




Chapter 3: Abstract Data Type (ADT) and Object-Oriented Programming (OOP)

3.4 Object-Oriented Programming (OOP)


Ming Liu

March 25, 2019

Outline

- 
1. **Criteria of Object-Orientation**
 2. **Basic concepts: object, class, attribute, method, and interface**
 3. **Distinct features of OOP**
 - Encapsulation and information hiding
 - Inheritance and overriding
 - Polymorphism, subtyping and overloading
 - Static and Dynamic dispatch
 - Composition and delegation
 4. **Some important Object methods in Java**
 5. **To write an immutable class**
 6. **History of OOP**
 7. **Summary**

Objective of this lecture

- 
- **Separating the interface of an abstract data type from its implementation, and using Java interface types to enforce that separation.**
 - **Define ADTs with interfaces, and write classes that implement interfaces.**



1 Criteria of Object-Orientation



Criteria of Object-Orientation

- An OO programming method / language should have the notion of **class** as the central concept.
- The language should make it possible to equip a class and its features **with assertions (preconditions, postconditions and invariants) and exceptional handling**, relying on tools to produce documentation out of these assertions and, optionally, monitor them at run time. **⇒ADT**
 - They help produce reliable software;
 - They provide systematic documentation;
 - They are a central tool for testing and debugging object-oriented software.
- **Static typing:** A well-defined type system should, by enforcing a number of type declaration and compatibility rules, guarantee the run-time type safety of the systems it accepts.

Criteria of Object-Orientation

- **Genericity for “ready for change” and “design for/with reuse”:** It should be possible to write classes with formal generic parameters representing arbitrary types.
- **Inheritance:** It should be possible to define a class as inheriting from another, to control the resulting potential complexity.
- **Polymorphism:** It should be possible to attach entities (names in the software texts representing run-time objects) to run-time objects of various possible types, under the control of the inheritance-based type system.
- **Dynamic dispatch/binding:** Calling a feature on an entity should always trigger the feature corresponding to the type of the attached run-time object, which is not necessarily the same in different executions of the call.



2 Basic concepts: object, class, attribute, and method



Object

- Real-world objects share two characteristics: they all have *state* and *behavior*. Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.
 - Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail).
 - Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes).
- For each object that you see, ask yourself two questions, and these real-world observations all translate into the world of object-oriented programming.
 - What possible states can this object be in?
 - What possible behavior can this object perform?

Object

- An object is a bundle of *state* and *behavior*
- **State – the data contained in the object.**
 - In Java, these are the fields of the object
- **Behavior – the actions supported by the object**
 - In Java, these are called methods
 - Method is just OO-speak for function
 - invoke a method = call a function

Classes

- **Every object has a class**
 - A class defines methods and fields
 - Methods and fields collectively known as members
- **Class defines both type and implementation**
 - type \approx where the object can be used
 - implementation \approx how the object does things
- **Loosely speaking, the methods of a class are its Application Programming Interface (API)**
 - Defines how users interact with instances

Class example – complex numbers

```

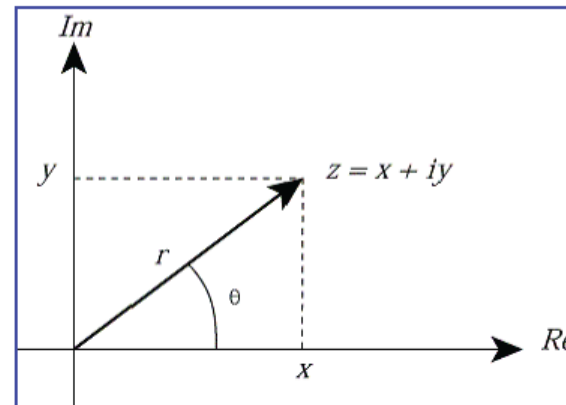
class Complex {
    private double re; // Real Part
    private double im; // Imaginary Part

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public double realPart() { return re; }
    public double imaginaryPart() { return im; }
    public double r() { return Math.sqrt(re * re + im * im); }
    public double theta() { return Math.atan(im / re); }

    public Complex add(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }
    public Complex subtract(Complex c) { ... }
    public Complex multiply(Complex c) { ... }
    public Complex divide(Complex c) { ... }
}

```



Class usage example

```
public class ComplexUser {  
    public static void main(String args[]) {  
        Complex c = new Complex(-1, 0);  
        Complex d = new Complex(0, 1);  
  
        Complex e = c.plus(d);  
        System.out.println(e.realPart() + " + "  
                           + e.imaginaryPart() + "i");  
  
        e = c.times(d);  
        System.out.println(e.realPart() + " + "  
                           + e.imaginaryPart() + "i");  
    }  
}
```

- When you run this program, what does it print?

-1.0 + 1.0i

-0.0 + -1.0i

Static vs. instance variables/methods of a class

- ***Class variable***: a variable associated with the class rather than with an instance of the class. You can also associate methods with a class--***class methods***.
 - To refer to class variables and methods, you join the class's name and the name of the class method or class variable together with a period ('.').
- **Methods and variables that are not class methods or class variables are known as *instance methods* and *instance variables*.**
 - To refer to instance methods and variables, you must reference the methods and variables from an instance of the class.
- **Summary:**
 - Class variables and class methods are associated with a class and occur once per class. Using them doesn't require object creation.
 - Instance methods and variables occur once per instance of a class.

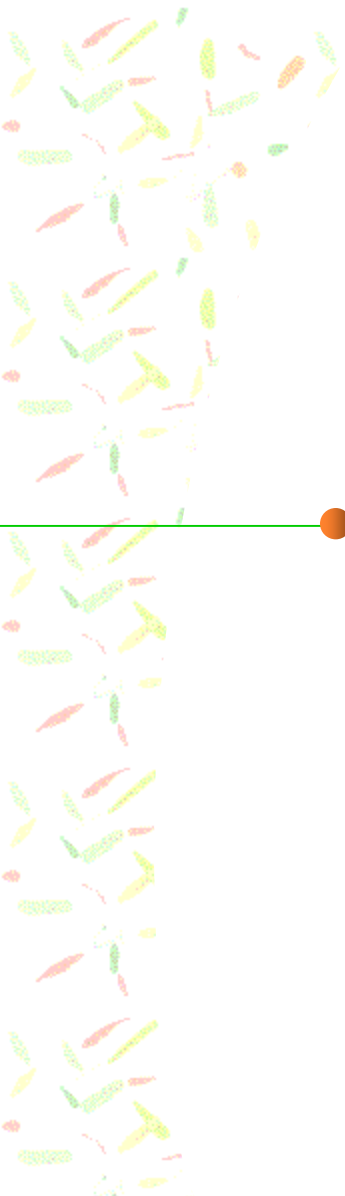
Static vs. instance variables/methods of a class

```
class DateApp {  
    public static void main(String args[]) {  
        Date today = new Date();  
        System.out.println(today);  
    }  
}
```

```
class Another {  
    public static void main(String[] args) {  
        int result;  
        result = Math.min(10, 20);  
        System.out.println(result);  
        System.out.println(Math.max(100, 200));  
    }  
}
```



3 Interface



Interface

- **Java's interface is a useful language mechanism for designing and expressing an ADT, with its implementation as a class implementing that interface.**
 - An interface in Java is a list of method signatures, but no method bodies.
 - A class implements an interface if it declares the interface in its implements clause, and provides method bodies for all of the interface's methods.
 - An interface can extend one or more others
 - A class can implement multiple interfaces

An interface to go with the example class

```
public interface Complex {  
    // No constructors, fields, or implementations!  
  
    double realPart();  
    double imaginaryPart();  
    double r();  
    double theta();  
  
    Complex plus(Complex c);  
    Complex minus(Complex c);  
    Complex times(Complex c);  
    Complex dividedBy(Complex c);  
}
```

Modifying class to use interface

```
class OrdinaryComplex implements Complex {
    double re; // Real Part
    double im; // Imaginary Part

    public OrdinaryComplex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public double realPart() { return re; }
    public double imaginaryPart() { return im; }
    public double r() { return Math.sqrt(re * re + im * im); }
    public double theta() { return Math.atan(im / re); }

    public Complex add(Complex c) {
        return new OrdinaryComplex(re + c.realPart(), im + c.imaginaryPart());
    }
    public Complex subtract(Complex c) { ... }
    public Complex multiply(Complex c) { ... }
    public Complex divide(Complex c) { ... }
}
```

Modifying client to use interface

```
public class ComplexUser {  
    public static void main(String args[]) {  
        Complex c = new OrdinaryComplex(-1, 0);  
        Complex d = new OrdinaryComplex(0, 1);  
  
        Complex e = c.plus(d);  
        System.out.println(e.realPart() + " + "  
                           + e.imaginaryPart() + "i");  
        e = c.times(d);  
        System.out.println(e.realPart() + " + "  
                           + e.imaginaryPart() + "i");  
    }  
}
```

Interface permits multiple implementations

```
class PolarComplex implements Complex {
    double r;
    double theta;

    public PolarComplex(double r, double theta) {
        this.r = r;
        this.theta = theta;
    }

    public double realPart()      { return r * Math.cos(theta) ; }
    public double imaginaryPart() { return r * Math.sin(theta) ; }
    public double r()             { return r; }
    public double theta()         { return theta; }

    public Complex plus(Complex c)      { ... } // Completely different impls
    public Complex minus(Complex c)     { ... }
    public Complex times(Complex c)     { ... }
    public Complex dividedBy(Complex c) { ... }
}
```

Interface decouples client from implementation

```
public class ComplexUser {  
    public static void main(String args[]) {  
        Complex c = new PolarComplex(Math.PI, 1); // -1  
        Complex d = new PolarComplex(Math.PI/2, 1); // i  
  
        Complex e = c.plus(d);  
        System.out.println(e.realPart() + " + "  
                           + e.imaginaryPart() + "i");  
  
        e = c.times(d);  
        System.out.println(e.realPart() + " + "  
                           + e.imaginaryPart() + "i");  
    }  
}
```

Java interfaces and classes

■ Interfaces vs. classes

- Interface: specifies expectations
- Class: delivers on expectations (the implementation)

■ Classes *do* define types

- Public class methods usable like interface methods
- Public fields directly accessible from other classes

■ But prefer the use of interfaces

- Use interface types for variables and parameters unless you know one implementation will suffice.
- Supports change of implementation;
- Prevents dependence on implementation details

```
Set<Criminal> senate = new HashSet<>();  
HashSet<Criminal> senate = new HashSet<>();
```

```
// Do this...  
// Not this
```

Another interface example: MyString

```
/** MyString represents an immutable sequence of characters. */
public interface MyString {

    // We'll skip this creator operation for now
    // /** @param b a boolean value
    //  * @return string representation of b, either "true" or "false" */
    // public static MyString valueOf(boolean b) { ... }

    /** @return number of characters in this string */
    public int length();

    /** @param i character position (requires 0 <= i < string length)
     *  * @return character at position i */
    public char charAt(int i);

    /** Get the substring between start (inclusive) and end (exclusive).
     *  * @param start starting index
     *  * @param end ending index. Requires 0 <= start <= end <= string length.
     *  * @return string consisting of charAt(start)...charAt(end-1) */
    public MyString substring(int start, int end);
}
```

Implementation 1 of MyString

```
public class SimpleMyString implements MyString {

    private char[] a;

    /* Create an uninitialized SimpleMyString. */
    private SimpleMyString() {}

    /** Create a string representation of b, either "true" or "false".
     * @param b a boolean value */
    public SimpleMyString(boolean b) {
        a = b ? new char[] { 't', 'r', 'u', 'e' }
            : new char[] { 'f', 'a', 'l', 's', 'e' };
    }

    @Override public int length() { return a.length; }

    @Override public char charAt(int i) { return a[i]; }

    @Override public MyString substring(int start, int end) {
        SimpleMyString that = new SimpleMyString();
        that.a = new char[end - start];
        System.arraycopy(this.a, start, that.a, 0, end - start);
        return that;
    }
}
```


Implementation 2 of MyString

```
public class FastMyString implements MyString {

    private char[] a;
    private int start;
    private int end;

    /* Create an uninitialized FastMyString. */
    private FastMyString() {}

    /** Create a string representation of b, either "true" or "false".
     * @param b a boolean value */
    public FastMyString(boolean b) {
        a = b ? new char[] { 't', 'r', 'u', 'e' }
              : new char[] { 'f', 'a', 'l', 's', 'e' };
        start = 0;
        end = a.length;
    }

    @Override public int length() { return end - start; }

    @Override public char charAt(int i) { return a[start + i]; }

    @Override public MyString substring(int start, int end) {
        FastMyString that = new FastMyString();
        that.a = this.a;
        that.start = this.start + start;
        that.end = this.start + end;
        return that;
    }
}
```

To use MyString and its implementations

```
MyString s = new FastMyString(true);  
System.out.println("The first character is: " + s.charAt(0));
```

- **Problem: breaks the abstraction barrier**
 - Clients must know the name of the concrete representation class.
 - Because interfaces in Java cannot contain constructors, they must directly call one of the concrete class' constructors.
 - The spec of that constructor won't appear anywhere in the interface, so there's no static guarantee that different implementations will even provide the same constructors.

Using static factory instead of constructor

```
public interface MyString {  
  
    /** @param b a boolean value  
     * @return string representation of b, either "true" or "false" */  
    public static MyString valueOf(boolean b) {  
        return new FastMyString(true);  
    }  
  
    // ...  
}
```

```
MyString s = MyString.valueOf(true);  
System.out.println("The first character is: " + s.charAt(0));
```

Advantages of interface

- **Interface specifies the contract for the client and nothing more.**
 - The interface is all a client programmer needs to read to understand the ADT.
 - The client can't create inadvertent dependencies on the ADT's rep, because instance variables can't be put in an interface at all.
 - The implementation is kept well and truly separated, in a different class altogether.
- **Multiple different representations of the abstract data type can co-exist in the same program, as different classes implementing the interface.**
 - When an abstract data type is represented just as a single class, without an interface, it's harder to have multiple representations.

Why multiple implementations?

- **Different performance**

- Choose implementation that works best for your use

- **Different behavior**

- Choose implementation that does what you want
- Behavior *must* comply with interface spec (“contract”)

- **Often performance and behavior *both* vary**

- Provides a functionality – performance tradeoff
- Example: HashSet, TreeSet



4 Encapsulation and information hiding



Information hiding

- Single most important factor that distinguishes a well-designed module from a bad one is the degree to which it hides internal data and other implementation details from other modules
- Well-designed code hides *all* implementation details
 - Cleanly separates API from implementation
 - Modules communicate *only* through APIs
 - They are oblivious to each others' inner workings
- Known as *information hiding* or *encapsulation*, a fundamental tenet of software design.

Benefits of information hiding

- **Decouples** the classes that comprise a system
 - Allows them to be developed, tested, optimized, used, understood, and modified in isolation
- **Speeds up system development**
 - Classes can be developed in parallel
- **Eases burden of maintenance**
 - Classes can be understood more quickly and debugged with little fear of harming other modules
- **Enables effective performance tuning**
 - “Hot” classes can be optimized in isolation
- **Increases software reuse**
 - Loosely-coupled classes often prove useful in other contexts

Information hiding with interfaces

- **Declare variables using interface type**
- **Client can use only interface methods**
- **Fields not accessible from client code**
- **But this only takes us so far**
 - Client can access non-interface members directly
 - In essence, it's voluntary information hiding

Best practices for information hiding

- Carefully design your API
- Provide *only* functionality required by clients, and all other members should be private
- You can always make a private member public later without breaking clients
 - But not vice-versa!



5 Inheritance and Overriding



Inheritance and subtyping

- **Inheritance** is for code reuse

Class A extends B

- Write code once and only once
- Superclass features implicitly available in subclass

- **Subtyping** is for polymorphism

Class A implements I

- Accessing objects the same way, but getting different behavior
- Subtype is substitutable for supertype

- **An interface defines expectations / commitments for clients**

- **A class fulfills the expectations of an interface**

- An abstract class is a convenient hybrid
- A subclass specializes a class's implementation



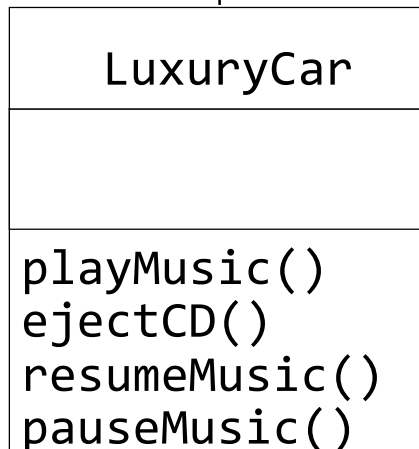
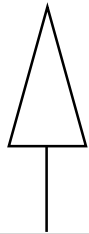
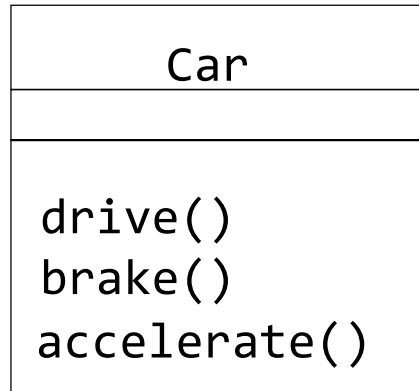
(1) Overriding



Rewriteable Methods and Strict Inheritance

- **Rewriteable Method: A method which allow a re-implementation.**
 - In Java methods are rewriteable by default, i.e. there is no special keyword.
- **Strict inheritance**
 - The subclass can only add new methods to the superclass, it cannot overwrite them
 - If a method cannot be overwritten in a Java program, it must be prefixed with the keyword **final**.

Strict Inheritance



■ Superclass

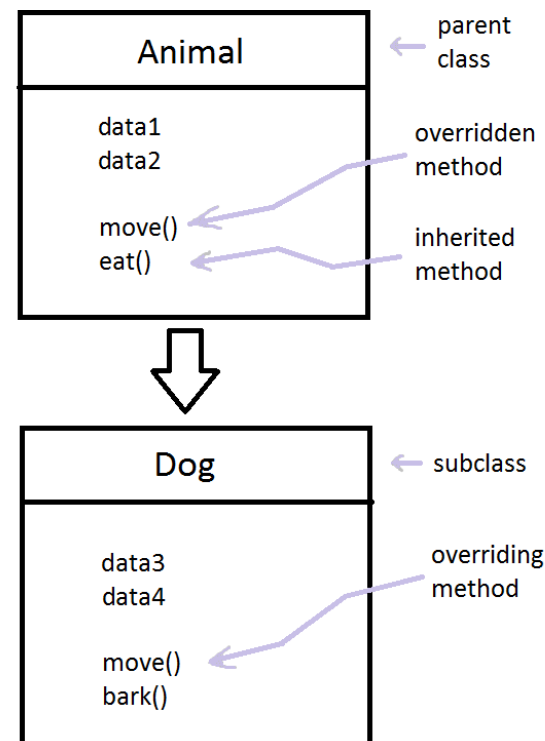
```
public class Car {
    public final void drive() {...}
    public final void brake() {...}
    public final void accelerate() {...}
}
```

■ Subclass

```
public class LuxuryCar extends Car {
    public void playMusic() {...}
    public void ejectCD() {...}
    public void resumeMusic() {...}
    public void pauseMusic() {...}
}
```

Overriding (覆盖/重写)

- **Method overriding is a language feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes.**
 - The implementation in the subclass overrides (replaces) the implementation in the superclass by providing a method that has same name, same parameters or signature, and same return type as the method in the parent class.
 - The version of a method that is executed will be determined by the object that is used to invoke it.
 - If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.



Overriding (覆盖/重写)

- When a subclass contains a method that overrides a method of the superclass, it can also invoke the superclass method by using the keyword **super**.

```
class Thought {
    public void message() {
        System.out.println("I feel like I am diagonally parked in a parallel universe.");
    }
}

public class Advice extends Thought {
    @Override // @Override annotation in Java 5 is optional but helpful.
    public void message() {
        System.out.println("Warning: Dates in calendar are closer than they appear.");
    }
}
```

```
Thought parking = new Thought();
parking.message(); // Prints "I feel like I am diagonally parked in a parallel universe."
```

```
Thought dates = new Advice(); // Polymorphism
dates.message(); // Prints "Warning: Dates in calendar are closer than they appear."
```

```
public class Advice extends Thought {
    @Override
    public void message() {
        System.out.println("Warning: Dates in calendar are closer than they appear.");
        super.message(); // Invoke parent's version of method.
    }
}
```

Constructors with `this` and `super`

```
public class CheckingAccountImpl
    extends AbstractAccount implements CheckingAccount {
```

```
    private long fee;
```

```
    public CheckingAccountImpl(long initialBalance, long fee) {
        super(initialBalance);
        this.fee = fee;
    }
```

Invokes a constructor of the superclass. Must be the first statement of the constructor.

```
    public CheckingAccountImpl(long initialBalance) {
        this(initialBalance, 500);
    }
    /* other methods... */ }
```

Invokes another constructor in this same class

Bad Use of Overwriting Methods

- One can overwrite the operations of a superclass with completely new meanings.

- Example:

```
Public class SuperClass {  
    public int add (int a, int b) { return a+b; }  
    public int subtract (int a, int b) { return a-b; }  
}  
  
Public class SubClass extends SuperClass {  
    public int add (int a, int b) { return a-b; }  
    public int subtract (int a, int b) { return a+b; }  
}
```

- We have redefined addition as subtraction and subtraction as addition!!

final

- A **final** field: prevents reassignment to the field after initialization
- A **final method**: prevents overriding the method
- A **final** class: prevents extending the class
 - e.g., `public final class CheckingAccountImpl { ...`



(2) Abstract Class



Abstract Methods and Abstract Classes

- **Abstract method:**

- A method with a signature but without an implementation (also called abstract operation)
- Defined by the keyword **abstract**

- **Abstract class:**

- A class which contains at least one **abstract method** is called abstract class

- **Interface: An abstract class which has only abstract methods**

- An interface is primarily used for the specification of a system or subsystem. The implementation is provided by a subclass or by other mechanisms.

- **Concrete class → Abstract Class → Interface**



6 Polymorphism, subtyping and overloading





(1) Three Types of Polymorphism



Three Types of Polymorphism (多态)

- A **polymorphic type** is one whose operations can be applied to values of some other types.
 - **Ad hoc polymorphism**: when a function denotes different and potentially heterogeneous implementations depending on a limited range of individually specified types and combinations. Ad hoc polymorphism is supported in many languages using **function overloading**.
 - **Parametric polymorphism**: when code is written without mention of any specific type and thus can be used transparently with any number of new types. In the object-oriented programming community, this is often known as generics or **generic programming**.
 - **Subtyping** (also called **subtype polymorphism** or **inclusion polymorphism**): when a name denotes instances of many different classes related by some common superclass. (We have already discussed it in previous section)



(2) Ad hoc polymorphism and Overloading



Ad hoc polymorphism

- **Ad-hoc polymorphism is obtained when a function works on several different types (which may not exhibit a common structure) and may behave in unrelated ways for each type.**

```
public class OverloadExample {  
    public static void main(String args[]) {  
        System.out.println(add("C","D"));  
        System.out.println(add("C","D","E"));  
        System.out.println(add(2,3));  
    }  
    public static String add(String c, String d) {  
        return c.concat(d);  
    }  
    public static String add(String c, String d, String e){  
        return c.concat(d).concat(e);  
    }  
    public static int add(int a, int b) {  
        return a+b;  
    }  
}
```



(3) Overloading



Overloading (重载)

- **doTask() and doTask(object 0) are overloaded methods.**
 - To call the latter, an object must be passed as a parameter, whereas the former does not require a parameter, and is called with an empty parameter field.
 - A common error would be to assign a default value to the object in the second method, which would result in an ambiguous call error, as the compiler wouldn't know which of the two methods to use.
- **A method Print(object 0): one might like the method to be different when printing, for example, text or pictures.**
 - Two different methods may be overloaded as Print(text_object T); Print(image_object P).
 - If we write the overloaded print methods for all objects our program will "print", we never have to worry about the type of the object, and the correct function call again, the call is always: Print(something).

Overloading (重载)

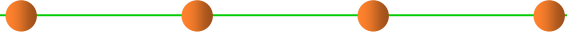
- **Function overloading** is the ability to create multiple methods of the same name with different implementations.
 - Calls to an overloaded function will run a specific implementation of that function appropriate to the context of the call, allowing one function call to perform different tasks depending on context.

- **Overloading is a static polymorphism**
 - A function call is resolved using the 'best match technique', i.e. the function is resolved depending upon the argument list.
 - Static type checking in function calls
 - The determination of which of these methods are used is resolved at compile time.

Overloading rules

- **Rules in function overloading: the overloaded function must **differ either by the arity or data types****
 - Overloaded methods **MUST** change the argument list.
 - Overloaded methods **CAN** change the return type.
 - Overloaded methods **CAN** change the access modifier.
 - Overloaded methods **CAN** declare new or broader checked exceptions.
 - A method can be overloaded in the same class or in a subclass.

Legal Overloads



```
public void changeSize(int size,  
                        String name, float pattern) { }
```

- **The following methods are legal overloads of the changeSize() method:**
 - `public void changeSize(int size, String name) { }`
 - `public int changeSize(int size, float pattern) { }`
 - `public void changeSize(float pattern, String name){ }`

Invoking overloaded methods

```
class Animal {  
    public void eat() {}  
}  
  
class Horse extends Animal {  
    public void eat(String food) {}  
}  
  
public class UseAnimals {  
    public void doStuff(Animal a) {  
        System.out.println("Animal");  
    }  
    public void doStuff(Horse h) {  
        System.out.println("Horse");  
    }  
}
```

```
public class TestUseAnimals {  
  
    public static void main (String [] args) {  
        UseAnimals ua = new UseAnimals();  
  
        Animal animalobj = new Animal();  
        Horse horseobj = new Horse();  
        Animal animalRefToHorse = new Horse();  
  
        ua.doStuff(animalobj);  
        ua.doStuff(horseobj);  
        ua.doStuff(animalRefToHorse);  
    }  
}
```

Which overridden version of the method to call is decided at runtime based on object type, but which overloaded version of the method to call is based on the reference type of the argument passed at compile time.

Invoking overloaded methods

```
class Animal {
    public void eat() {}
}

class Horse extends Animal {
    public void eat(String food) {}
}
```

Method Invocation Code	Result
<code>Animal a = new Animal(); a.eat();</code>	Generic Animal Eating Generically
<code>Horse h = new Horse(); h.eat();</code>	Horse eating hay
<code>Animal ah = new Horse(); ah.eat();</code>	Horse eating hay Polymorphism works—the actual object type (Horse), not the reference type (Animal), is used to determine which eat() is called.
<code>Horse he = new Horse(); he.eat("Apples");</code>	Horse eating Apples The overloaded eat(String s) method is invoked.
<code>Animal a2 = new Animal(); a2.eat("treats");</code>	Compiler error! Compiler sees that Animal class doesn't have an eat() method that takes a String.
<code>Animal ah2 = new Horse(); ah2.eat("Carrots");</code>	Compiler error! Compiler <i>still</i> looks only at the reference, and sees that Animal doesn't have an eat() method that takes a String. Compiler doesn't care that the actual object might be a Horse at runtime.

Check your understanding

```
interface Animal {  
    void vocalize();  
}  
class Dog implements Animal {  
    public void vocalize() { System.out.println("Woof!"); }  
}  
class Cow implements Animal {  
    public void vocalize() { moo(); }  
    public void moo() {System.out.println("Moo!"); }  
}
```

■ What will happen?

1. `Animal a = new Animal();`
 `a.vocalize();`
2. `Dog d = new Dog();`
 `d.vocalize();`
3. `Animal b = new Cow();`
 `b.vocalize();`
4. `b.moo();`

Overriding vs. Overloading

	Overloading	Overriding
Argument list	Must change	Must not change
Return type	Can Change	Must not change
Exceptions	Can Change	Can reduce or eliminate Must not throw new or broader checked exception
Access	Can Change	Must not make more restrictive (can be less restrictive)
Invocation	Reference type determines which overloaded version (based on declared argument types) is selected. Happens at compile time. The actual method that's invoked is still a virtual method invocation that happens at runtime, but the compiler will always know the signature of the method that is to be invoked. So at runtime, the argument match will have already been nailed down, just not the actual class in which the method lives	Object type (in other words, the type of the actual instance on the heap) determines which method is selected Happens at runtime.



(3) Parametric polymorphism and Generic programming



Parametric polymorphism

- **Parametric polymorphism is obtained when a function works uniformly on a range of types; these types normally exhibit some common structure.**
 - It is the ability to define functions and types in a generic way so that it works based on the parameter passed at runtime, i.e., allowing static type-checking without fully specifying the type.
 - This is what is called “Generics (泛型)” in Java.
- **Generic programming** is a style of programming in which data types and functions are written in terms of **types to-be-specified-later** that are then instantiated when needed for specific types provided as parameters.

Generic programming centers around the idea of *abstracting from concrete*, efficient algorithms to obtain generic algorithms that can be combined with different data representations *to produce a wide variety of useful software.*

Template in C++

```
template<typename T>
class List {
    /* class contents */
};

List<Animal> list_of_animals;
List<Car> list_of_cars;
```

C++ Standard Library includes the Standard Template Library (STL) that provides a framework of templates for common data structures and algorithms.

```
template<typename T>
void Swap(T & a, T & b) {
    T temp = b;
    b = a;
    a = temp;
}

string hello = "world!"
string world = "Hello,";
Swap( world, hello );
cout << hello << world << endl;
```

Type variables

- Using <>, the diamond operator, to help declare type variables.
- For example:
 - `List<Integer> ints = new ArrayList<Integer>();`
 - `public interface List<E>`
 - `public class Entry<KeyType, ValueType>`


```
public class PapersJar<T> {  
  
    private List<T> itemList = new ArrayList<>();  
  
    public void add(T item) {  
        itemList.add(item);  
    }  
  
    public T get(int index) {  
        return (T) itemList.get(index);  
    }  
  
    public static void main(String args[]) {  
        PapersJar<String> papersStr = new PapersJar<>();  
        papersStr.add("Lion");  
        String str = (String) papersStr.get(0);  
        System.out.println(str);  
  
        PapersJar papersInt = new PapersJar();  
        papersInt.add(new Integer(100));  
        Integer integerObj = (Integer) papersInt.get(0);  
        System.out.println(integerObj);  
    }  
}
```

Example

```
public class Pair<E> {  
    private final E first, second;  
    public Pair(E first, E second) {  
        this.first = first;  
        this.second = second;  
    }  
    public E first() { return first; }  
    public E second() { return second; }  
}
```

Client:

```
Pair<String> p = new Pair<>("Hello", "world");  
String result = p.first();
```

Another example: Java Set

- **Set** is the ADT of finite sets of elements of some other type **E** .

```
/** A mutable set.  
 * @param <E> type of elements in the set */  
public interface Set<E> {
```

- **Set** is an example of a generic type : a type whose specification is in terms of a placeholder type to be filled in later.
- Instead of writing separate specifications and implementations for **Set<String>** , **Set<Integer>** , and so on, we design and implement one **Set<E>** .

Another example: Java Set

■ Creator

```
// example creator operation
/** Make an empty set.
 * @param <E> type of elements in the set
 * @return a new set instance, initially empty */
public static <E> Set<E> make() { ... }
```

■ Observer

```
// example observer operations

/** Get size of the set.
 * @return the number of elements in this set */
public int size();

/** Test for membership.
 * @param e an element
 * @return true iff this set contains e */
public boolean contains(E e);
```

■ Mutator

```
// example mutator operations

/** Modifies this set by adding e to the set.
 * @param e element to add */
public void add(E e);

/** Modifies this set by removing e, if found.
 * If e is not found in the set, has no effect.
 * @param e element to remove */
public void remove(E e);
```

Generic Interfaces

- **Suppose we want to implement the generic `Set<E>` interface.**
 - Way 1: Generic interface, non-generic implementation: to implement `Set<E>` for a particular type `E`.

```
public interface Set<E> {

    // ...

    /**
     * Test for membership.
     * @param e an element
     * @return true iff this set contains e
     */
    public boolean contains(E e);

    /**
     * Modifies this set by adding e to the set.
     * @param e element to add
     */
    public void add(E e);

    // ...

}
```

```
public class CharSet1 implements Set<Character> {

    private String s = "";

    // ...

    @Override
    public boolean contains(Character e) {
        checkRep();
        return s.indexOf(e) != -1;
    }

    @Override
    public void add(Character e) {
        if (!contains(e)) s += e;
        checkRep();
    }

    // ...

}
```

Generic Interfaces

■ Way 2: Generic interface, generic implementation.

- We can also implement the generic `Set<E>` interface without picking a type for `E`.
- In that case, we write our code blind to the actual type that clients will choose for `E`.
- Java's `HashSet` does that for `Set`.

```
public interface Set<E> {  
  
    // ...
```

```
public class HashSet<E> implements Set<E> {  
  
    // ...
```

Some Java Generics details

- **Can have multiple type parameters**
 - e.g., `Map<E, F>`, `Map<String, Integer>`
- **Wildcards**
 - e.g. `List<?>` or `List<? extends Animal>` or `List<? super Animal>`
- **Generics are type invariant**
 - `ArrayList<String>` is a subtype of `List<String>`
 - `List<String>` is not a subtype of `List<Object>`
 - `List<String>` is a subtype of `List<? extends Object>`
 - `List<Object>` is a subtype of `List<? super String>`
- **Generic type info is erased (i.e. compile-time only)**
 - Cannot use `instanceof()` to check generic type
- **Cannot create Generic arrays**
 - `Pair<String>[] foo = new Pair<String>[42];` // won't compile



(4) Subtyping Polymorphism



Inheritance and subtyping

- **Inheritance** is for code reuse

Class A extends B

- Write code once and only once
- Superclass features implicitly available in subclass

- **Subtyping** is for polymorphism

Class A implements I

- Accessing objects the same way, but getting different behavior
- Subtype is substitutable for supertype

- **An interface defines expectations / commitments for clients**

- **A class fulfills the expectations of an interface**

- An abstract class is a convenient hybrid
- A subclass specializes a class's implementation

Subtypes

- **A type is a set of values.**
 - The Java `List` type is defined by an interface. If we think about all possible `List` values, none of them are `List` objects: we cannot create instances of an interface.
 - Instead, those values are all `ArrayList` objects, or `LinkedList` objects, or objects of another class that implements `List` .
- **A subtype is simply a subset of the supertype**
 - `ArrayList` and `LinkedList` are subtypes of `List` .

Subtypes

- **“B is a subtype of A” means “every B is an A.”**
- **In terms of specifications: “every B satisfies the specification for A.”**
 - B is only a subtype of A if B’s specification is at least as strong as A’s specification.
 - When we declare a class that implements an interface, the Java compiler enforces part of this requirement automatically: it ensures that every method in A appears in B, with a compatible type signature.
 - Class B cannot implement interface A without implementing all of the methods declared in A.

Static checking on subtypes

- **But the compiler cannot check that we haven't weakened the specification in other ways:**
 - Strengthening the precondition on some inputs to a method
 - Weakening a postcondition
 - Weakening a guarantee that the interface abstract type advertises to clients.
- **If you declare a subtype in Java (e.g., implementing an interface), then you must ensure that the subtype's spec is at least as strong as the supertype's.**

Interface inheritance for an account type hierarchy

```
public interface Account {
    public long getBalance();
    public void deposit(long amount);
    public boolean withdraw(long amount);
    public boolean transfer(long amount, Account target);
    public void monthlyAdjustment();
}

public interface CheckingAccount extends Account {
    public long getFee();
}

public interface SavingsAccount extends Account {
    public double getInterestRate();
}

public interface InterestCheckingAccount
    extends CheckingAccount, SavingsAccount {
}
```

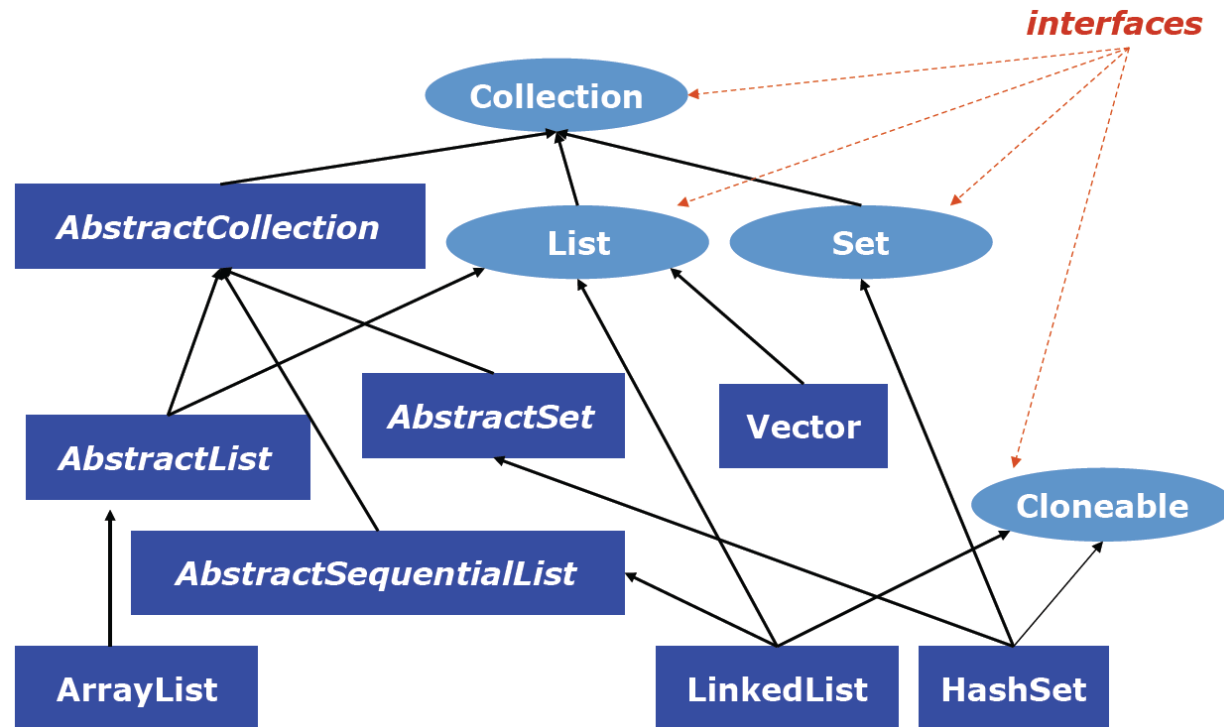
Subtype polymorphism

- Different kinds of objects can be treated uniformly by client code
- Each object behaves according to its type (e.g., if you add new kind of account, client code does not change)

```
If today is the last day of the month:  
    For each acct in allAccounts:  
        acct.monthlyAdjustment();
```

Inheritance and Subtype: a glimpse at the hierarchy

■ Java Collections API



- **Benefits of inheritance/subtype:** Reuse of code, Modeling flexibility
- **In Java:** Each class can directly extend only one parent class; A class can implement multiple interfaces.

Type casting

- **Sometimes you want a different type than you have**

```
double pi = 3.14;
```

```
int indianaPi = (int) pi;
```

- **Useful if you know you have a more specific subtype:**

```
Account acct = ...;
```

```
CheckingAccount checkingAcct = (CheckingAccount) acct;
```

```
long fee = checkingAcct.getFee();
```

- **But it will get a `ClassCastException` if types are incompatible**

- **Advice:**

- Avoid downcasting types

instanceof

- Operator that tests whether an object is of a given class

```
public void doSomething(Account acct) {  
    long adj = 0;  
    if (acct instanceof CheckingAccount) {  
        checkingAcct = (CheckingAccount) acct;  
        adj = checkingAcct.getFee();  
    } else if (acct instanceof SavingsAccount) {  
        savingsAcct = (SavingsAccount) acct;  
        adj = savingsAcct.getInterest();  
    }  
    ...  
}
```

- Advice: avoid instanceof() if possible.



7 Dynamic dispatch



Dynamic dispatch (binding)

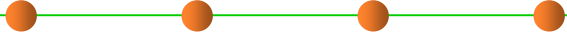
- **Dynamic dispatch is the process of selecting which implementation of a polymorphic operation (method or function) to call at run time.**
 - Object-oriented systems model a problem as a set of interacting objects that enact operations referred to by name.
 - Polymorphism is the phenomenon wherein somewhat interchangeable objects each expose an operation of the same name but possibly differing in behavior.

Determining which method to call at runtime, i.e., a call to an overridden or polymorphic method is resolved at runtime

Dynamic dispatch

- As an example, a File object and a Database object both have a StoreRecord method that can be used to write a personnel record to storage. Their implementations differ.
- A program holds a reference to an object which may be either a File object or a Database object. Which it is may have been determined by a run-time setting, and at this stage, the program may not know or care which.
- When the program calls StoreRecord on the object, something needs to decide which behavior gets enacted.
- The program sends a StoreRecord message to an object of unknown type, leaving it to the run-time support system to dispatch the message to the right object. The object enacts whichever behavior it implements.

Dynamic dispatch



```
dividend.divide(divisor)  # dividend / divisor
```

- This is thought of as sending a message named `divide` with parameter `divisor` to `dividend`.
- An implementation will be chosen based only on `dividend`'s type (perhaps rational, floating point, matrix), disregarding the type or value of `divisor`.

Dynamic method dispatch

1. **(Compile time) Determine which class to look in**
2. **(Compile time) Determine method signature to be executed**
 - Find all accessible, applicable methods
 - Select most specific matching method
3. **(Run time) Determine dynamic class of the receiver**
4. **(Run time) From dynamic class, locate method to invoke**
 - Look for method with the same signature found in step 2
 - Otherwise search in superclass and etc.



8 Delegation and composition



A Sorting example

- **Version A:**

```
static void sort(int[] list, boolean ascending) {  
    ...  
    boolean mustSwap;  
    if (ascending) {  
        mustSwap = list[i] < list[j];  
    } else {  
        mustSwap = list[i] > list[j];  
    }  
    ...  
}
```

- **Version B:**

```
interface Comparator {  
    boolean compare(int i, int j);  
}  
final Comparator ASCENDING = (i, j) -> i < j;  
final Comparator DESCENDING = (i, j) -> i > j;  
  
static void sort(int[] list, Comparator cmp) {  
    ...  
    boolean mustSwap =  
        cmp.compare(list[i], list[j]);  
    ...  
}
```


Delegation

- **Delegation** is simply when one object relies on another object for some subset of its functionality (one entity passing something to another entity)
 - e.g. here, the Sorter is delegating functionality to some Comparator
- **Judicious delegation enables code reuse**
 - Sorter can be reused with arbitrary sort orders
 - Comparators can be reused with arbitrary client code that needs to compare integers
- **Explicit delegation:** passing the sending object to the receiving object
- **Implicit delegation:** by the member lookup rules of the language
- **Delegation** can be described as a low level mechanism for sharing code and data between entities.

A simple Delegation example

```
class RealPrinter {    // the "receiver"
    void print() {
        System.out.println("Hello world!");
    }
}

class Printer {    // the "sender"
    RealPrinter p = new RealPrinter(); // create the receiver
    void print() {
        p.print(); // calls the receiver
    }
}

public class Main {
    public static void main(String[] arguments) {
        // to the outside world it looks like Printer actually prints.
        Printer printer = new Printer();
        printer.print();
    }
}
```

A simple Delegation example

```

interface I {
    void f();
    void g();
}

class A implements I {
    public void f() { System.out.println("A: doing f()"); }
    public void g() { System.out.println("A: doing g()"); }
}

class B implements I {
    public void f() { System.out.println("B: doing f()"); }
    public void g() { System.out.println("B: doing g()"); }
}

// changing the implementing object in run-time (normally done in compile time)
class C implements I {
    I i = null;
    // forwarding
    public C(I i) { setI(i); }
    public void f() { i.f(); }
    public void g() { i.g(); }

    // normal attributes
    public void setI(I i) { this.i = i; }
}

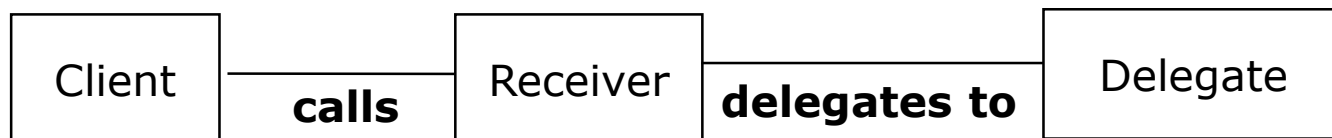
public class Main {
    public static void main(String[] arguments) {
        C c = new C(new A());
        c.f(); // output: A: doing f()
        c.g(); // output: A: doing g()
        c.setI(new B());
        c.f(); // output: B: doing f()
        c.g(); // output: B: doing g()
    }
}

```

Switch to
another
forwarder

Delegation

- The delegation pattern is a software design pattern for implementing delegation, though this term is also used loosely for consultation or forwarding.
- Delegation is dependent upon dynamic binding, as it requires that a given method call can invoke different segments of code at runtime.
- **Process**
 - The Receiver object delegates operations to the Delegate object
 - The Receiver object makes sure, that the Client does not misuse the Delegate object.



Using delegation to extend functionality

- Consider `java.util.List`

```
public interface List<E> {  
    public boolean add(E e);  
    public E        remove(int index);  
    public void     clear();  
    ...  
}
```

- Suppose we want a list that logs its operations to the console...

- The `LoggingList` is composed of a `List`, and delegates (the non-logging) functionality to that `List`.

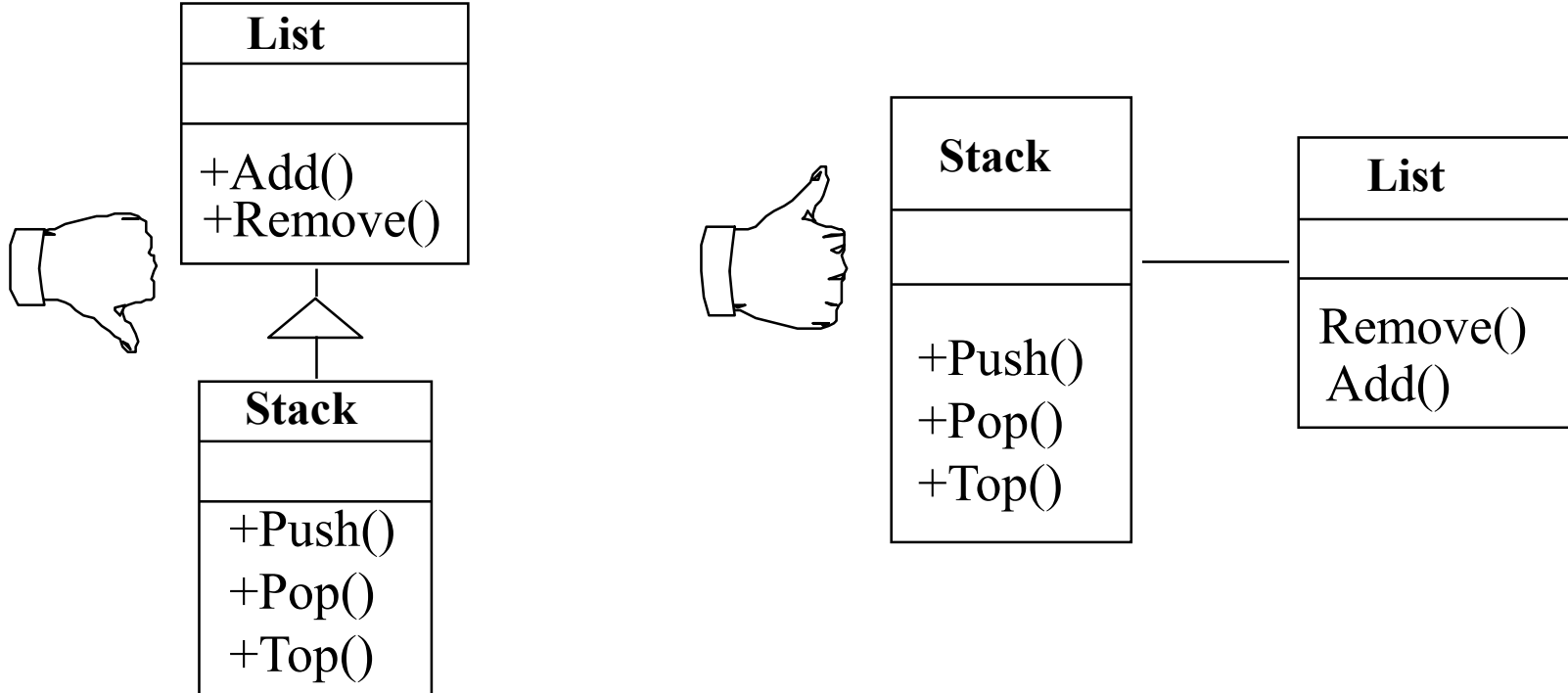
```
public class LoggingList<E> implements List<E> {  
    private final List<E> list;  
    public LoggingList<E>(List<E> list) { this.list = list; }  
    public boolean add(E e) {  
        System.out.println("Adding " + e);  
        return list.add(e);  
    }  
    public E remove(int index) {  
        System.out.println("Removing at " + index);  
        return list.remove(index);  
    }  
}
```

Forwarding

- **Forwarding:** using a member of an object (either a property or a method) results in actually using the corresponding member of a different object: the use is forwarded to another object.
- The forwarding object is frequently called a **wrapper** object, and explicit forwarding members are called wrapper functions.
- **The difference between forwarding and delegation is the binding of the self parameter in the wrappee when called through the wrapper.**
 - With delegation, the self parameter is bound to the wrapper, with forwarding it is bound to the wrappee.
 - Forwarding is a form of automatic message resending; delegation is a form of inheritance with binding of the parent (superclass) at run time, rather than at compile/link time as with 'normal' inheritance.

Delegation vs. Inheritance

- **Inheritance:** Extending a Base class by a new operation or overwriting an operation.
- **Delegation:** Catching an operation and sending it to another object.
- Many design patterns use a combination of inheritance and delegation.



Delegation vs. Inheritance

- Calling `b.foo()` will result in what?

```
class A {  
    void foo() {  
        this.bar();  
    }  
  
    void bar() {  
        print("A.bar");  
    }  
};  
  
class B extends A {  
    public B() {}  
  
    void foo() {  
        super.foo(); // call foo() of the superclass (A)  
    }  
  
    void bar() {  
        print("B.bar");  
    }  
};  
  
b = new B();
```


Delegation: association and dependency

- **Association: a persistent relationship between classes of objects that allows one object instance to cause another to perform an action on its behalf.**
 - This relationship is structural, because it specifies that objects of one kind are connected to objects of another and does not represent behavior.
- **Dependency: a temporary relationship that an object requires other objects (suppliers) for their implementation.**

Delegation: composition and aggregation

- **Composition is a way to combine simple objects or data types into more complex ones.**
 - An object of a composite type (e.g. car) "has an" object of a simpler type (e.g. wheel).
 - Composition is implemented such that an object contains another object.
 - A special composition: **aggregation**
- **In composition, when the owning object is destroyed, so are the contained objects. In aggregation, this is not necessarily true.**
 - A university owns various departments, and each department has a number of professors. If the university closes, the departments will no longer exist, but the professors in those departments will continue to exist.
 - A University can be seen as a composition of departments, whereas departments have an aggregation of professors. A Professor could work in more than one department, but a department could not be part of more than one university.

Five types of relations between objects



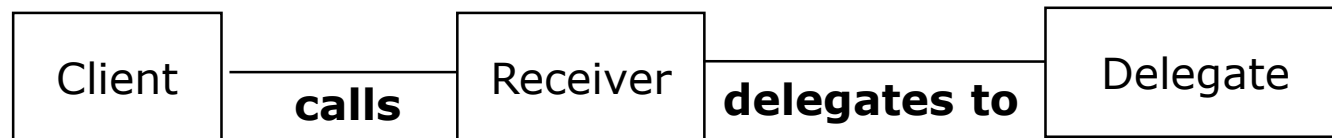
■ Delegation

- Association
- Dependency/use
- Composition
- aggregation

■ Inheritance

Types of delegation

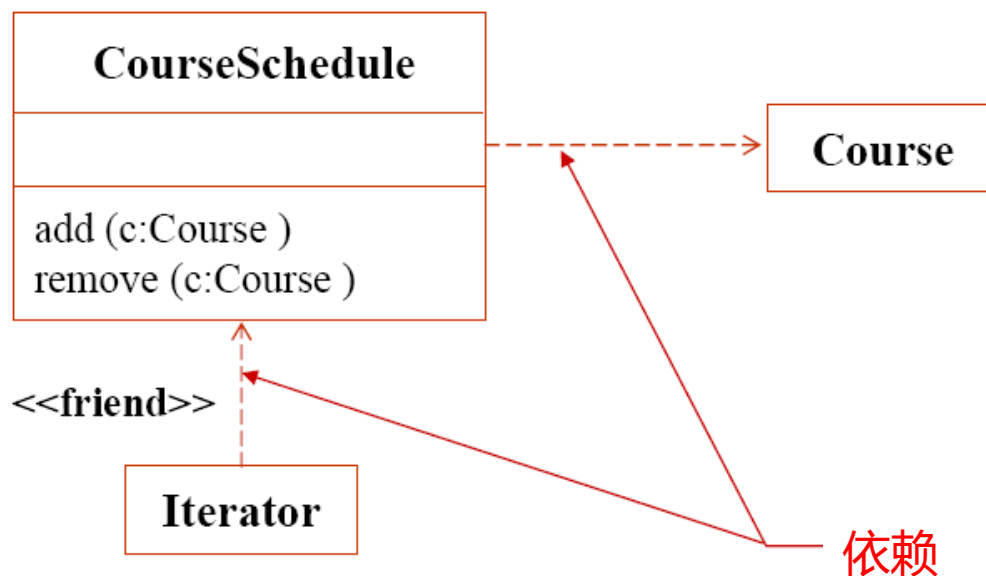
- Use (A use B)
- Composition/aggregation (A owns B)
- Association (A has B)



**Use
Composition
Aggregation
Association**

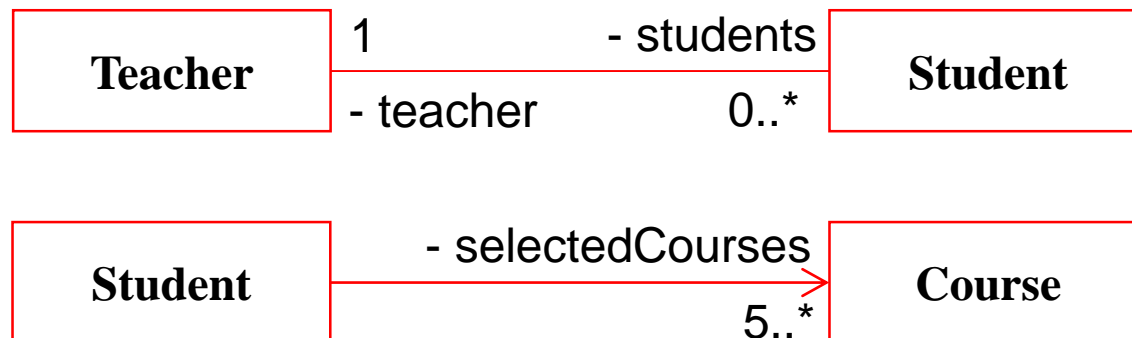
Approaches of reusing a class: use

- The simplest form of using classes is **calling its methods**;
- This form of relationship between two classes is called “**uses-a**” relationship
- Uses (in which one class makes use of another without actually incorporating it as a property. -it may, for example, be a parameter or used locally in a method)



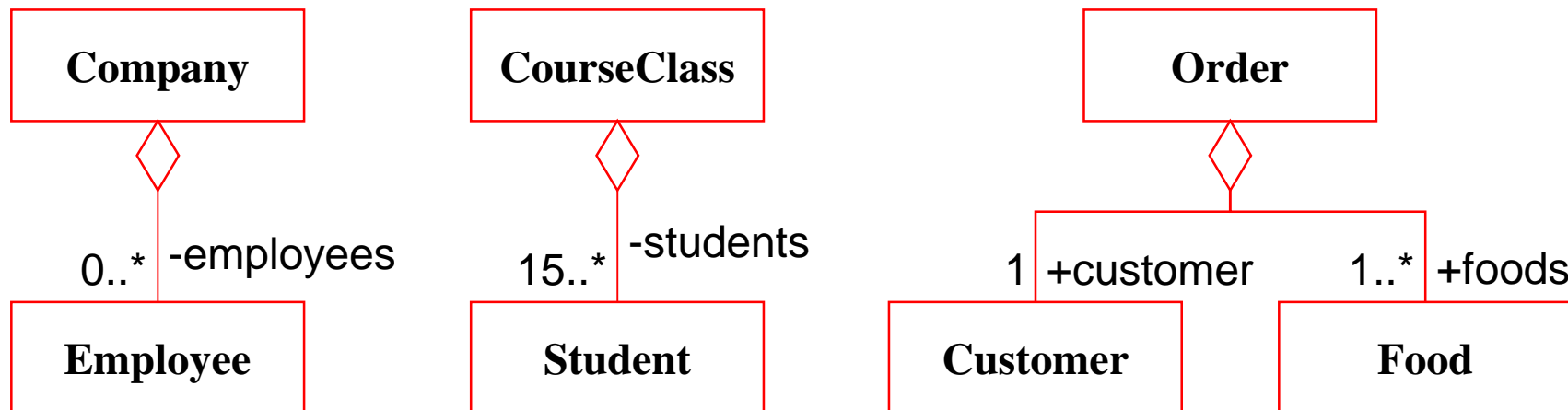
Approaches of reusing a class: association

- A closer form of reuse is association
- Association (or **has_a** in which one class has another as a property/instance variable)



Approaches of reusing a class: composition

- Another closer form of reuse is composition
- Composition (or **owns_a** in which one class has another as a property/instance variable)





9 Some important Object methods in Java



Overriding Object methods

- `equals()` – true if the two objects are “equal”
- `hashCode()` – a hash code for use in hash maps
- `toString()` – a printable string representation

- `toString()` – **ugly and uninformative**
 - You know what your object is so you can do better
 - Always override unless you know it won’t be called
- `equals` & `hashCode` – *identity semantics*
 - You *must* override if you want *value* semantics
 - Otherwise don’t

Overriding toString()

```
final class PhoneNumber {  
    private final short areaCode;  
    private final short prefix;  
    private final short lineNumber;  
    ...  
    @Override public String toString() {  
        return String.format("(%03d) %03d-%04d",  
                               areaCode, prefix, lineNumber);  
    }  
}
```

```
Number jenny = ...;  
System.out.println(jenny);  
Prints: (707) 867-5309
```

equals Override Example

```
public final class PhoneNumber {  
    private final short areaCode;  
    private final short prefix;  
    private final short lineNumber;  
  
    @Override public boolean equals(Object o) {  
        if (!(o instanceof PhoneNumber)) // Does null check  
            return false;  
        PhoneNumber pn = (PhoneNumber) o;  
        return pn.lineNumber == lineNumber  
            && pn.prefix == prefix  
            && pn.areaCode == areaCode;  
    }  
  
    ...  
}
```

hashCode override example

```
public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    @Override public int hashCode() {
        int result = 17; // Nonzero is good
        result = 31 * result + areaCode; // Constant must be odd
        result = 31 * result + prefix;   // " " " "
        result = 31 * result + lineNumber; // " " " "
        return result;
    }

    ...
}
```

Alternative hashCode override

- **Less efficient, but otherwise equally good!**

```
public final class PhoneNumber {  
    private final short areaCode;  
    private final short prefix;  
    private final short lineNumber;  
  
    @Override public int hashCode() {  
        return arrays.hashCode(areaCode, prefix, lineNumber);  
    }  
  
    ...  
}
```

What does this print?

```
public class Name {  
    private final String first, last;  
    public Name(String first, String last) {  
        if (first == null || last == null)  
            throw new NullPointerException();  
        this.first = first; this.last = last;  
    }  
    public boolean equals(Name o) {  
        return first.equals(o.first) && last.equals(o.last);  
    }  
    public int hashCode() {  
        return 31 * first.hashCode() + last.hashCode();  
    }  
    public static void main(String[] args) {  
        Set<Name> s = new HashSet<>();  
        s.add(new Name("Mickey", "Mouse"));  
        System.out.println(  
            s.contains(new Name("Mickey", "Mouse")));  
    }  
}
```

- (a) true
- (b) false
- (c) It varies
- (d) None of the above

- Name overrides hashCode but not equals! The two Name instances are thus unequal.

How do you fix it?

- Replace the overloaded equals method with an overriding equals method.

```
@Override public boolean equals(Object o) {  
    if (!(o instanceof Name))  
        return false;  
    Name n = (Name) o;  
    return n.first.equals(first) && n.last.equals(last);  
}
```



10 Recall: Mutable and immutable classes (and defensive copying)



Invariants of a class

- Immutable classes: **Class whose instances cannot be modified**
 - Examples: `String`, `Integer`, `BigInteger`
 - How, why, and when to use them

Immutability as a type of Invariants

- How do we guarantee that these Tweet objects are immutable – that, once a tweet is created, its author, message, and date can never be changed?

```
/**
 * This immutable data type represents a tweet from Twitter.
 */
public class Tweet {

    public String author;
    public String text;
    public Date timestamp;

    /**
     * Make a Tweet.
     * @param author    Twitter user who wrote the tweet
     * @param text      text of the tweet
     * @param timestamp date/time when the tweet was sent
     */
    public Tweet(String author, String text, Date timestamp) {
        this.author = author;
        this.text = text;
        this.timestamp = timestamp;
    }
}
```

It's mutable...

- The first threat to immutability comes from the fact that clients can — in fact must — directly access its fields. So nothing's stopping us from writing code like this:
- **What's the effect of this code?**

```
Tweet t = new Tweet("justinbieber",  
                    "Thanks to all those believers out there inspiring me every day",  
                    new Date());  
t.author = "rbmllr";
```

- This is a trivial example of **representation exposure**, meaning that code outside the class can modify the representation directly.
- Rep exposure like this threatens not only invariants, but also representation independence.
- We can't change the implementation of Tweet without affecting all the clients who are directly accessing those fields.

To make it immutable...

- The **private** and **public** keywords indicate which fields and methods are accessible only within the class and which can be accessed from outside the class.
- The **final** keyword also helps by guaranteeing that the fields of this immutable type won't be reassigned after the object is constructed.

```
public class Tweet {  
  
    private final String author;  
    private final String text;  
    private final Date timestamp;  
  
    public Tweet(String author, String text, Date timestamp) {  
        this.author = author;  
        this.text = text;  
        this.timestamp = timestamp;  
    }  
  
    /** @return Twitter user who wrote the tweet */  
    public String getAuthor() {  
        return author;  
    }  
  
    /** @return text of the tweet */  
    public String getText() {  
        return text;  
    }  
  
    /** @return date/time when the tweet was sent */  
    public Date getTimestamp() {  
        return timestamp;  
    }  
}
```

How about this ...

- What's the effect of this code?

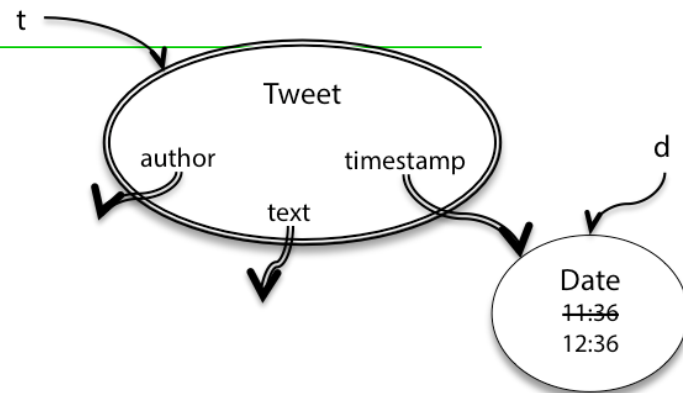
```
/** @return a tweet that retweets t, one hour later*/  
public static Tweet retweetLater(Tweet t) {  
    Date d = t.getTimestamp();  
    d.setHours(d.getHours()+1);  
    return new Tweet("rbmllr", t.getText(), d);  
}
```

- `retweetLater()` takes a tweet and should return another tweet with the same message (called a retweet) but sent an hour later.
- The `retweetLater()` method might be part of a system that automatically echoes funny things that Twitter celebrities say.

What's the problem?

■ What's the problem here?

- The `getTimestamp` call returns a reference to the same `Date` object referenced by tweet `t`. `t.timestamp` and `d` are aliases to the same mutable object.
- So when that date object is mutated by `d.setHours()`, this affects the date in `t` as well, as shown in the snapshot diagram.



- **Tweet's immutability invariant has been broken.**
- **The problem is that Tweet leaked out a reference to a mutable object that its immutability depended on.**
- **We exposed the rep, in such a way that Tweet can no longer guarantee that its objects are immutable.**
- **Perfectly reasonable client code created a subtle bug.**

How to solve it? --- Defensive copying

- We can patch this kind of rep exposure by using defensive copying: making a copy of a mutable object to avoid leaking out references to the rep.

```
public Date getTimestamp() {  
    return new Date(timestamp.getTime());  
}
```

- **Defensive copying is an approach of defensive programming**
 - Assume clients will try to destroy invariants --- May actually be true (malicious hackers), but more likely, honest mistakes
 - Ensure class invariants survive any inputs, to minimize mutability
 - 与第7章robustness联系起来

Copy and Clone()

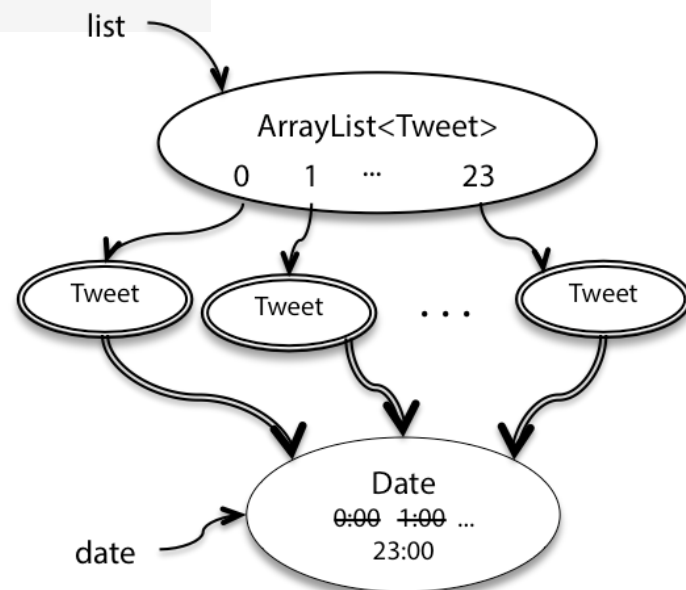
- **Mutable types often have a copy constructor that allows you to make a new instance that duplicates the value of an existing instance.**
 - In this case, `Date` 's copy constructor uses the timestamp value, measured in milliseconds since January 1, 1970.
 - As another example, `StringBuilder` 's copy constructor takes a `String` .
- **Another way to copy a mutable object is `clone()` , which is supported by some types but not all.**

Still rep exposure...

- What's the side-effect of this code?

```
/** @return a list of 24 inspiring tweets, one per hour today */
public static List<Tweet> tweetEveryHourToday () {
    List<Tweet> list = new ArrayList<Tweet>();
    Date date = new Date();
    for (int i = 0; i < 24; i++) {
        date.setHours(i);
        list.add(new Tweet("rbml1r", "keep it up! you can do it", date));
    }
    return list;
}
```

- The constructor of Tweet saves the reference that was passed in, so all 24 Tweet objects end up with the same time.



How to solve it? --- Again, defensive copying

```
public Tweet(String author, String text, Date timestamp) {  
    this.author = author;  
    this.text = text;  
    this.timestamp = new Date(timestamp.getTime());  
}
```

- In general, you should carefully inspect the argument types and return types of all your ADT operations. If any of the types are mutable, make sure your implementation doesn't return direct references to its representation.
- Doing that creates rep exposure.

Leave the responsibility to your clients?

- You may object that this seems wasteful. Why make all these copies of dates? Why can't we just solve this problem by a carefully written specification, like this?

```
/**
 * Make a Tweet.
 * @param author    Twitter user who wrote the tweet
 * @param text      text of the tweet
 * @param timestamp date/time when the tweet was sent. Caller must never
 *                  mutate this Date object again!
 */
public Tweet(String author, String text, Date timestamp) {
```

- Yes, it works, but the cost in your ability to reason about the program, and your ability to avoid bugs, is enormous.
- In the absence of compelling arguments to the contrary, it's almost always worth it for an abstract data type to guarantee its own invariants, and preventing rep exposure is essential to that.


To use immutable types, it's better!

- An even better solution is to prefer immutable types.
- If we had used an immutable date object, like `java.time.ZonedDateTime`, instead of the mutable `java.util.Date`, then these potential bugs would have disappeared and no further rep exposure would have been possible.

Immutable Wrappers around Mutable Data Types

- **The Java collections classes offer an interesting compromise: immutable wrappers.**
 - `Collections.unmodifiableList()` takes a (mutable) `List` and wraps it with an object that looks like a `List`, but whose mutators are disabled – `set()`, `add()`, `remove()`, etc. throw exceptions. So you can construct a list using mutators, then seal it up in an unmodifiable wrapper (and throw away your reference to the original mutable list, as discussed in Mutability & Immutability), and get an immutable list.
- **The downside here is that you get immutability at runtime, but not at compile time. Java won't warn you at compile time if you try to `sort()` this unmodifiable list. You'll just get an exception at runtime. But that's still better than nothing, so using unmodifiable lists, maps, and sets can be a very good way to reduce the risk of bugs.**

How to write an immutable class

- 
- Don't provide any mutators
 - Ensure that no methods may be overridden
 - Make all fields final
 - Make all fields private
 - Ensure security of any mutable components



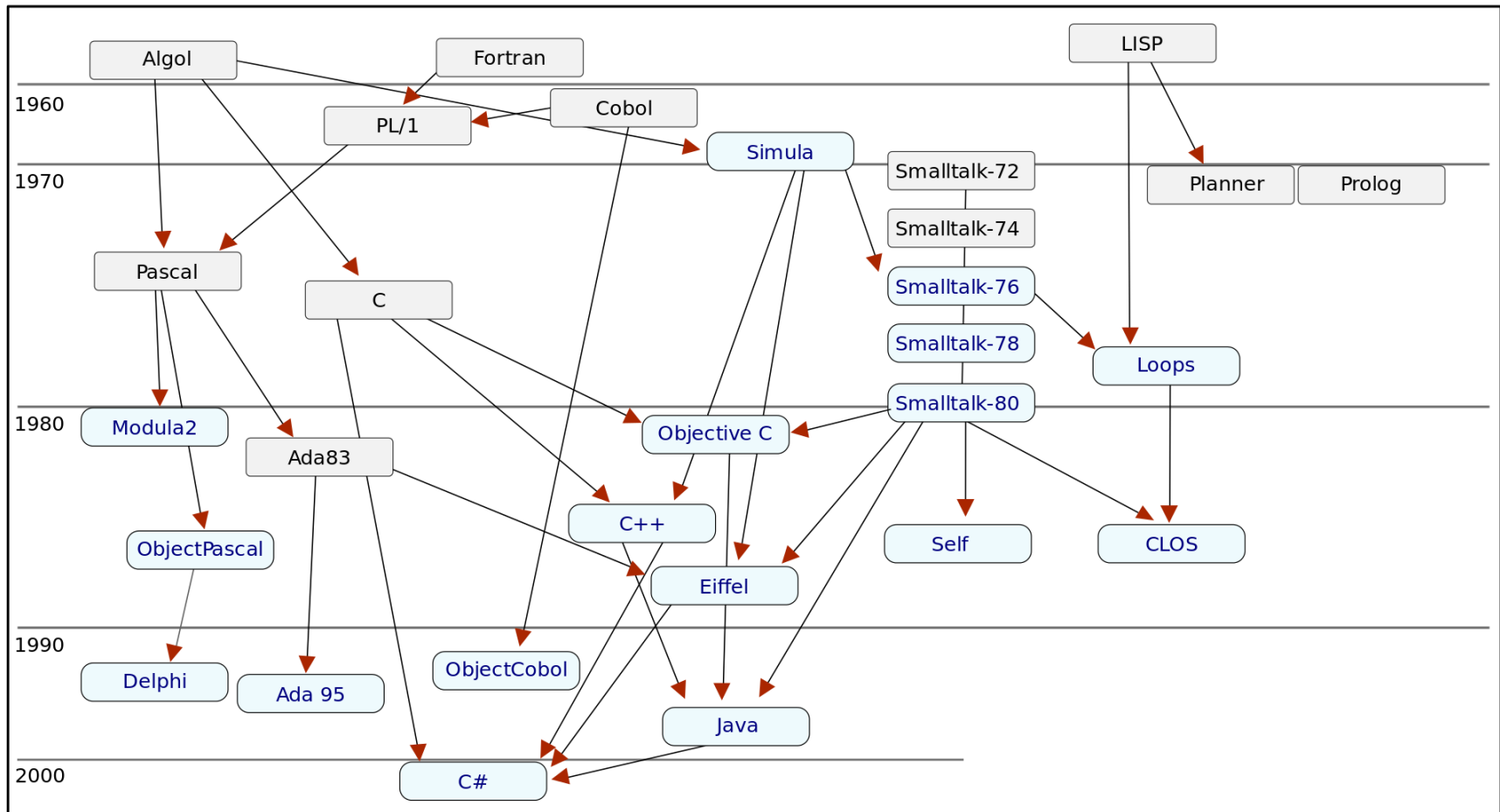
11 History of OOP



Simulation and the origins of OO programming

- **1960s: Simula 67** was the first object-oriented language developed by **Kristin Nygaard and Ole-Johan Dahl** at the Norwegian Computing Center, to support *discrete-event simulation*. (**Class, object, inheritance, etc**)
- The term "object oriented programming (OOP) " was first used by Xerox PARC in their Smalltalk language.
- 1980s: OOP had become prominent, and the primary factor in this is C++.
- Niklaus Wirth for modular programming and data abstraction, with Oberon and Modula-2;
- Eiffel and Java

History of OOP languages





Summary



Summary of Interface

- **Safe from bugs.** An ADT is defined by its operations, and interfaces do just that. When clients use an interface type, static checking ensures that they only use methods defined by the interface. If the implementation class exposes other methods — or worse, has visible representation — the client can't accidentally see or depend on them. When we have multiple implementations of a data type, interfaces provide static checking of the method signatures.
- **Easy to understand.** Clients and maintainers know exactly where to look for the specification of the ADT. Since the interface doesn't contain instance fields or implementations of instance methods, it's easier to keep details of the implementation out of the specifications.
- **Ready for change.** We can easily add new implementations of a type by adding classes that implement interface. If we avoid constructors in favor of static factory methods, clients will only see the interface. That means we can switch which implementation class clients are using without changing their code at all.



The end

March 25, 2019