Topics

- Support for Object-Oriented Programming in Java
- Support for Object-Oriented Programming in C#
- Support for Object-Oriented Programming in Ruby
- Implementation of Object-Oriented Constructs
- Reflection

Support for OOP in Java

- Because of its close relationship to C++, focus is on the differences from that language
- General Characteristics
  - All data are objects except the primitive types
  - All primitive types have wrapper classes that store one data value
  - All objects are heap-dynamic, are referenced through reference variables, and they are allocated with **`new`**
  - A **`finalize`** method is implicitly called when the garbage collector is about to reclaim the storage occupied by the object

Support for OOP in Java (continued)

- Inheritance
  - Java supports single inheritance only, but it can also implement one or more **`interfaces`**
  - An interface can include only method declarations and named constants, e.g.,

```
public interface Comparable <T> {
      public int comparedTo (T b);
}
```

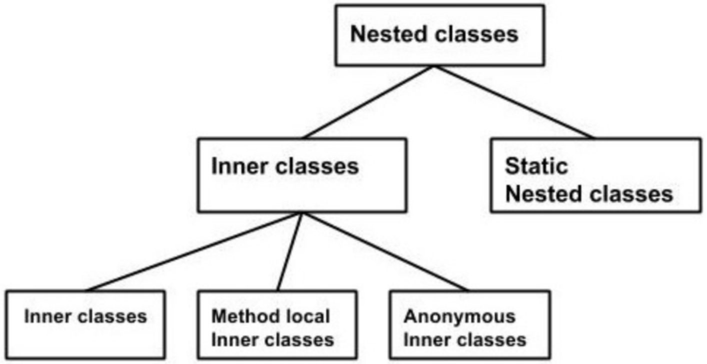
  - Methods can be **`final`** (cannot be overriden)
  - All subclasses could be subtypes

Support for OOP in Java (continued)

- Dynamic Binding
  - In Java, all messages are dynamically bound to methods, unless the method is `final` (i.e., it cannot be overriden, therefore dynamic binding serves no purpose)
  - Static binding is also used if the methods is `static` or `private` both of which disallow overriding

Support for OOP in Java (continued)

- Nested Classes
  - All are hidden from all classes in their package, except for the nesting class
  - Nonstatic nested classes are called *innerclasses*
    - An innerclass can access members of its nesting class
  - Nested classes can be anonymous
  - A *local nested class* is defined in a method of its nesting class
  - Static nested classes



Support for OOP in Java (continued)

- Evaluation
  - Design decisions to support OOP are similar to C++
  - No support for procedural programming
  - No parentless classes
  - Dynamic binding is used as “normal” way to bind method calls to method definitions
  - Uses interfaces to provide a simple form of support for multiple inheritance

Support for OOP in C#

- General characteristics
  - Support for OOP similar to Java
  - Includes both classes and `struct`s
  - Classes are similar to Java's classes
  - `struct`s are less powerful stack-dynamic constructs (e.g., no inheritance)

Support for OOP in C# (continued)

- Inheritance
  - Uses the syntax of C++ for defining classes
  - A method inherited from parent class can be replaced in the derived class by marking its definition with **`new`**
  - The parent class version can still be called explicitly with the prefix **`base`**:

    ```
    base.Draw()
    ```

  - Subclasses are subtypes if no members of the parent class is private
  - Single inheritance only

Support for OOP in C#

- Dynamic binding
  - To allow dynamic binding of method calls to methods:
    - The base class method is marked **`virtual`**
    - The corresponding methods in derived classes are marked **`override`**
  - Abstract methods are marked **`abstract`** and must be implemented in all subclasses
  - All C# classes are ultimately derived from a single root class, `Object`

Support for OOP in C#

- Evaluation
  - C# is a relatively recently designed C-based OO language
  - The differences between C#’s and Java’s support for OOP are relatively minor

Support for OOP in Ruby

- General Characteristics
  - Everything is an object
  - All computation is through message passing
  - Class definitions are executable, allowing secondary definitions to add members to existing definitions
  - Method definitions are also executable
  - All variables are type-less references to objects
  - Access control is different for data and methods
    - It is private for all data and cannot be changed
    - Methods can be either public, private, or protected
  - Getters and setters can be defined by shortcuts

Support for OOP in Ruby (continued)

- Inheritance
  - Access control to inherited methods can be different than in the parent class
  - Subclasses are not subtypes
- Dynamic Binding
  - All variables are typeless and polymorphic
- Evaluation
  - Does not support abstract classes
  - Does not fully support multiple inheritance(mixin)
  - Access controls are weaker than those of other languages that support OOP

Ruby mixin

```ruby
module A
    def a1
    end
    def a2
    end
end
```

```ruby
module B
    def b1
    end
    def b2
    end
end
```

```ruby
class Sample
include A
include B
    def s1
    end
end
```

```ruby
samp = Sample.new
samp.a1
samp.a2
samp.b1
samp.b2
samp.s1
```

Implementing OO Constructs

- Two interesting and challenging parts
  - Storage structures for instance variables
  - Dynamic binding of messages to methods

Instance Data Storage

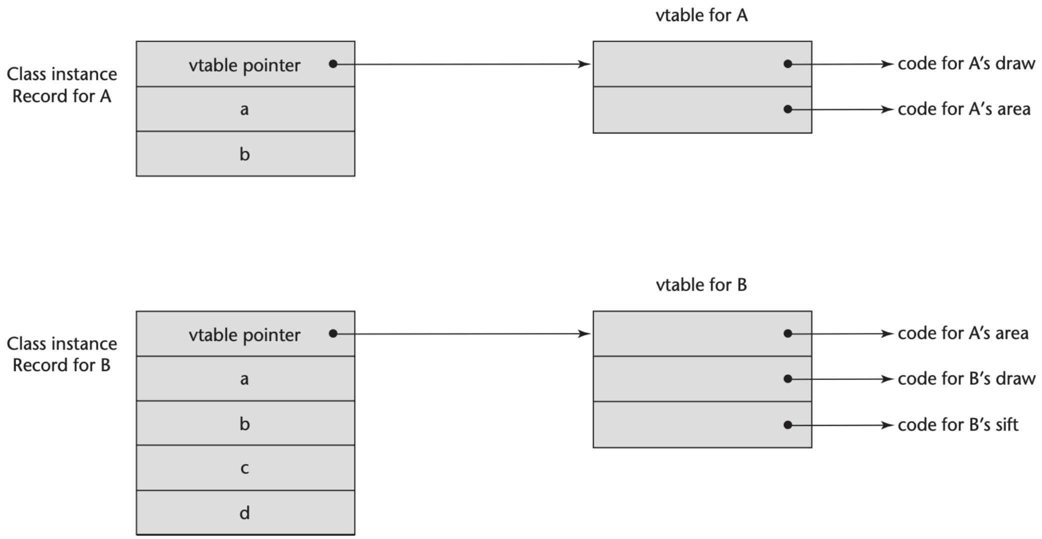
- Class instance records (CIRs) store the state of an object
  - Static (built at compile time)
- If a class has a parent, the subclass instance variables are added to the parent CIR
- Because CIR is static, access to all instance variables is done as it is in records
  - Efficient

Dynamic Binding of Methods Calls

- Methods in a class that are statically bound need not be involved in the CIR; methods that will be dynamically bound must have entries in the CIR
  - Calls to dynamically bound methods can be connected to the corresponding code thru a pointer in the CIR
  - The storage structure is sometimes called *virtual method tables* (vtable)
  - Method calls can be represented as offsets from the beginning of the vtable

An example of CIRs

```
public class A {
  public int a, b;
  public void draw() { . . . }
  public int area() { . . . }
}
public class B extends  A {
  public int c, d;
  public void draw() { . . . }
  public void sift() { . . . }
}
```



Reflection

- A programming language that supports reflection allows its programs to have runtime access to their types and structure and to be able to dynamically modify their behavior
- The types and structure of a program are called *metadata*
- The process of a program examining its metadata is called *introspection*
- Interceding in the execution of a program is called *intercession*

Reflection (continued)

- *Uses of reflection for software tools*:
  - Class browsers need to enumerate the classes of a program
  - Visual IDEs use type information to assist the developer in building type-correct code
  - Debuggers need to examine private fields and methods of classes
  - Test systems need to know all of the methods of a class

Reflection in Java

- Limited support from `java.lang.Class`
- Java runtime instantiates an instance of `Class` for each object in the program
- The `getClass` method of `Class` returns the `Class` object of an object

```
 float[] totals = new float[100];
Class fltlist = totals.getClass();
Class stg = "hello".getClass();
```

- If there is no object, use `class` field

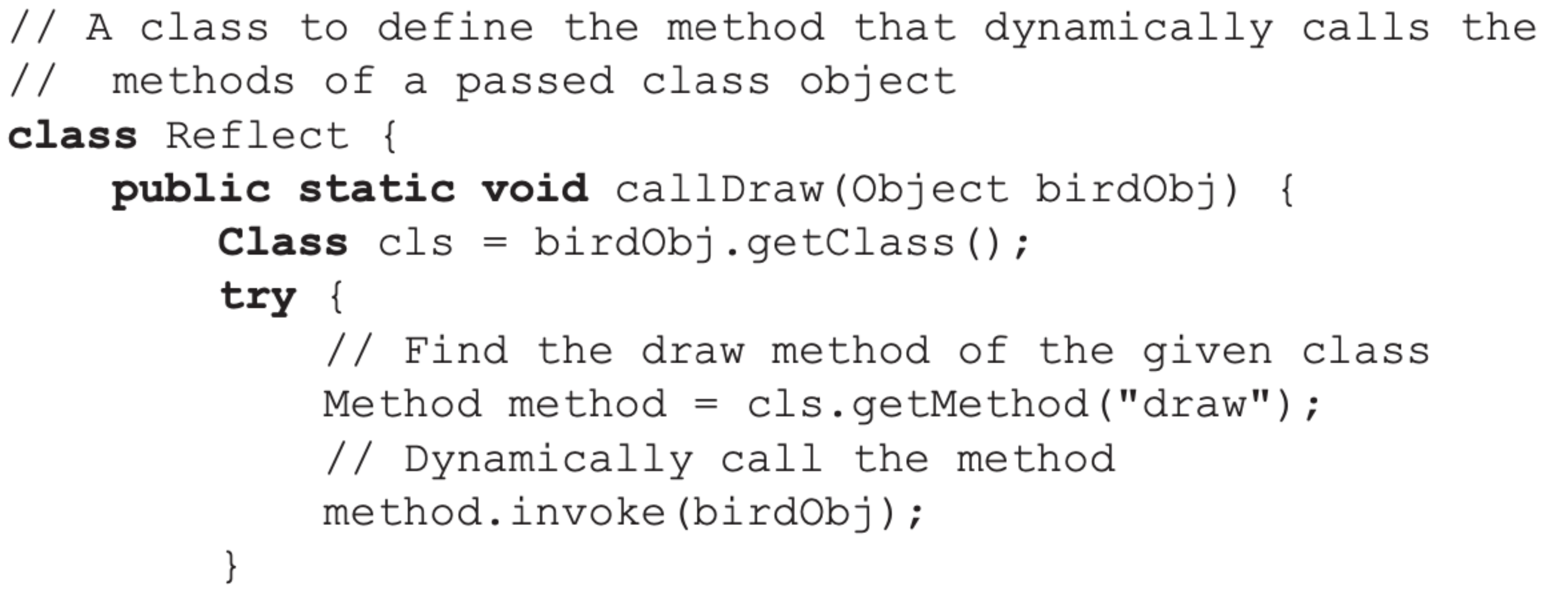
```
Class stg = String.class;
```

Reflection in Java (continued)

- `Class` *has four useful methods:*
- `getMethod` searches for a specific public method of a class
- `getMethods` returns an array of all public methods of a class
- `getDeclaredMethod` searches for a specific method of a class
- `getDeclaredMethods` returns an array of all methods of a class

Reflection in Java (continued)

- The `Method` class defines the invoke method, which is used to execute the method found by `getMethod`
- `method.invoke(obj, args)`

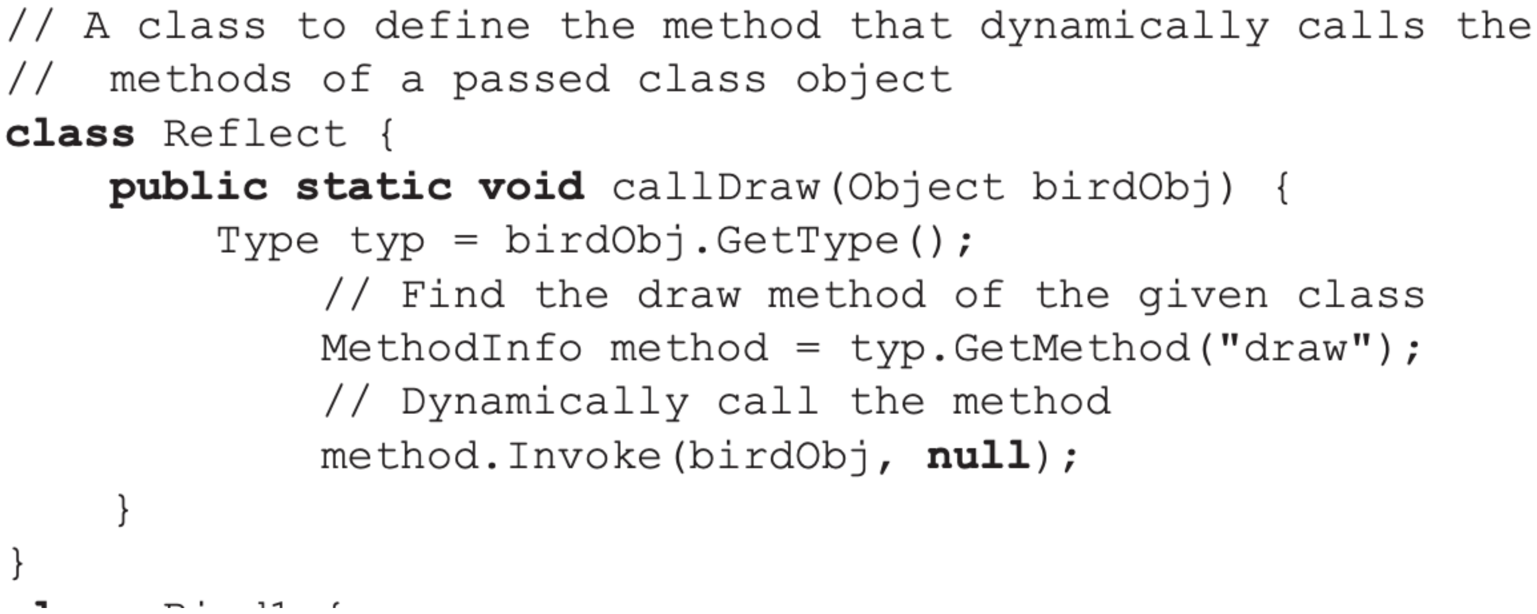


```
// A class to define the method that dynamically calls the
//  methods of a passed class object
class Reflect {
    public static void callDraw(Object birdObj) {
        Class cls = birdObj.getClass();
        try {
            // Find the draw method of the given class
            Method method = cls.getMethod("draw");
            // Dynamically call the method
            method.invoke(birdObj);
        }
```

Reflection in C#

- In the .NET languages the compiler places the intermediate code in an assembly, along with metadata about the program
- `System.Type` is the namespace for reflection
- `getType` is used instead of `getClass`
- `typeof` operator is used instead of `.class` field
- `System.Reflection.Emit` namespace provides the ability to create intermediate code and put it in an assembly (Java does not provide this capability)

Example of Reflection in C#



```
// A class to define the method that dynamically calls the
//  methods of a passed class object
class Reflect {
    public static void callDraw(Object birdObj) {
        Type typ = birdObj.GetType();
            // Find the draw method of the given class
            MethodInfo method = typ.GetMethod("draw");
            // Dynamically call the method
            method.Invoke(birdObj, null);
    }
}
class Bird1 {
```

Downsides of Reflection

- Performance costs
- Exposes private fields and methods
- Voids the advantages of early type checking
- Some reflection code may not run under a security manager, making code nonportable