
}
```



Object Methods

- When a class defines a virtual method, the compiler add a hidden member variable to the class.
- That pointer is to the ‘vtable’, or ‘virtual member table’.
- At instance creation (at runtime) table pointer will be set to point to the right vtable. i.e. the implementation for the *dynamic type*.
- Also, the ‘vtable’ is an array of *pointers to functions*.

Ok, with me so far?

Good, let's go on a tangent for a few slides

Internal & External Containment

- In C++, if you have members that are other classes, you have two choice of where to put that member in memory

- In Java, you *only* have external containment

- You can think of a vtable as one way of leveraging external containment.

```cpp
class PointContainer
{
  Point internal;
  Point* external;

  public:
  // accessors, mutators and so on...
};

// This is 'internal containment'.
// i.e. the memory for the point is inline
// with the memory for the PointContainer.
Point Point::internal = Point();

// This is 'external containment'.
// i.e. the memory for this point is elswhere
// and PointContainer has a pointer to it.
Point Point::external = new Point();
```

Higher Order Functions

- C++ has support for what is know as "higher order functions".
- A *higher order function* is defined as a function that does at least one of these things:
  - takes one or more functions as arguments
  - returns a function as its result
- Moreover, functions can be ***passed as arguments to other functions***

Function Pointers

- So how to we pass functions around?
- This is accomplished with function pointers
- Lets look at some code..
  - https://github.com/nyu-oop/function-pointers-cpp
- There is also this..
  - http://www.learncpp.com/cpp-tutorial/78-function-pointers/

C++: Static Initialization Order Fiasco

- Java has very strict rules about how and when order initialization of static members happens.

- C++ does not.

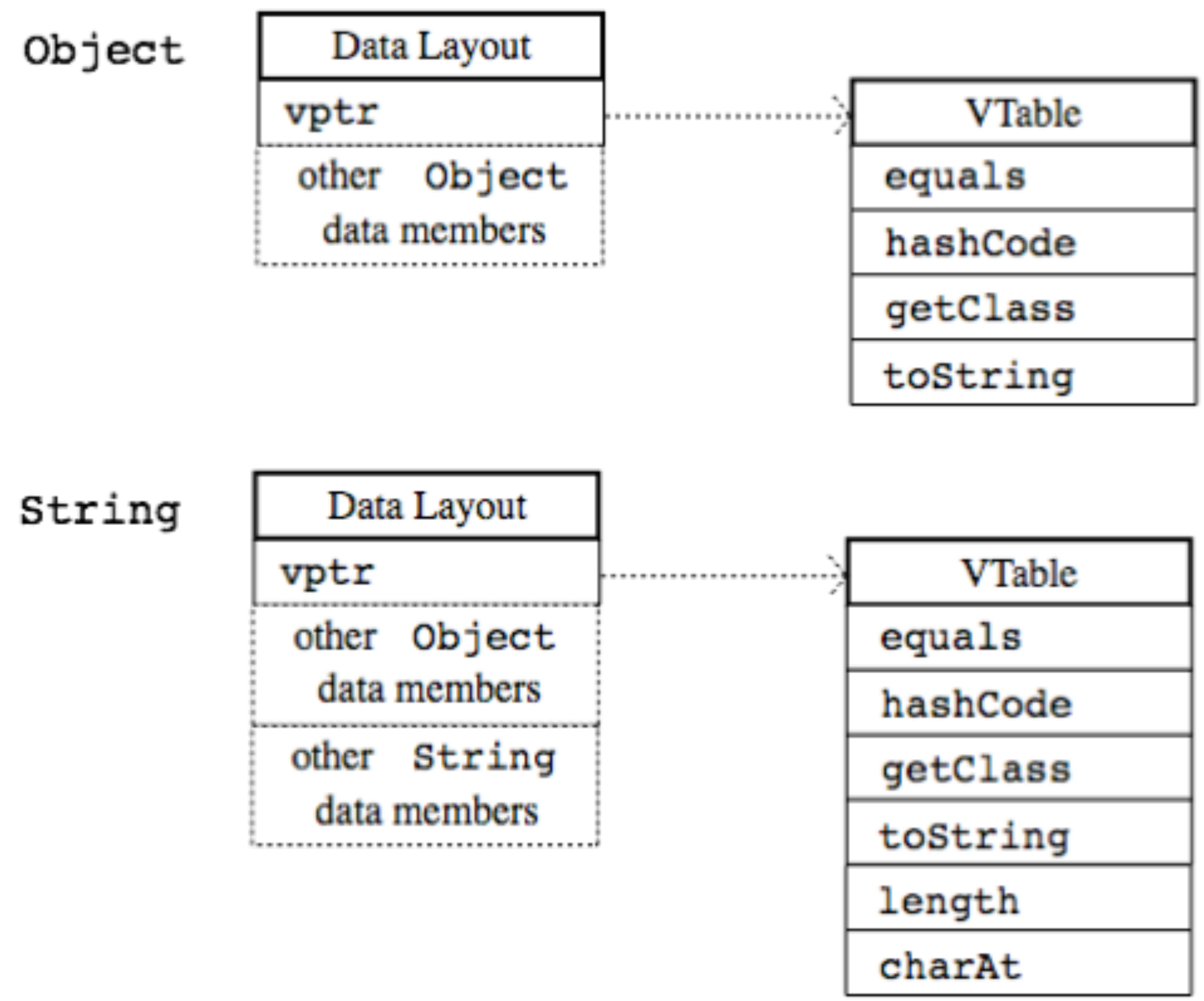
- What might happen in this code?

```
struct A
{
  static A  instanceA1;
  static A& getInstance() {
    return instanceA1;
  }
};

struct B
{
  static B instanceB1;
  static B& getInstance() {
    return instanceB1;
  }
  A& member;

  B(): member(A::getInstance()) {}
}

B b = new B();
```

Inheritance & Virtual Method Dispatch

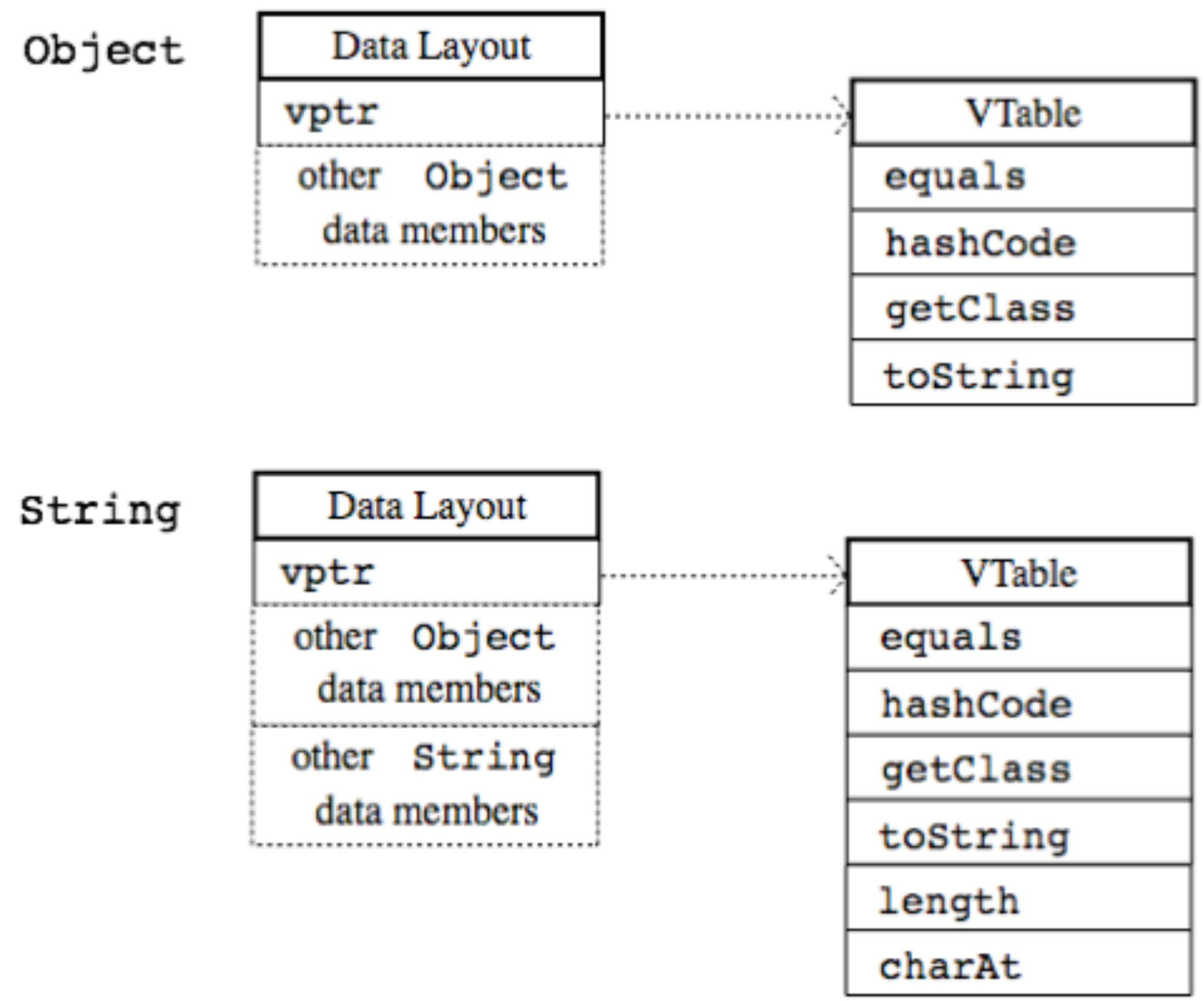
Data Layout of Class Inheritance in Java

- Java takes the previously described approach.
- Subclass members are stored below the memory reserved for the superclass.
- Code from the subclass can access the memory above using the same offsets as code from the superclass.



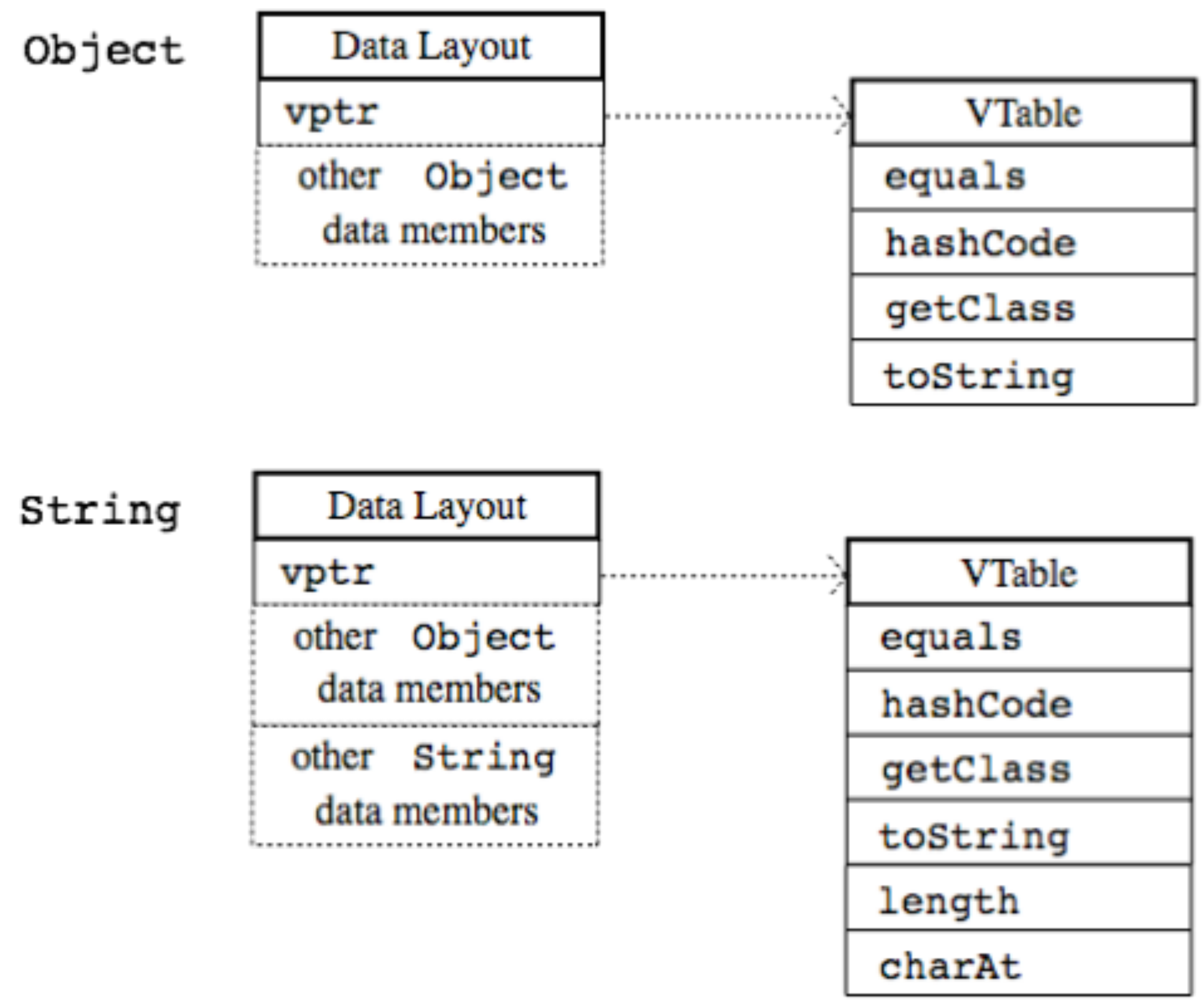
Data Layout of Class Inheritance in Java

- If we have data structures of variable sizes, how should we handle that in our layout?
- By pointer.. All objects in Java are passed by pointer.
- In other words, our data layout is a series of memory segments that store inline primitive types or pointers.



Method Layout of Class Inheritance

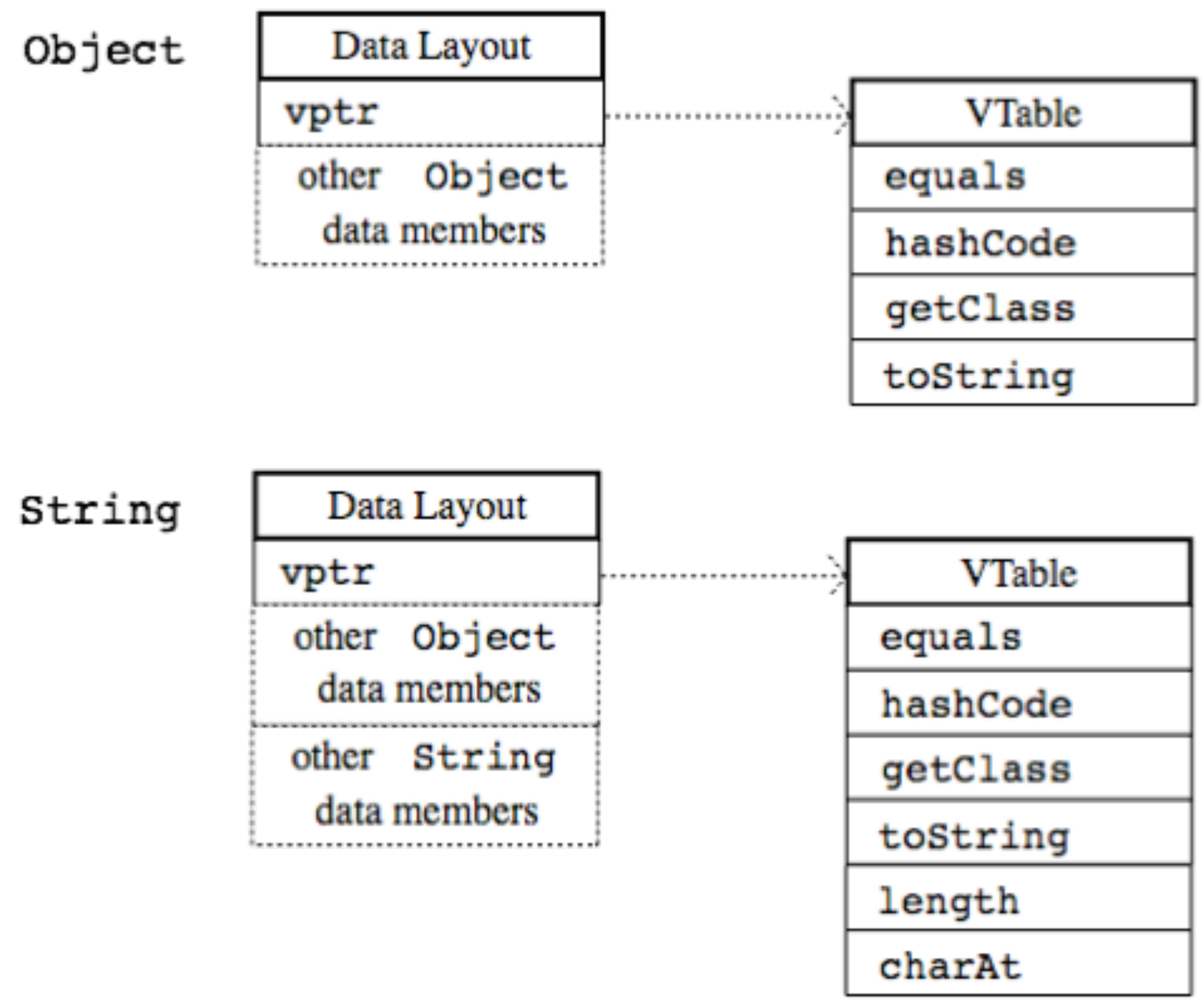
- For virtual methods we have a per ***class*** vtable containing pointers to the implementations for that **class**
- Every **object** has a vptr pointing to the one class vtable
- For example, at offset zero in the vtables we have the *equals* method
- New methods added by a subclass extend the vtable, adding new pointers at the end.
- Overriden methods go into an existing slot.



Method Layout of Class Inheritance

- Object and String have their own vtables, and String's vtable is an overriden clone of Object's vtable

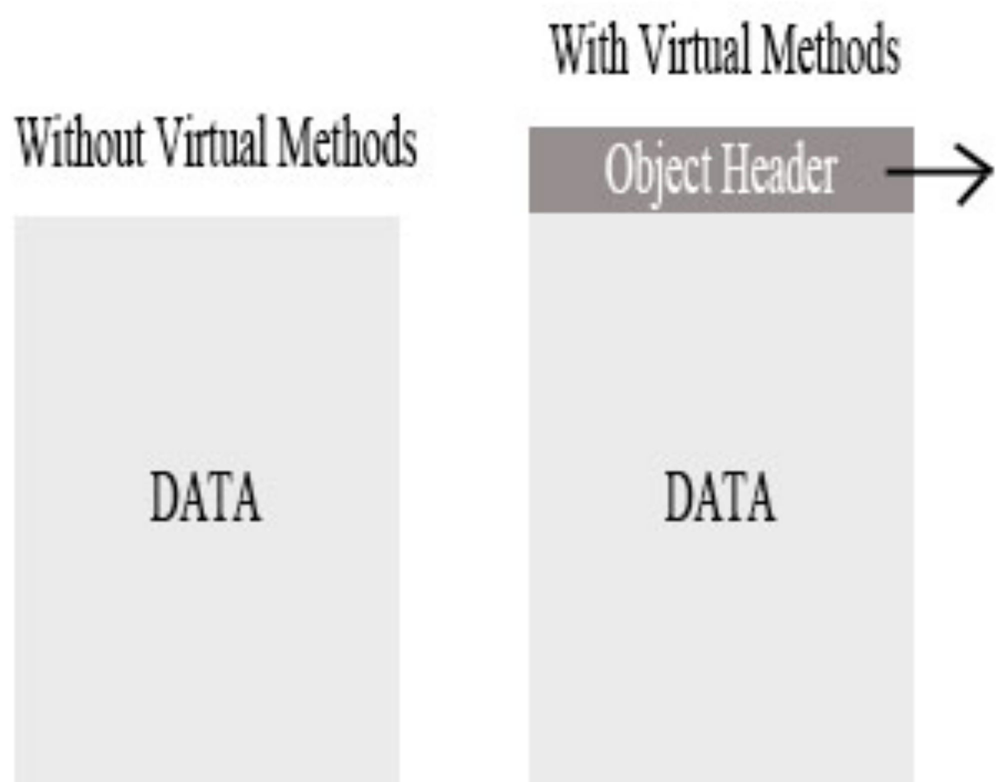
- In Java when you declare a subclass that extends a superclass, you clone the superclass' vtable and add the addresses of any new methods **that are not private or static** to the vtable



Method Layout of Class Inheritance

- The pointer required for an object in C++ to use virtual methods adds 8 bytes of space to the object, but any additional virtual methods will not further increase the size of the objects if we add more virtual methods,

- Moreover, once a class has a single virtual method the increase in size of the object in memory will not grow if more methods are added. (The vtable grows)



So lets implement this ourselves!

(https://github.com/nyu-oop/java-lang-1)

Specification

- From the site…
    - *Java includes a rather extensive set of platform libraries, with the facilities in the java.lang package being tightly integrated with Java's execution model. For our translator, we only support the hashCode(), equals(), and toString() methods in java.lang.Object and the printing of numbers and strings through out.print() and out.println() in java.lang.System. Also, some of the Class class.*
- We are implementing that portion of Java such that we can then translate the basics of the Java object model. I.E. java.lang.Object as the root of the class hierarchy

java::lang

- Everything you are about to see exists in Github and you will be using it in your translator
- However, you will be translating new classes in the target language in the same manner
- The challenge will be understanding it, you will do so via…

    1. This presentation
    2. Team exercise in class
    3. Independent study
    4. Piazza discussion
    5. Homework 3

Key Intuition

- There is a per-class vtable that represents the behavior of each class.
- That contract is necessary once for all instances.
- The behavior is mapped to an instance a pointer.

Java Type Declarations in C++

- We start by declaring types for our ‘Java’ classes with forward declarations.
- The double underscore prefix convention indicates types which are internal to java-lang.*
- typedefs are used to define the ‘Java’ type names as pointers to the internal structs.
- In this way the semantics of the type Object are the same as in java.lang.Object.

```cpp
// Forward declarations of
// data layout and vtables.
struct __Object;
struct __Object_VT;

struct __String;
struct __String_VT;

struct __Class;
struct __Class_VT;

// Definition of type names, which
// are equivalent to Java semantics,
// i.e., an instance is the address
// of the object's data layout.
typedef __Object* Object;
typedef __Class* Class;
typedef __String* String;
```

Data Layout of java.lang.Object in C++

- __Object data layout has a..
  - __vptr field which points to its virtual method table (L3)
  - methods: hashCode, equals, getClass, toString (L8-11)
  - __class method which returns the class object representing itself. (L15)
  - static __vtable field *for all instances* (L18)

```
1  struct __Object {
2    // The pointer to our vtable.
3    __Object_VT* __vptr;
4    // The constructor.
5    __Object();
6
7    // Methods implemented by java.lang.Object
8    static int32_t hashCode(Object);
9    static bool equals(Object, Object);
10   static Class getClass(Object);
11   static String toString(Object);
12
13   // The function returning the class
14   // object representing java.lang.Object
15   static Class __class();
16
17   // The vtable for java.lang.Object.
18   static __Object_VT __vtable;
19 };
```

Data Layout of java.lang.Object in C++

- Every instance of __Object has a *pointer* to the vtable, otherwise the whole premise of virtual methods would not work!
- We use static Class __class() function as opposed to declaring _class to avoid the “static initialization order fiasco”
- More on the fiasco here… https://isocpp.org/wiki/faq/ctors#static-init-order

```cpp
struct __Object {
  // The pointer to our vtable.
  __Object_VT* __vptr;
  // The constructor.
  __Object();

  // Methods implemented by java.lang.Object
  static int32_t hashCode(Object);
  static bool equals(Object, Object);
  static Class getClass(Object);
  static String toString(Object);

  // The function returning the class
  // object representing java.lang.Object
  static Class __class();

  // The vtable for java.lang.Object.
  static __Object_VT __vtable;
};
```

Vtable of java.lang.Object in C++

- __Object_VT has
  - an __isa property which points to its class. More on that soon. (L2)
  - member variables that are function pointers to the methods of java.lang.Object. (L3-6)
  - a “no-argument” constructor which inits function pointers with the *addresses* of __Object’s hashCode, equals, getClass and toString methods (L9-13)

```
1  struct __Object_VT {
2    Class __isa;
3    int32_t (*hashCode)(Object);
4    bool (*equals)(Object, Object);
5    Class (*getClass)(Object);
6    String (*toString)(Object);
7
8    __Object_VT()
9      : __isa(__Object::__class()),
10       hashCode(&__Object::hashCode),
11       equals(&__Object::equals),
12       getClass(&__Object::getClass),
13       toString(&__Object::toString) {}
14 };
```

Vtable of java.lang.Object in C++

- int32_t (*hashCode)(Object) is a pointer to a function that takes an Object as a parameter and returns an int32_t type.
- Notice how we use & to store an address in the no-argument constructor
- In Java there is an implicit argument for every instance method 'this'
- Therefore each vtable method has __Object as the first parameter.

```cpp
struct __Object_VT {
  Class __isa;
  int32_t (*hashCode)(Object);
  bool (*equals)(Object, Object);
  Class (*getClass)(Object);
  String (*toString)(Object);

  __Object_VT()
    : __isa(__Object::__class()),
      hashCode(&__Object::hashCode),
      equals(&__Object::equals),
      getClass(&__Object::getClass),
      toString(&__Object::toString) {}
};
```

Data Layout of java.lang.String in C++

- The data layout of java.lang.String in C++ is a clone of __Object except…
  - it adds data which uses the C++ std::string type
  - it adds a length and charAt method.

```
struct __String {
  // The vtable pointer
  __String_VT* __vptr;
  // Additional data
  std::string data;

  // The constructor
  __String(std::string data);

  // The methods implemented by
  // java.lang.String.
  static int32_t hashCode(String);
  static bool equals(String, Object);
  static String toString(String);
  static int32_t length(String);
  static char charAt(String, int32_t);

  // The function returning the class
  // object representing java.lang.String.
  static Class __class();

  // The vtable for java.lang.String.
  static __String_VT __vtable;
};
```

Vtable of java.lang.String in C++

- Again, similar to Object with some additions..

- The type of the first parameter for __Object's getClass and __String's getClass differ (the implicit this), so we need a cast.

```
struct __String_VT {
    Class __isa;
    int32_t (*hashCode)(String);
    bool (*equals)(String, Object);
    Class (*getClass)(String);
    String (*toString)(String);
    int32_t (*length)(String);
    char (*charAt)(String, int32_t);

    __String_VT()
    : __isa(__String::__class()),
      hashCode(&__String::hashCode),
      equals(&__String::equals),
      getClass((Class(*)(String))&__Object::getClass),
      toString(&__String::toString),
      length(&__String::length),
      charAt(&__String::charAt) {
    }
};
```

Data Layout of java.lang.Class in C++

- __Class
    - a name field to denote the classname (L3)
    - a parent field to reference the parent class. (L4)
    - The latter is used to implement the getSuperclass method, which return's a reference to an object’s superclass.
- __Class is important because without it we would not be able to track the dynamic type of objects.
- __Class is what links objects in the inheritance hierarchy.

```
1  struct __Class {
2    __Class_VT* __vptr;
3    String name;
4    Class parent;
5
6    // The constructor.
7    __Class(String name, Class parent);
8
9    // The instance methods of java.lang.Class.
10   static String toString(Class);
11   static String getName(Class);
12   static Class getSuperclass(Class);
13   static bool isInstance(Class, Object);
14
15   // The function returning the class
16   // object representing java.lang.Class.
17   static Class __class();
18
19   // The vtable for java.lang.Class.
20   static __Class_VT __vtable;
21 };
```

Odds & Ends

- The __rt namespace contains
    - a null value function
    - a String literal function to convert a C string to a java::lang::String
    - This is instead of letting C++ implicitly convert the C string to a std::string

```
namespace __rt {
  // Function returning the canonical null value.
  java::lang::Object null();

  // Function for converting a C string literal
  // to a translated Java string.
  inline java::lang::String literal(const char * s){
    // C++ implicitly converts the C string
    // to a std::string.
    return new java::lang::__String(s);
  }
}
```

Implementations

- We’ve just looked at the declarations (i.e. .h files) for all our Java primitives in C++

- There are of course implementations for all these definitions, again in https://github.com/nyu-oop/java-lang-1

- You’ll get familiar with them in an in-class exercise.

Adding a new class

- Ex. class B, which inherits from A, which inherits from Object, and we want to do the data layout for B.
    - B, by definition of inheritance, has all the same fields and data as A.
    - Therefore the data layout for B consists of the data layout for A, and then the new data of B appended to that.
    - Similarly, the data layout for A consists of the data layout for Object, and the new data of A appended.

Key Idea

- The vtable for Object has the __isa pointer and the pointers to the four methods we need – that's the contract of __Object
- If we override a method with a sublcass, there exists a slot in the vtable for it already.
- That is, overriding of virtual methods is implemented by replacing a pointer in the vtable.

- You’ll need to study the code.
  - https://github.com/nyu-oop/java-lang-1)
- You’ll have a homework and an in-class exercise to help you.