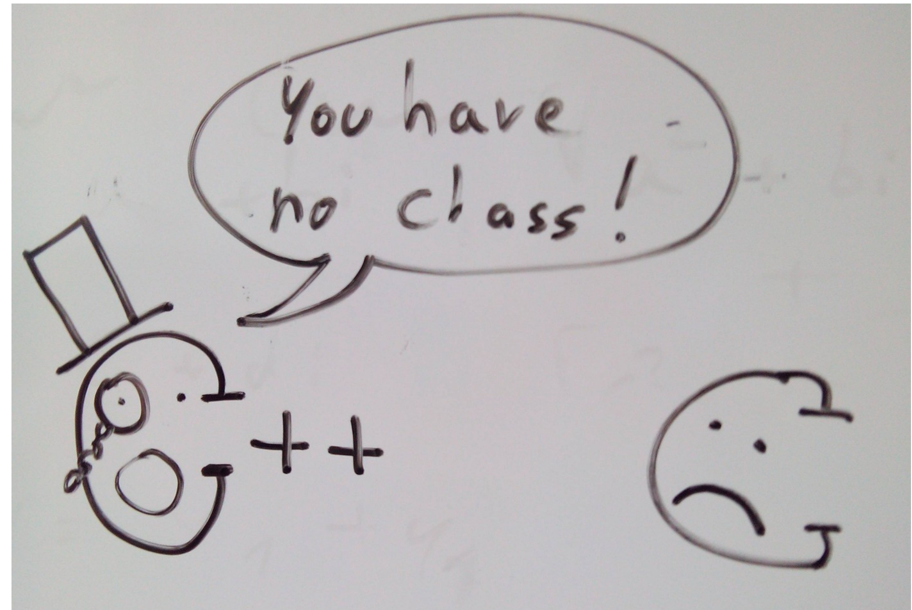
CS 430
Spring 2022

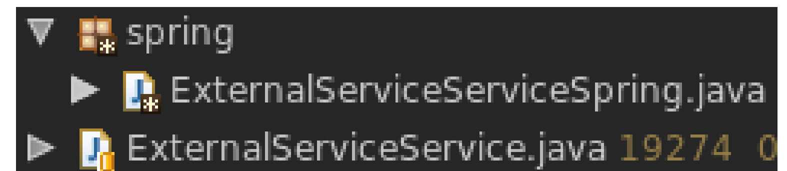
Mike Lam, Professor



Abstraction and Object-Oriented Programming

Abstraction

- Abstraction is a fundamental concept in CS
- Textbook definition: "*a view or representation of an entity that includes only the most significant attributes*"
- Mathematical notion: "*equivalence classes*"
- Practical reality: the first line of defense against software complexity!
- Key: finding the most appropriate level of abstraction



org.apache.xmlrpc.server

Interface RequestProcessorFactoryFactory

All Known Implementing Classes:

RequestProcessorFactoryFactory.RequestSpecificProcessorFactoryFactory, RequestProcessorFactoryFactory.StatelessProcessorFactoryFactory

Types of abstraction

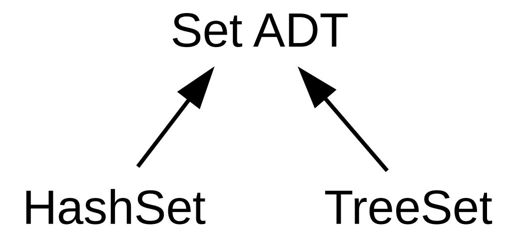
- Process/procedure abstraction
  - Structured (block) syntax
  - Subprograms, methods
- Data abstraction
  - Abstract data types, classes, and interfaces
  - Polymorphism and generics
- Grouping abstraction
  - Packages, namespaces, classes, modules

Abstract data types

- Abstract data type (ADT)
  - Set of values (carrier set)
  - List of supported operations
    - Common operations: constructor, accessors, iterators, destructors
  - Not specified: underlying representation
    - Exists purely as a mathematical construct
- Examples
  - List: `append(value), get(index), remove(index)`
  - Stack: `push(value), pop`
  - Set: `add(value), isMember(value), union(otherSet)`
  - Map: `store(key, value), lookup(key)`
  - Floating-point: `add, sub, mul, div, sqrt`

Abstract data types

- Concrete data type
  - Implementation of an ADT on a computer
  - Specifies value size and format
  - Often supports only a subset of values from the ADT
  - Most languages support user-defined concrete data types
- Examples (in Java)
  - List: `ArrayList`, `LinkedList`
  - Set: `HashSet`, `TreeSet`
  - Floating-point: `float`, `double`



Abstract data types

- Abstract data types can be implemented in **some** programming languages as data types
  - Basic requirement: encapsulation mechanisms
    - Grouping of related code and data
  - Easier w/ information hiding mechanisms
    - Keeping implementation details private or inaccessible
  - Information hiding implies encapsulation (but not converse)
  - Even easier w/ some form of polymorphism

Design issues

- Encapsulation: grouping of related code and data
  - Header files, namespaces, packages, modules, etc.
  - Structs, unions, classes, interfaces
  - Writability/readability vs. extensibility and maintainability
- Information hiding: keeping implementation details hidden
  - Levels: public, private, protected
  - Public fields vs. getters and setters
  - Convenience/writability vs. safety and extensibility
- Polymorphism: allowing parameterization by data type
  - Specifying parameters
  - Specifying restrictions on the parameters
  - Readability vs. power/expressivity

Encapsulation

- Advantages
  - Organization
  - Separate compilation
  - Avoiding name collisions

- Physical vs. logical encapsulation
  - Contiguous (e.g., single file) vs. non-contiguous code
- Naming vs. non-naming encapsulation
  - Referenced by name vs. not
- Grouping-only vs. information hiding encapsulation
  - Everything public vs. some things non-public

Encapsulation

	Physical	Logical
Naming	Java Class Java Package	Ruby Class Ada Package C++ Namespace Ruby Module
Non-naming	.c, .cpp, or .h file	
Grouping only	.h file	Ruby Module C++ Namespace
Information hiding	.c or .cpp file Java Class Java Package	Ruby Class Ada Package

Original table courtesy of Dr. Chris Fox

Object-oriented programming

- Primary advantage: inheritance
  - Original motivation: code re-use
  - Parent/superclass vs. child/derived/subclass
  - “Pure” (everything is an object) vs. hybrid
  - Overridable vs. non-overridable methods
  - Abstract methods and classes
  - Single vs. multiple inheritance (simplicity vs. power)
  - Static vs. dynamic dispatch (speed vs. power)

Dispatch

```
class DispatchTest1
{
    void foo(Object o) { System.out.println("foo(Object)"); }
    void foo(String s) { System.out.println("foo(String)"); }
    void bar(Object o) {
        foo(o);
    }
    public static void main(String[] args) {
        (new DispatchTest1()).bar("What gets run?");
    }
}
```

What will this program print?

Dispatch

```
class DispatchTest1
{
    void foo(Object o) { System.out.println("foo(Object)"); }
    void foo(String s) { System.out.println("foo(String)"); }
    void bar(Object o) {
        foo(o);
    }
    public static void main(String[] args) {
        (new DispatchTest1()).bar("What gets run?");
    }
}

class DispatchTest2
{
    static class A {
        void foo() { System.out.println("A.foo()"); }
    }
    static class B extends A {
        void foo() { System.out.println("B.foo()"); }
    }
    void bar(A a) {
        a.foo();
    }
    public static void main(String[] args) {
        (new DispatchTest2()).bar(new B());
    }
}
```

What about this one?

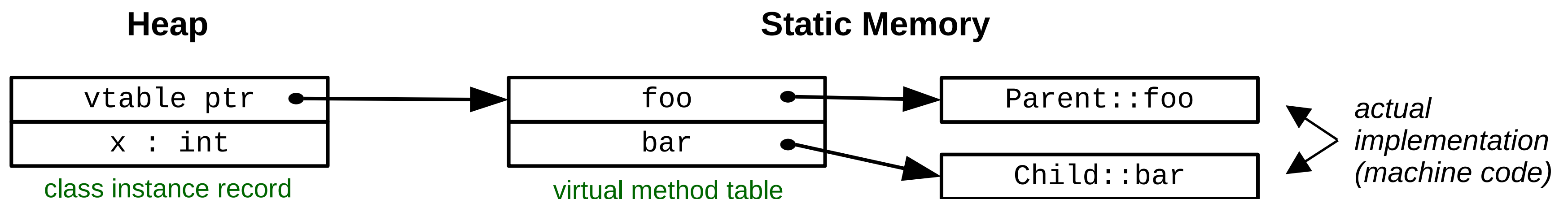
Dispatch

```
class DispatchTest1
{
    void foo(Object o) { System.out.println("foo(Object)"); }
    void foo(String s) { System.out.println("foo(String)"); }
    void bar(Object o) {
        foo(o);
    }
    public static void main(String[] args) {
        (new DispatchTest1()).bar("What gets run?");
    }
}

class DispatchTest3
{
    static class A {
        static void foo() { System.out.println("A.foo()"); }
    }
    static class B extends A {
        static void foo() { System.out.println("B.foo()"); }
    }
    void bar(A a) {
        a.foo();
    }
    public static void main(String[] args) {
        (new DispatchTest3()).bar(new B());
    }
}
```

How about now?

Dynamic dispatch

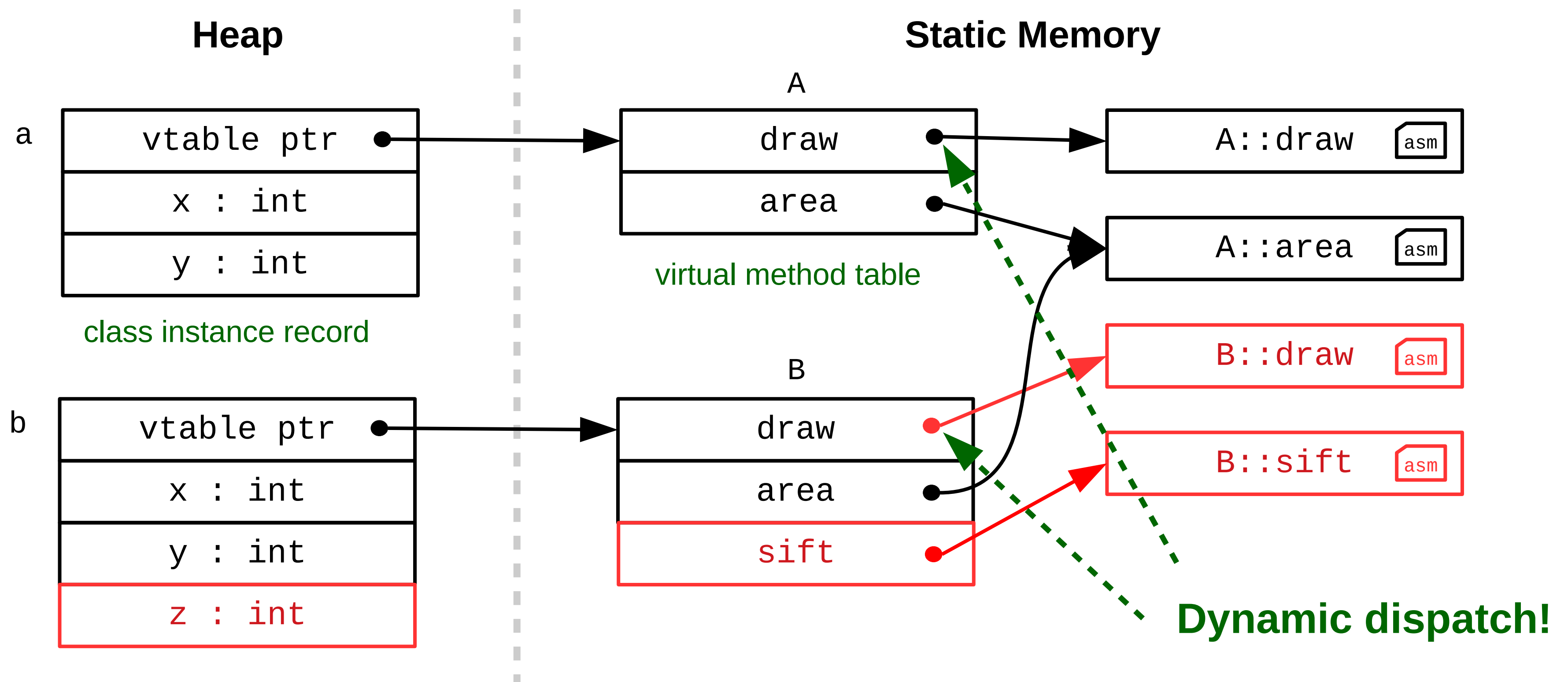
- Dispatch
  - Static dispatch: all method calls can be resolved at compile time
  - Dynamic dispatch: polymorphic method calls resolved at run time
  - Single vs. multiple dispatch (one object’s type vs. multiple objects’ type)
- Class instance record – one per object
  - Storage for object member variables and vtable pointer
  - Subclass CIR is a copy of the parents' with (potentially) added fields
- Virtual method table (*vtable*) – one per class
  - List of virtual methods w/ pointers to implementations



Dynamic dispatch

```
public class A {
    public int x, y;
    public void draw() { … }
    public int area() { … }
}

a = new A();
```

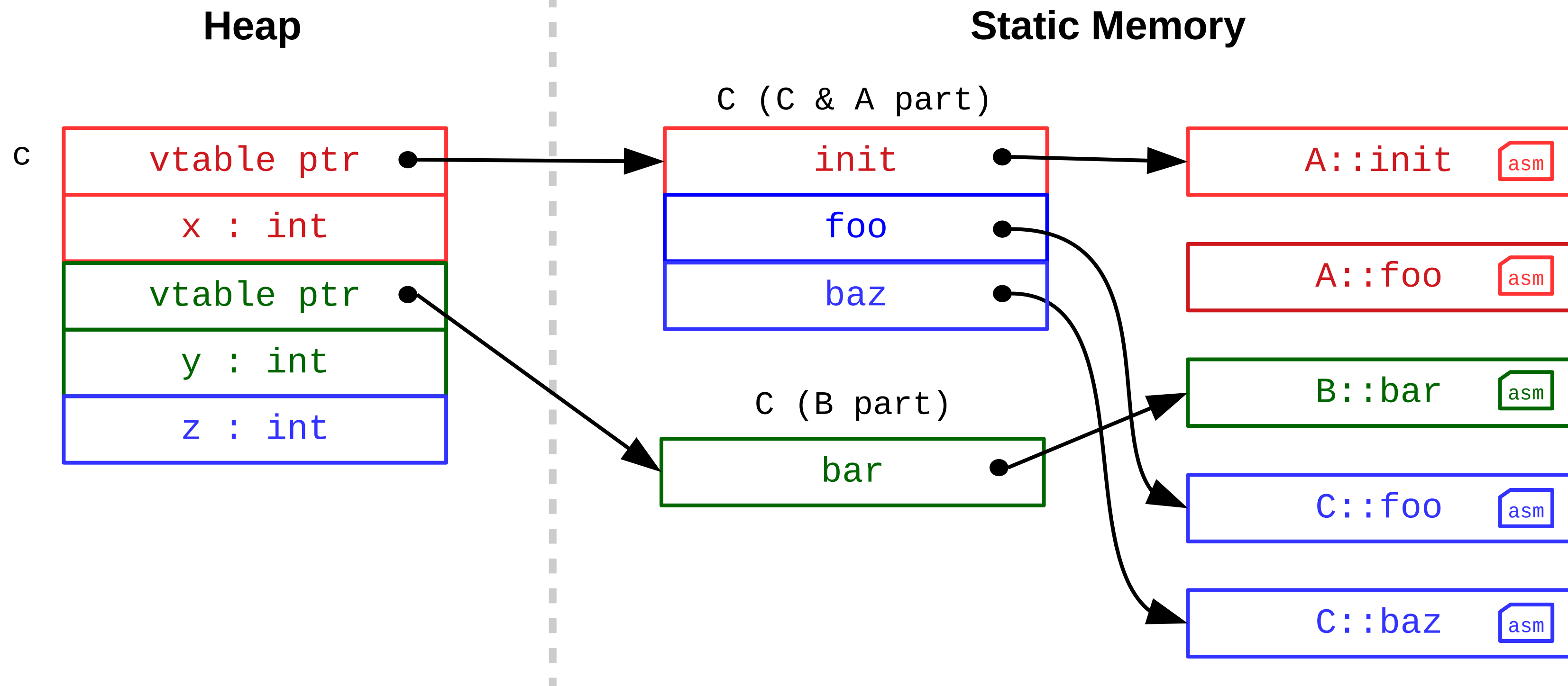
```
public class B extends A {
    public int z;
    public void draw() { … }
    public void sift() { … }
}

b = new B();
```



Multiple inheritance

```
class A {
  public:
    int x;
    virtual void init() { … }
    virtual void foo() { … }
}
class B {
  public:
    int y;
    virtual void bar { … }
}
```

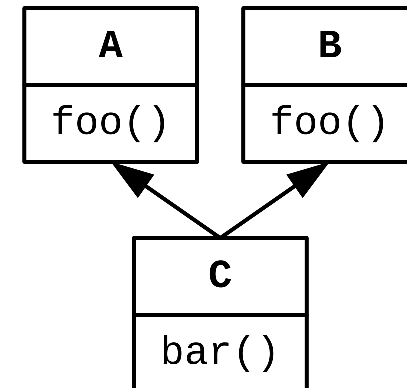
```
class C : public A, public B {
  public:
    int z;
    virtual void foo() { … }
    virtual void baz() { … }
}

c = new C();
```



Multiple inheritance

- Ambiguity problem
  - If C inherits from A and B, both of which implement method "foo," which is called by default from C?

```
class C : public A, B {
  public:
    void bar() {
        foo();  // which foo?
    }
};
```



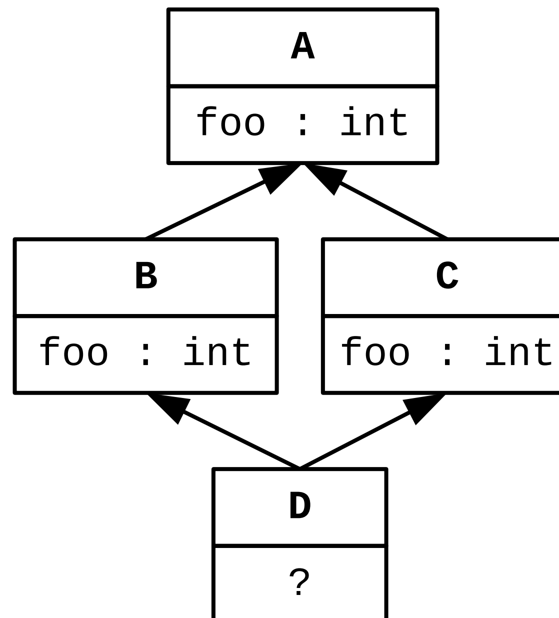
C++ solution: require fully-qualified names
(e.g., A`::foo()` or B`::foo()`)

Java solution #1: only implement multiple interfaces
(so **no** implementation of foo here)

Java solution #2: compiler error
(Java 8 adds default methods for interfaces)

Multiple inheritance

- Diamond problem
  - If D inherits from B and C with common ancestor A, and B and C both inherit field “foo”, does D have one copy or two?
  - Even if resolved manually, this increases the complexity of the dependencies among classes



Inheritance and the stack

- How to handle class instance records on stack in case of copying to a superclass variable?
  - No space for subclass data
  - Object slicing: remove subclass data
  - Causes a loss of information!

```
class A { int x; }
class B inherits A { int y; }

B b = new B();
A a = b;            // copy; A's CIR is smaller
                    // b.y is lost!
                    // no way to cast a back
```

Templates vs. generics

- Templates (C++)
  - Compiles different versions w/ mangled names
- Generics (Java)
  - Type erasure: compiler changes generic type to Object and inserts runtime casts (expensive!)
  - No runtime difference between `HashSet<String>` and `HashSet<Integer>`
    - Example: no arrays of generics (array members must be type-checked at runtime)
  - Only one set of static member data

```
template <class T>
class Foo {
    T data;
  public:
    void bar(T x) {
        this.data = x;
    }
}
```

Templates in C++

```
class Foo<T> {
    T data;
    void bar(T x) {
        this.data = x;
    }
}
```

Generics in Java

Reflection

- A language with reflection provides runtime access to type and structure metadata
  - Sometimes with the ability to **modify** the structure
  - Often incurs a severe runtime penalty because of data structures required
- Examples:
  - Ruby: `methods` and `send`
  - Java: `java.lang.Class` and `java.lang.reflect.Method`

```
"Hello".send(
  "str".methods
       .grep(/upcase/)[0])
```

Reflection in Ruby

```
try {
    System.out.println("str".getClass()
                            .getMethod("toUpperCase")
                            .invoke("Hello"));
}
catch (NoSuchMethodException ex)     {}
catch (IllegalAccessException ex)    {}
catch (InvocationTargetException ex) {}
```

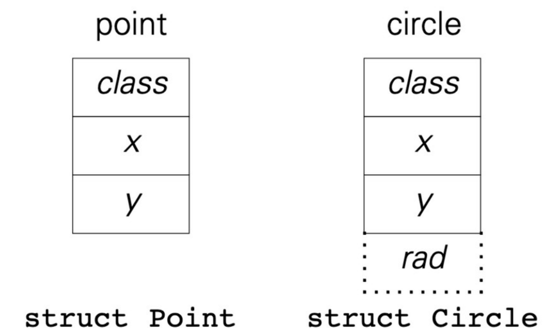
Reflection in Java

History of OOP

- Simula (1967): data abstractions for simulation and modeling
- Smalltalk (1980): objects and messages
- C++ (1985): originally “C with classes”
- Java (1995) and C# (2000): goal was “C++ but better”
- Ruby (1996): pure, dynamic OOP language
- Most modern languages have some form of OOP
  - Abstract data types
  - Inheritance
  - Dynamic binding

Abstraction in C

- Structs for encapsulation
- `void*` pointers for polymorphism and inheritance
- Stack or heap allocation (manual management)
- Header file and implementation file
- Function pointers for dynamic dispatch
- No reflection by default




```
struct Class {
    size_t size;
    void * (* ctor) (void * self, va_list * app);
    void * (* dtor) (void * self);
    void * (* clone) (const void * self);
    int (* differ) (const void * self, const void * b);
};
struct String {
    const void * class; /* must be first */
    char * text;
};
```

```
struct Point {
    const void * class;
    int x, y;                /* coordinates */
};

struct Circle { const struct Point _; int rad; };
```

https://www.cs.rit.edu/~ats/books/ooc.pdf

Abstraction in C++

- Classes and structs
- Stack or heap allocation
- Manual memory management: constructors and destructors
  - “Resource acquisition is initialization” (RAII)
- Header file and implementation file
- Visibility: public (default for structs) or private (default for classes)
  - "Friend" functions for private access outside class
- All forms of polymorphism (parametric via **templates**)
- Static dispatch by default (override via “`virtual`” keyword)
- Multiple inheritance w/ resolution via inheritance order
- Namespaces for naming and encapsulation
- No reflection by default

Abstraction in Java

- Classes similar to C++
- Single inheritance tree (rooted at Object)
- No stack allocation (everything on heap)
- Automatic memory management
- Visibility modifiers required (`public`, `private`, `protected`, `package`)
- No separate header file
- All forms of polymorphism (parametric via **generics**)
- Dynamic dispatch by default (override via "`static`" keyword)
- Interfaces for pseudo-multiple inheritance
- Packages for naming and encapsulation
- Reflection via `java.lang.reflect` package

Abstraction in Ruby

- “Pure” OOP: everything is an object!
- Dynamic classes
- Members can be added/removed at run time
- Multiple definitions of a single class allowed
- Keywords for function visibility (public by default)
- All data is private
  - "@" symbol for instance variables
  - Attributes accessed through methods
- Polymorphism and dispatch via dynamic types; no overloading
  - “Duck” typing: if it has the required methods, it’s a valid parameter
- Modules for encapsulation and multiple inheritance (mixins)
- Built-in reflection

Language comparison

Table 12.1 *Designs*

DESIGN ISSUE/ LANGUAGE	SMALLTALK	C++	OBJECTIVE-C	JAVA	C#	RUBY
Exclusivity of objects	All data are objects	Primitive types plus objects	Primitive types plus objects	Primitive types plus objects	Primitive types plus objects	All data are objects
Are subclasses subtypes?	They can be and usually are	They can be and usually are if the derivation is public	They can be and usually are	They can be and usually are	They can be and usually are	No subclasses are subtypes
Single and multiple inheritance	Single only	Both	Single only, but some effects with protocols	Single only, but some effects with interfaces	Single only, but some effects with interfaces	Single only, but some effects with modules
Allocation and deallocation of objects	All objects are heap allocated; allocation is explicit and deallocation is implicit	Objects can be static, stack dynamic, or heap dynamic; allocation and deallocation are explicit	All objects are heap dynamic; allocation is explicit and deallocation is implicit	All objects are heap dynamic; allocation is explicit and deallocation is implicit	All objects are heap dynamic; allocation is explicit and deallocation is implicit	All objects are heap dynamic; allocation is explicit and deallocation is implicit
Dynamic and static binding	All method bindings are dynamic	Method binding can be either	Method binding can be either	Method binding can be either	Method binding can be either	All method bindings are dynamic
Nested classes?	No	Yes	No	Yes	Yes	Yes
Initialization	Constructors must be explicitly called	Constructors are implicitly called	Constructors must be explicitly called	Constructors are implicitly called	Constructors are implicitly called	Constructors are implicitly called