哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

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

3.4 Object-Oriented Programming (OOP)

Xu Hanchuan
xhc@hit.edu.cn

April 1, 2018

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, method and interface

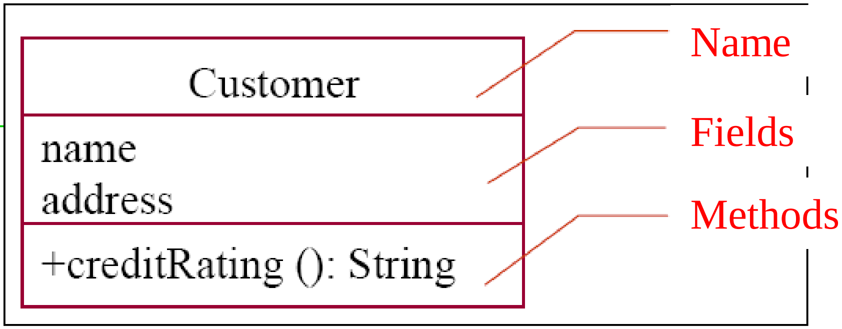
Object

- **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(踏板节奏), 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

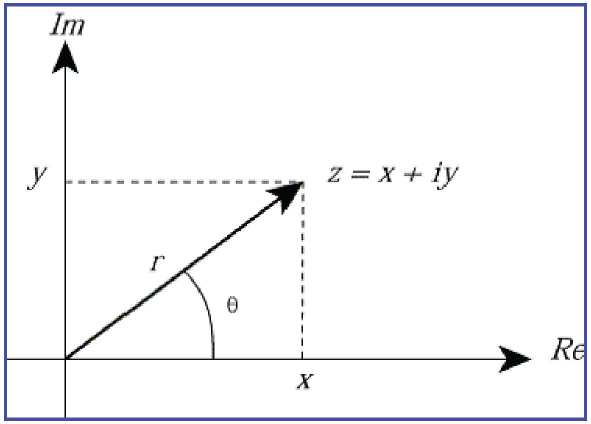
- **An object is a bundle of *state* and *behavior***
- **State – the data contained in the object.**
  - In Java, these are the fields(字段) of the object
- **Behavior – the actions supported by the object**
  - In Java, these are called methods
  - Method is just OO-speak for function
  - invoke a method = 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 ≈ where the object can be used
  - implementation ≈ 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 复数




```
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(*Static variable in Java)**: a variable associated with the class rather than with an instance of the class. You can also associate methods with a class--*class methods((*Static methods in Java).**
  - 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));
  }
}
```

Static vs. instance variables/methods of a class

- **Static methods are not associated with any particular instance of a class, while instance methods (declared without the static keyword) must be called on a particular object.**

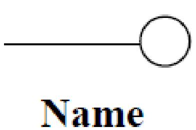
```
class Difference {

  public static void main(String[] args) {
    display();  //calling without object
    Difference t = new Difference();
    t.show();  //calling using object
  }

  static void display() {
    System.out.println("Programming is amazing.");
  }

  void show(){
    System.out.println("Java is awesome.");
  }
}
```

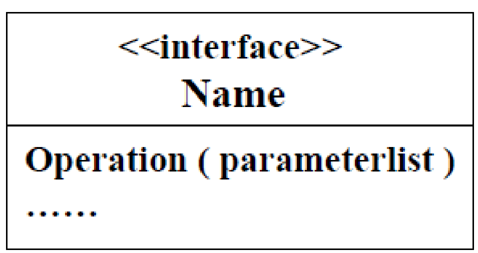
```
public class MyStaticMethods{

    private String name;
    private static String staticStr = "STATIC-STRING";

    public MyStatic (String n){
        this.name = n;
    }

    public static void testStaticMethod(){
        System.out.println("Hey... I am in static method..."); //you can call static variables here
        System.out.println(MyStaticMethods.staticStr);       //you can not call instance variables here.
    }

    public void testObjectMethod(){
        System.out.println("Hey i am in non-static method");//you can also call static variables here
        System.out.println(MyStaticMethods.staticStr);       //you can call instance variables here
        System.out.println("Name: "+this.name);
    }

    public static void main(String a[]){
        //By using class name, you can call static method
        MyStatic.testStaticMethod();
        MyStatic msm = new MyStatic ("Java2novice");
        msm.testObjectMethod();
    }
}
```

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 一个接口可以扩展其他接口
  - A class can implement multiple interfaces 一个类可以实现多个接口

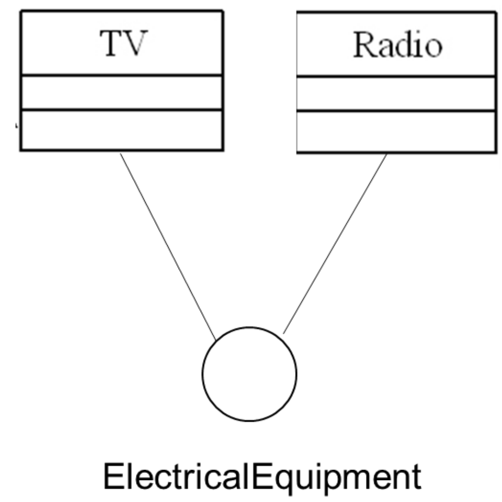


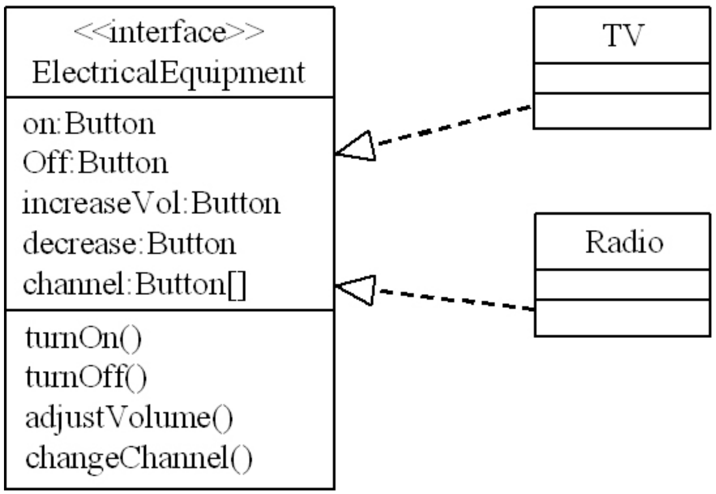



```java
public interface Complex {
    // No constructors, fields, or implementations!

    double realPart();
    double imaginaryPart();
    double r();
    double theta();

    Complex plus(Complex c);
    Complex minus(Complex c);
    Complex times(Complex c);
    Complex dividedBy(Complex c);
}
```

Interfaces and implementations

- **Multiple implementations of API can coexist**
  - Multiple classes can implement the same API
  - They can differ in performance and behavior,
- **In Java, an API is specified by *interface* or *class***
  - Interface provides only an API
  - An interface defines but does not implement API
  - Class provides an API and an implementation
  - 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<>();       // Do this…
HashSet<Criminal> senate = new HashSet<>();   // Not this
```

Another interface example: `MyString`

```java
/** 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`

```java
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 破坏了通过抽象建立的屏障**
  - 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

- **From Java 8， interfaces are allowed to contain static methods, so we can implement the creator operation valueOf as a static factory method in the interface MyString:**

```
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));
```

Using static factory instead of constructor

- **Hiding the implementation completely is a tradeoff, because sometimes the client wants a choice of implementations with different characteristics.**
- **That's why ArrayList and LinkedList are exposed by the Java library, because they vary in performance of operations like get() and insert().**

```
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`

Summary of Interface

- **Documentation for both the compiler and for humans**
  - Not only does an interface help the compiler catch ADT implementation bugs, but it is also much more useful for a human to read than the code for a concrete implementation. Such an implementation intersperses ADT-level types and specs with implementation details. (有助于编译器帮助检查ADT实习中的bug，也有助于用户脱离代码理解方法)

- **Allowing performance trade-offs**
  - Different implementations of the ADT can provide methods with very different performance characteristics. Different applications may work better with different choices, but we would like to code these applications in a way that is representation-independent. From a correctness standpoint, it should be possible to drop in any new implementation of a key ADT with simple, localized code changes.(可以根据需求选择合适的实现，实现性能的折中)

Summary of Interface

- **Methods with intentionally underdetermined specifications**
  - An ADT for finite sets could leave unspecified the element order one gets when converting to a list. Some implementations might use slower method implementations that manage to keep the set representation in some sorted order, allowing quick conversion to a sorted list. Other implementations might make many methods faster by not bothering to support conversion to sorted lists. (ADT的规格说明中，对方法的实现存在未明确指定，可以有多种方式实现)
- **Multiple views of one class**
  - A Java class may implement multiple interfaces. For instance, a user interface widget displaying a drop-down list is natural to view as both a widget and a list. The class for this widget could implement both interfaces. In other words, we don't implement an ADT multiple times just because we are choosing different data structures; we may make multiple implementations because many different sorts of objects may also be seen as special cases of the ADT, among other useful perspectives.(一个类可以实现多个接口，展现多个视图，是对Java不支持多继承的一种补偿)

Summary of Interface

- **More and less trustworthy implementations**
  - Another reason to implement an interface multiple times might be that it is easy to build a simple implementation that you believe is correct, while you can work harder to build a fancier version that is more likely to contain bugs. You can choose implementations for applications based on how bad it would be to get bitten by a bug. (通过对多个实现在性能和bug free方面的比较，进行实现的选择)

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内部数据和实现细节的隐藏程度是模块化设计质量评价的重要标准**
- **Well-designed code hides *all* implementation details**
  - Cleanly separates API from implementation
  - Modules communicate *only* through APIs
  - They are oblivious(不知道的) to each others' inner workings
- **Known as *information hiding* or *encapsulation,* a fundamental tenet(原则) of software design.**

Benefits of information hiding

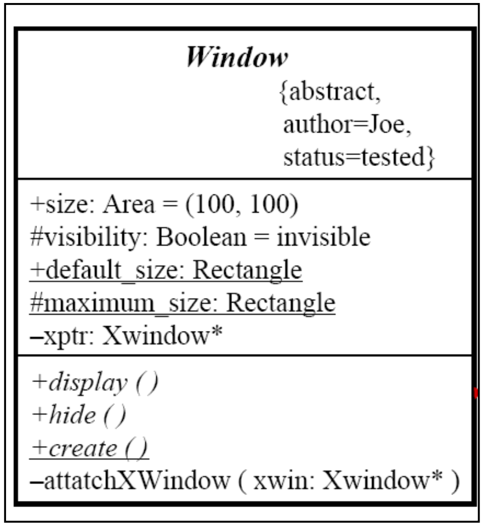
- **Decouples** the classes that comprise a system
  - Allows them to be developed, tested, optimized, used, understood, and modified in isolation
- **Speeds up system development**
  - Classes can be developed in parallel
- **Eases burden of maintenance**
  - Classes can be understood more quickly and 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

Visibility modifiers for members

- `private` – Accessible *only* from declaring class
- `protected` – Accessible from subclasses of declaring class (and within package)
- `public` – Accessible from anywhere




```
class OrdinaryComplex implements Complex {
  private double re;  // Real Part
  private 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)   { ... }
}
```

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

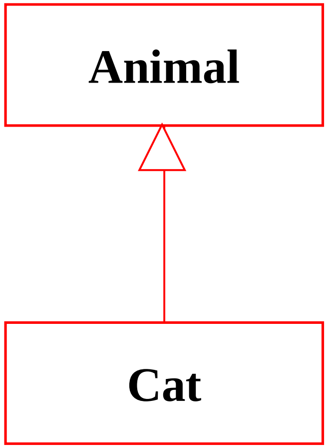
`Class A extends B`

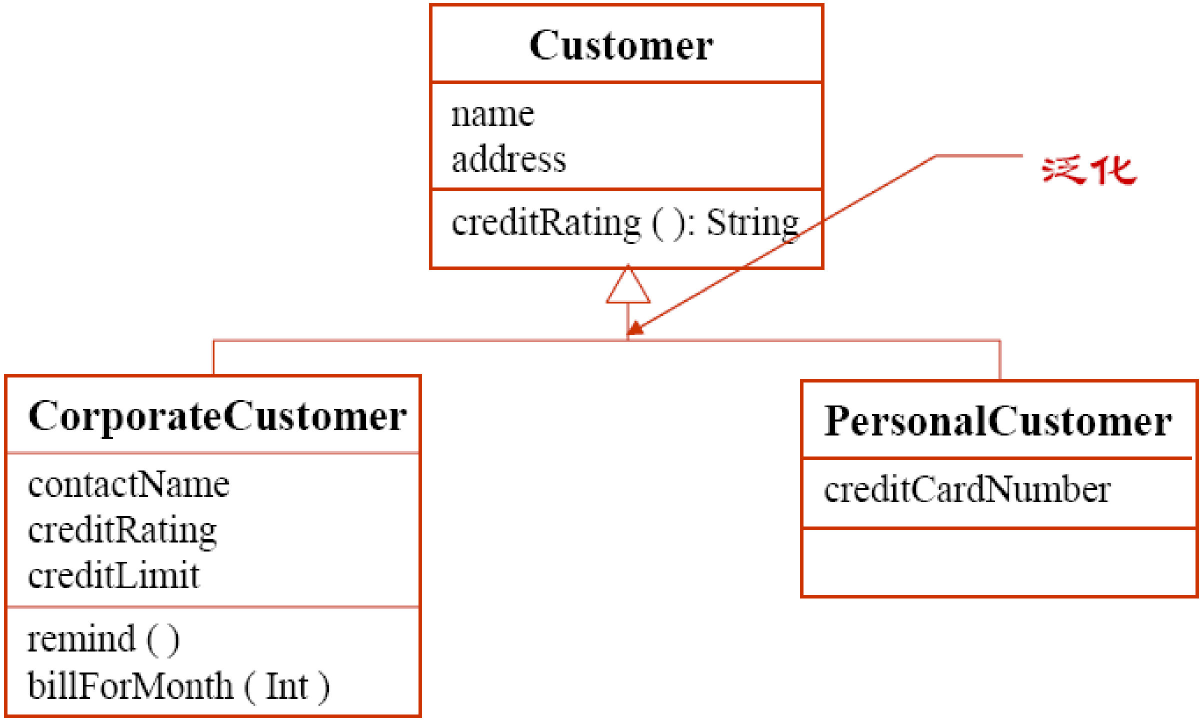
- **Subtyping is for polymorphism**
  - Accessing objects the same way, but getting different behavior
  - **Subtype is substitutable for supertype**

`Class A implements I`

- **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

Inheritance and subtyping

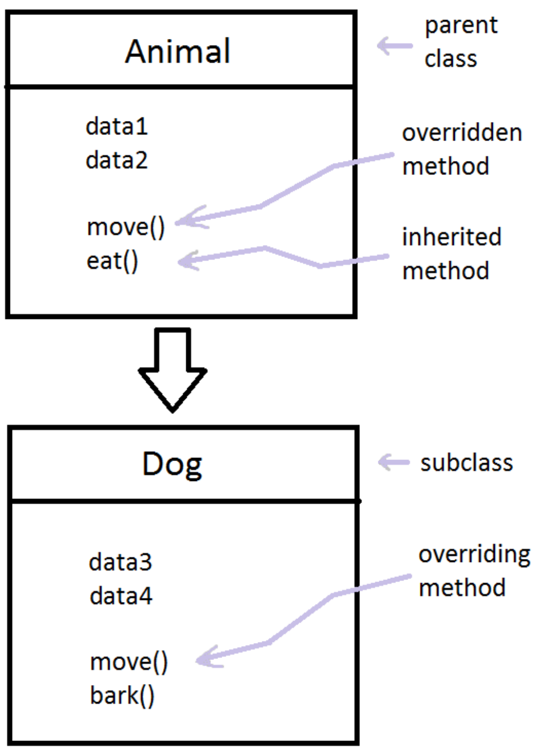




(1) Overriding（覆盖/重写)

Overriding (覆盖/重写)

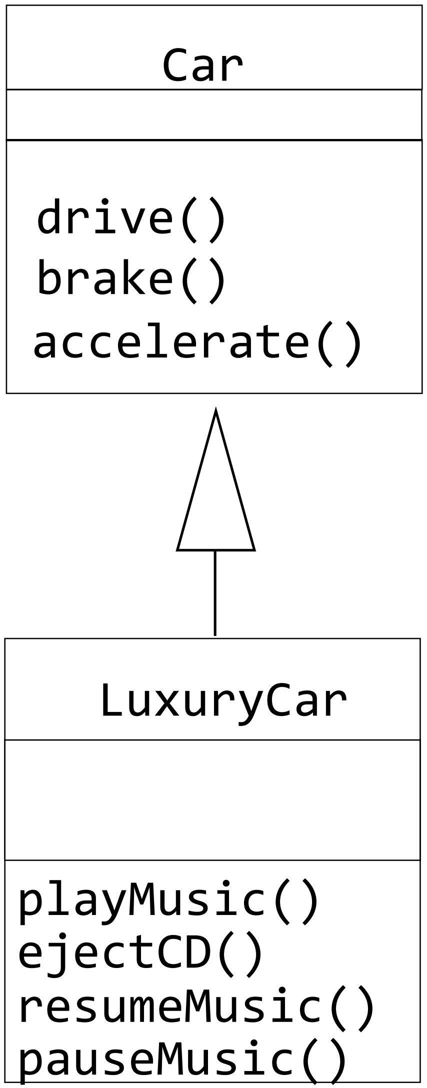
- **Method overriding is a language feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes.**
  - The implementation in the subclass overrides (replaces) the implementation in the superclass 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.



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() {…}
}
```

Strict Inheritance and Rewriteable Methods

```
class Device {
     int serialnr;
     public final void help() {….}
     public void setSerialNr(int n) {
        serialnr = n;
     }
}

class Valve extends Device {
     Position s;
     public void on() {
        ….
     }
}
```

help() not overwritable

setSerialNr() overwritable

Example: Overwriting a Method

```
class Device {
     int serialnr;
     public final void help() {….}
     public void setSerialNr(int n) {
       serialnr = n;
     }
}
class Valve extends Device {
     Position s;
     public void on() {
       ….
     }
}
```

```
class Device {
     int serialnr;
     public final void help() {….}
     public void setSerialNr(int n) {
       serialnr = n;
     }
}

class Valve extends Device {
     Position s;
     public void on() {
       …
     }
     public void setSerialNr(int n) {
        serialnr = n + s.serialnr;
     }
}
```

Rewriteable Methods are set to empty

```
class Device {
     int serialnr;
     public void setSerialNr(int n) {}
}
class Valve extends Device {
   Position s;
   public void on() {
      …..
   }
   public void setSerialNr(int n) {
      seriennr = n + s.serialnr;
   }
} // class Valve
```

I expect that the method `setSerialNr()` will be overwritten. I only write an empty body(called stub)

Overwriting of the method `setSerialNr()` of Class Device

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. 子类中可以通过super关键字调用父类中被重写的方法**

```
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.
      }
```

Extended reuse with super

```
public abstract class AbstractAccount implements Account {
   protected long balance = 0;
   public boolean withdraw(long amount) {
       // withdraws money from account (code not shown)
   }
}

public class ExpensiveCheckingAccountImpl
      extends AbstractAccount implements CheckingAccount {
   public boolean withdraw(long amount) {
      balance -= HUGE_ATM_FEE;
      boolean success = super.withdraw(amount)
      if (!success)
         balance += HUGE_ATM_FEE;
      return success;
   }
}
```

Overrides `withdraw` but also uses the superclass `withdraw` 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;
   }

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

调用父类的构造方法时，必须是构造方法的第一行

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

Invokes another constructor in this same class

Bad Use of Overriding 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 { …`

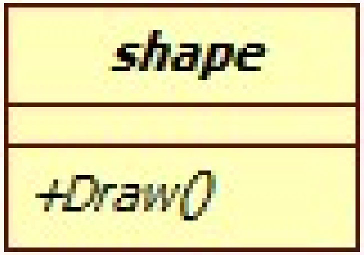
Tips for overriding a method

- **If you want to override a method:**
  - Make sure signatures match
  - Use `@Override` so compiler has your back(**compiler 会检查覆盖方法和被覆盖的方法签名是否完全一致**)
  - *Do* copy-and-paste declarations (or let IDE do it for you)

(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**



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

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.(多态类型，其操作可以用于其他类型的值)
  - **Ad hoc polymorphism**: when a function denotes different and potentially heterogeneous(异构的) implementations depending on a limited range of individually specified types and combinations. Ad hoc polymorphism is supported in many languages using function overloading. 一个函数可以有多个同名的实现
  - **Parametric polymorphism(参数多态性)**: 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. 一个类型名字可以代表多个类型
  - **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) 一个变量名字可以代表多个类的实例

HARBIN INSTITUTE OF TECHNOLOGY

哈爾濱工業大學

(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;
        }

}
```

Overloading 重载

- **Overloaded methods let you reuse the same method name in a class, but with different arguments (and optionally, a different return type).**
- **Overloading a method often means you're being a little nicer to those who call your methods, because your code takes on the burden of coping with different argument types rather than forcing the caller to do conversions prior to invoking your method.**

```
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;
        }

}
```

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(参数数量) 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

```
public class Adder {
  public int addThem(int x, int y) {
    return x + y;
  }
  public double addThem(double x,
         double y) {
    return x + y;
  }
}
```

```
public class TestAdder {
  public static void main (String [] args) {
    Adder a = new Adder();
    int b = 27;
    int c = 3;

    // Which addThem is invoked?
    int result = a.addThem(b,c);
    double doubleResult = a.addThem(22.5,9.3);

    System.out.println (result);
    System.out.println (doubleResult);
   }
}
```

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()
      {System.out.println("I'm an animal. I like eating everything!");}
}
class Horse extends Animal {
   public void eat(String food)
      {System.out.println("I'm a horse. I like eating "+ food);}

   public void eat()
      {System.out.println(“I‘m a horse. I like eating grass! "}
}
```

Method Invocation Code	Result
Animal a = new Animal(); a.eat();	**I'm an animal. I like eating everything!**
Horse h = new Horse(); h.eat();	**I'm a horse. I like eating grass!**
Animal ah = new Horse(); ah.eat();	**I'm a horse. I like eating grass! Polymorphism works- the actual object type(Horse), not the reference type(Animal), is used to determine which eat() is called.**
Horse he = new Horse(); he.eat("Apples!");	**I'm a horse. I like eating Apples! The overloaded eat(String s) method in Horse is invoked.**
Animal a2 = new Animal(); a2.eat("treats");	**Compiler error! Animal class doesn’t have an eat() method that takes a String**
Animal ah2 = new Horse(); ah2.eat("Carrots");	**Compiler error! Compiler still looks only at the reference, and sees that Animal doesn’t have an eat() method that takes a String.**

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

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
  }
}

class B {
  public void p(int i) {
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
  }
}

class B {
  public void p(int i) {
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

The method p(int i) in class A overrides the same method defines in class B.

The method p(int i) in class A overloads the same method defines in class B.

Overriding vs. Overloading

- **Do not confuse overriding a method in a derived class(派生类) with overloading a method name**
  - When a method is overridden, the new method definition given in the derived class has the exact same number and types of parameters as in the base class
  - When a method in a derived class has a different signature from the method in the base class, that is overloading
  - Note that when the derived class overloads the original method, it still inherits the original method from the base class as well

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's 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;
```

```
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;
```

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

Generics in Java

- A **type variable(类型变量)** is an unqualified(无限制的) identifier.
  - They are introduced by **generic class declarations, generic interface declarations, generic method declarations, and by generic constructor declarations. (范型的四种使用方式)**
- A **class** is generic if it declares one or more type variables.
  - These type variables are known as the **type parameters(类型参数)** of the class.
  - It defines one or more type variables that act as parameters.
  - A generic class declaration defines a set of parameterized types, one for each possible invocation of the type parameter section.
  - All of these parameterized types share the same class at runtime.

Generics in Java

- An **interface** is generic if it declares one or more type variables.
  - These type variables are known as the type parameters of the interface.
  - It defines one or more type variables that act as parameters.
  - A generic interface declaration defines a set of types, one for each possible invocation of the type parameter section.
  - All parameterized types share the same interface at runtime.
- A **method** is generic if it declares one or more type variables.
  - These type variables are known as the formal type parameters of the method.
  - The form of the formal type parameter list is identical to a type parameter list of a class or interface.

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 = papersStr.get(0);
                System.out.println(str);

                PapersJar<Integer>papersInt = new PapersJar<>();
                papersInt.add(new Integer(100));
                Integer integerObj = 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**

```java
// 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**

```java
// 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**

```java
// 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`.

```java
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);

    // ...
}
```

```java
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(通配符)**
  - 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**
  - Write code once and only once
  - Superclass features implicitly available in subclass

`Class A extends B`

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

`Class A implements I`

- **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. (编译器会检查，确保接口中的所有方法在实现类中均被实现)**

Static checking on subtypes

- **But the compiler cannot check that we haven't weakened the specification in other ways: (规格说明的强弱，编译器无法判断)**
  - 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.**

Variation in the real world

- **Two types of "Bank Account"**

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


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

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 {
}
```

Implementation inheritance for code reuse

```
public abstract class AbstractAccount
        implements Account {
    protected long balance = 0;
    public long getBalance() {
        return balance;
    }
    abstract public void monthlyAdjustment();
    // other methods…
}

public class CheckingAccountImpl
        extends AbstractAccount
        implements CheckingAccount {
    public void monthlyAdjustment() {
        balance -= getFee();
    }
    public long getFee() { … }
}
```

An `abstract` class is missing the implementation of one or more methods

`Protected` elements are visible in subclasses

An `abstract` method is left to be implemented in a subclass

No need to define `getBalance()`, and the code is inherited from `AbstractAccount`

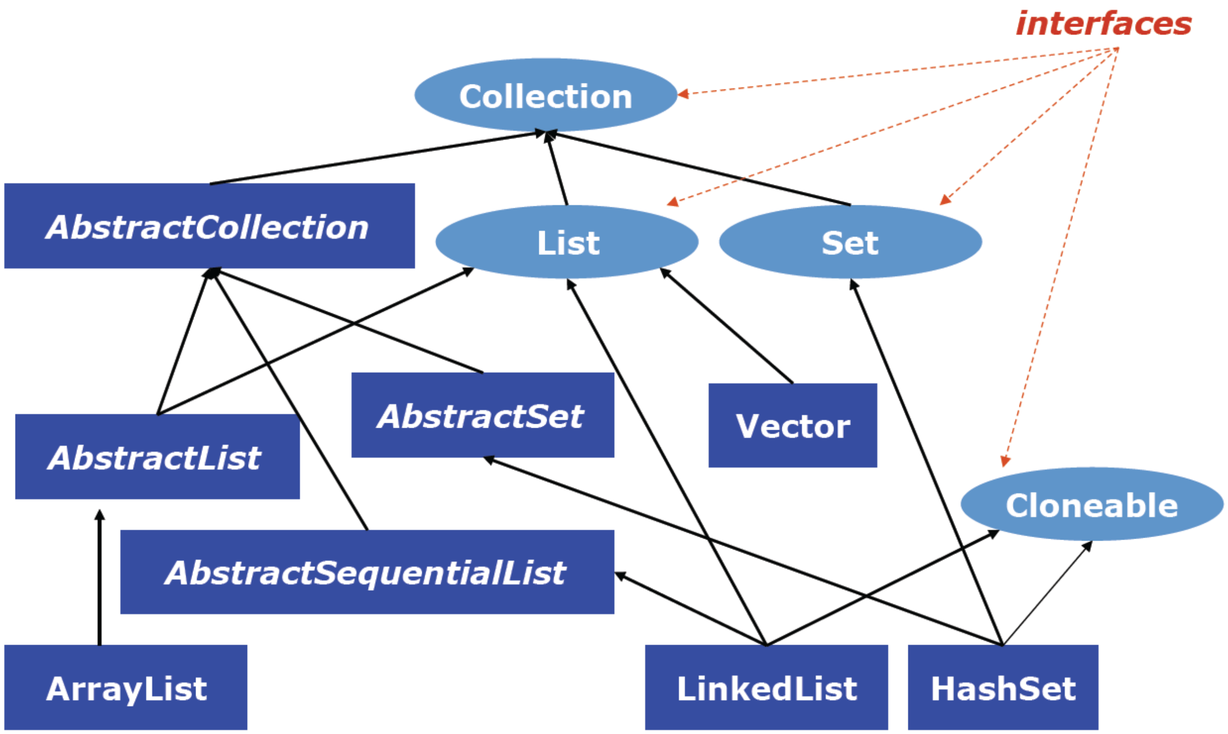
Subtype polymorphism

- **Different kinds of objects can be treated uniformly by client code （客户端可以对不同类型的对象，使用统一的方式进行处理)**
- **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 类型转换

- **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 避免向下类型转换
  - Never downcast within superclass to a subclass

instanceof

- **Operator that tests whether an object is of a given class(instanceof 是操作符，不是方法)**

```
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, and never use `instanceof()` in a superclass to check type against subclass.**
- **instanceof is disallowed anywhere except for implementing equals**

Behavioral subtyping **- discussed in 5.2**

- **Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.**

— —Barbara Liskov

- **Compiler-enforced rules in Java:**
  - Subtypes can add, but not remove methods
  - Concrete class must implement all undefined methods
  - Overriding method must return same type or subtype
  - Overriding method must accept the same parameter types
  - Overriding method may not throw additional exceptions
- **Also applies to specified behavior:**
  - Same or stronger invariants
  - Same or stronger postconditions for all methods
  - Same or weaker preconditions for all methods

Liskov Substitution Principle (LSP)

Behavioral subtyping (LSP)

- **Subclass fulfills the same invariants (and additional ones)**
- **Overridden method has the same pre and postconditions**

```
abstract class Vehicle {
        int speed, limit;

        //@ invariant speed < limit;

        //@ requires speed != 0;
        //@ ensures speed < \old(speed)
        void brake();
}
```

```
class Car extends Vehicle {
        int fuel;
        boolean engineOn;
        //@ invariant speed < limit;
        //@ invariant fuel >= 0;

        //@ requires fuel > 0 && !engineOn;
        //@ ensures engineOn;
        void start() { … }

        void accelerate() { … }

        //@ requires speed != 0;
        //@ ensures speed < \old(speed)
        void brake() { … }
}
```

Behavioral subtyping (LSP)

- **Subclass fulfills the same invariants (and additional ones)**
- **Overridden method `start` has weaker precondition**
- **Overridden method `brake` has stronger postcondition**

```
class Car extends Vehicle {
        int fuel;
        boolean engineOn;
        //@ invariant fuel >= 0;

        //@ requires fuel > 0 && !engineOn;
        //@ ensures engineOn;
        void start() { … }

        void accelerate() { … }

        //@ requires speed != 0;
        //@ ensures speed < old(speed)
        void brake() { … }
}
```

```
class Hybrid extends Car {
        int charge;
        //@ invariant charge >= 0;

        //@ requires (charge > 0 || fuel > 0)
                   && !engineOn;
        //@ ensures engineOn;
        void start() { … }

        void accelerate() { … }

        //@ requires speed != 0;
        //@ ensures speed < \old(speed)
        //@ ensures charge > \old(charge)
        void brake() { … }
}
```

Behavioral subtyping (LSP)

- **How about these two classes? Is LSP satisfied?**

```
class Rectangle {
        int h, w;

        Rectangle(int h, int w) {
                this.h=h; this.w=w;
        }

        //methods
}
```

```
class Square extends Rectangle {
        Square(int w) {
                super(w, w);
        }
}
```

```
class Rectangle {
        //@ invariant h>0 && w>0;
        int h, w;

        Rectangle(int h, int w) {
                this.h=h; this.w=w;
        }

        //methods
}
```

```
class Square extends Rectangle {
        //@ invariant h>0 && w>0;
        //@ invariant h==w;
        Square(int w) {
                super(w, w);
        }
}
```

Behavioral subtyping (LSP)

- **How about these two classes? Is LSP satisfied?**

```
class Rectangle {
        //@ invariant h>0 && w>0;
        int h, w;

        Rectangle(int h, int w) {
                this.h=h; this.w=w;
        }

        //@ requires factor > 0;
        void scale(int factor) {
                w=w*factor;
                h=h*factor;
        }
}
```

```
class Square extends Rectangle {
        //@ invariant h>0 && w>0;
        //@ invariant h==w;
        Square(int w) {
                super(w, w);
        }
}
```

Behavioral subtyping (LSP)

- **How about these two classes? Is LSP satisfied?**

```
class Rectangle {
        //@ invariant h>0 && w>0;
        int h, w;

        Rectangle(int h, int w) {
            this.h=h; this.w=w;
        }

        //@ requires factor > 0;
        void scale(int factor) {
            w=w*factor;
            h=h*factor;
        }
        //@ requires neww > 0;
        void setWidth(int neww) {
            w=neww;
        }
}
```

```
class Square extends Rectangle {
        //@ invariant h>0 && w>0;
        //@ invariant h==w;
        Square(int w) {
            super(w, w);
        }
}
```

```
class GraphicProgram {
  void scaleW(Rectangle r, int factor) {
    r.setWidth(r.getWidth() * factor);
  }
}
```

Invalidates stronger invariant (w==h) in subclass

7 Dynamic dispatch

Dynamic dispatch动态分派

- **Dynamic dispatch is the process of selecting which implementation of a polymorphic operation (method or function) to call at run time. 决定在运行时，一个具有多态的操作，哪个具体实现被选择执行**
  - 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`.**

(基于dividend的类型选择具体实现，而不考虑divisor的值和类型)

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.

Example

```
class Game {
 public void type(){
    System.out.println("Indoor & outdoor"); }
}

Class Cricket extends Game {
 public void type() {
    System.out.println("outdoor game"); }

 public static void main(String[] args) {
   Game gm = new Game();
   Cricket ck = new Cricket();
   gm.type();
   ck.type();
   gm=ck;       //gm refers to Cricket object
   gm.type();   //calls Cricket's version of type
 }
}
```

Upcasting: a Parent class variable refers to Child class object

8 Delegation and composition

(will be discussed in 5.2)

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*** **see 3.5**
  - You *must* override if you want *value* semantics
  - Otherwise don't

Overriding `toString()`

- `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

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

System.out.println(pn);

//it will call the toString() in object class
//the result maybe is like “PhoneNumber@70dea4e”
```

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`

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

Invariants of a class

- **Class invariants are critical properties of the fields of an object**
  - Established by the constructor
  - Maintained by public method invocations, and may be invalidated temporarily during method execution.

- 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 beliebers 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.

```java
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;
    }

}
```

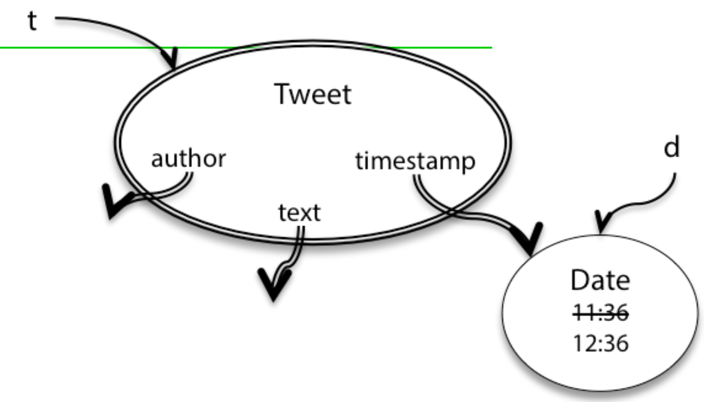
How about this ...

- **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.

```java
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

  - See 7.3

Copy and Clone()

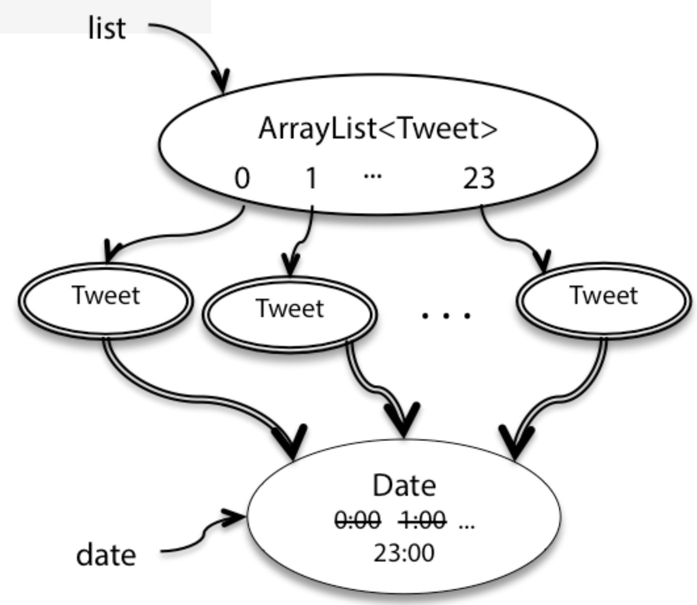
- **Mutable types often have a copy constructor that allows you to make a new instance that duplicates the value of an existing instance.**
  - In this case, `Date` 's copy constructor uses the timestamp value, measured in milliseconds since January 1, 1970.
  - As another example, `StringBuilder` 's 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?**

```java
/** @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("rbmllr", "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.

Another example about mutable Date

```
public final class Period {
   private final Date start, end; // Invariant: start <= end

   /**
    * @throws IllegalArgumentException if start > end
    * @throws NullPointerException if start or end is null
    */
   public Period(Date start, Date end) {
      if (start.after(end))
          throw new IllegalArgumentException(start + " > " + end);
      this.start = start;
      this.end   = end;
   }

   public Date start() { return start; }
   public Date end()   { return end; }
   ... // Remainder omitted
}
```

- **How about this code?**

```
Date start = new Date();  // (The current time)
Date end   = new Date();  //   "      "       "
Period p = new Period(start, end);
end.setYear(78);   // Modifies internals of p!
```

How to solve it?

- **By defensive copying**

```
// Repaired constructor - defensively copies parameters
public Period(Date start, Date end) {
   this.start = new Date(start.getTime());
   this.end   = new Date(end.getTime());
   if (this.start.after(this.end))
     throw new IllegalArgumentException(start +“ > "+ end);
}
```

- **How about this code?**

```
Period p = new Period(new Date(), new Date());
Date d = p.end();
p.end.setYear(78); // Modifies internals of p!
```

- **To solve it?**

```
// Repaired accessors - defensively copy fields
public Date start() {
    return new Date(start.getTime());
}
public Date end() {
    return new Date(end.getTime());
}
```

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.**

Summary

- **Don’t incorporate mutable parameters into object; make defensive copies**
- **Return defensive copies of mutable fields…**
  - Return new instance instead of modifying
- **Or return *unmodifiable view* of mutable fields**
- **Real lesson – use immutable components, to eliminate the need for defensive copying**

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

Immutable class example

```java
public final class Complex {
    private final double re, im;

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

    // Getters without corresponding setters
    public double realPart()      { return re; }
    public double imaginaryPart() { return im; }

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

Immutable class example

```
    @Override public boolean equals(Object o) {
        if (!(o instanceof Complex)) return false;
        Complex c = (Complex)o;
        return Double.compare(re, c.re) == 0 &&
               Double.compare(im, c.im) == 0;
    }

    @Override public int hashCode() {
       return 31*Double.hashCode(re) + Double.hashCode(im);
    }

    @Override public String toString() {
        return String.format("%d + %di”, re, im)";
    }
}
```

Advantages of immutable classes

- **Simplicity**
- **Inherently Thread-Safe**
- **Can be shared freely**
- **No need for defensive copies**
- **Excellent building blocks**

When to make classes immutable

- **Always, unless there's a good reason not to**
- Always make small “value classes” immutable!
  - Examples: Color, PhoneNumber, Unit
  - Date and Point were mistakes!
  - Experts often use long instead of Date

When to make classes mutable

- **Class represents entity whose state changes**
  - Real-world - BankAccount, TrafficLight
  - Abstract - Iterator, Matcher, Collection
  - Process classes - Thread, Timer
- **If class must be mutable, *minimize mutability***
  - Constructors should fully initialize instance
  - Avoid reinitialize methods

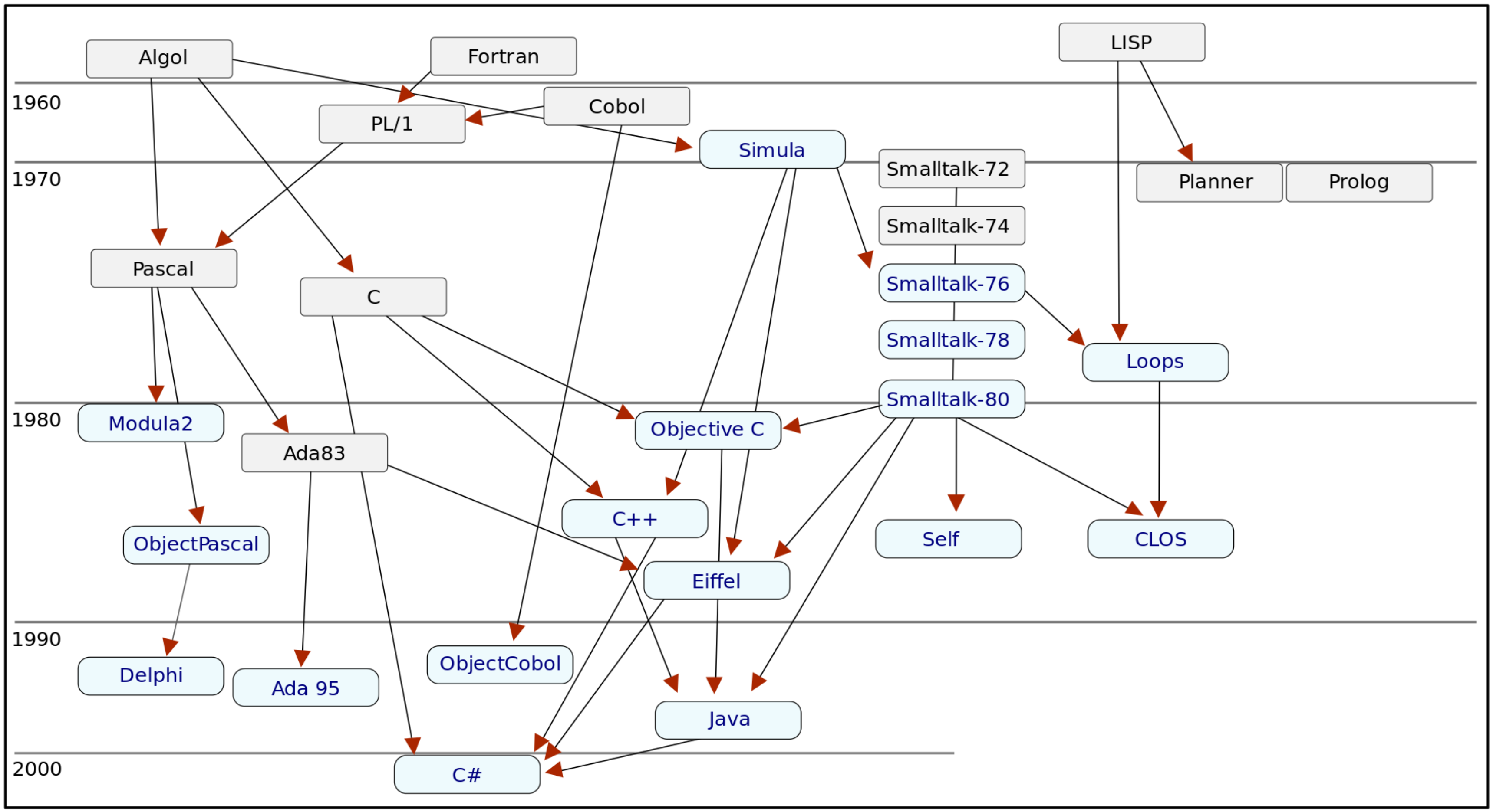
哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

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 languanges



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

April 1, 2018