哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

7 Object-Oriented Programming (OOP)
面向对象的编程

Xu Hanchuan
xhc@hit.edu.cn

May 10, 2022

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.**
- **用OOP/接口/类实现ADT**

第6部分 学习了ADT理论
本节学习ADT的具体实现技术：OOP

Outline

- **Basic concepts: object, class, attribute, method, interface, and enumerations OOP的基本概念**
- **Distinct features of OOP**

  Encapsulation and information hiding 封装与信息隐藏

  Inheritance and overriding 继承与重写

  Polymorphism, subtyping and overloading 多态、子类型、重载

  *Static and Dynamic dispatch 静态与动态分派
- **Some important `Object` methods in Java**
- **Designing good classes**
- **History of OOP**
- **Summary**

Reading

- **MIT 6.031：12**
- **CMU 17-214：Sep 3、Sep 10、Sep 19、Sep 24**
- **Java编程思想：第7-9章**
- **代码大全：第6章**
- **Effective Java：第3-4、7-9章**

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

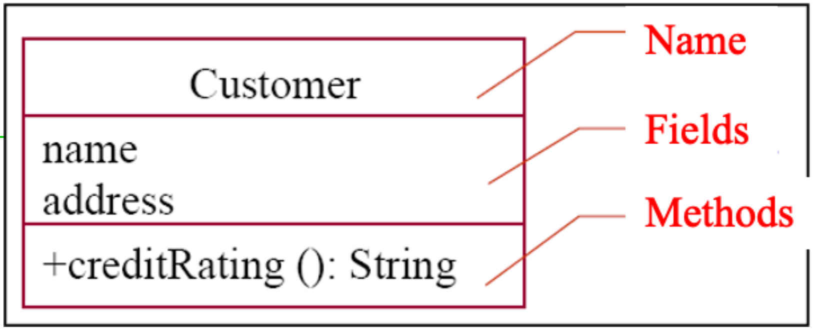
Object

- **Real-world objects share two characteristics: they all have *states* and *behaviors*.**
- **Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of OOP.**
  - Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail).
  - Bicycles 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 OOP:**
  - What possible states can this object be in? 状态有哪些？
  - What possible behavior can this object perform? 行为有哪些？

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 ≈ 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

Static vs. instance variables/methods of a class

- ***Class variable* 类成员变量: a variable associated with the class rather than with an instance of the class. You can also associate methods with a class--*class methods*类方法.**
  - 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* 实例方法 and *instance variables* 实例成员变量.**
  - 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);
    }
}
```

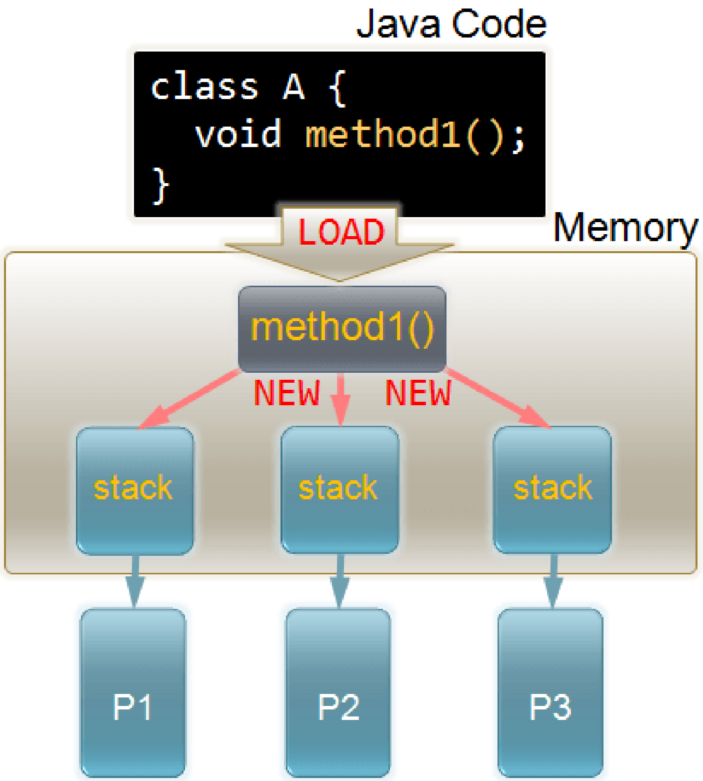
```
class Another {
  public static void main(String[] args) {
    int result;
    result = Math.min(10, 20);
    System.out.println(result);
    System.out.println(Math.max(100, 200));
  }
}
```

```java
public class MyStatic {

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

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

    public static void testStaticMethod(){
         //you can call static variables here, can not call instance variables
        System.out.println(MyStatic.staticStr);
    }

    public void testObjectMethod(){
        //you can also call static and instance variables here
        System.out.println(MyStatic.staticStr);
        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();
    }
}
```

静态方法无法直接调用非静态成员

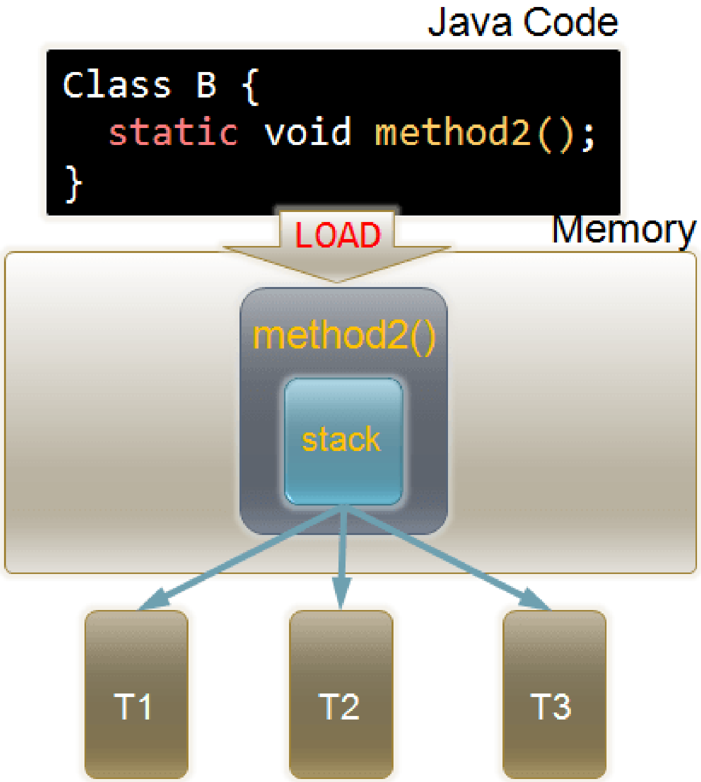
Static and Instance methods

Instance Methods



```
class A {
  void method1();
}
```

Static/Class Methods

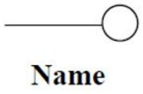


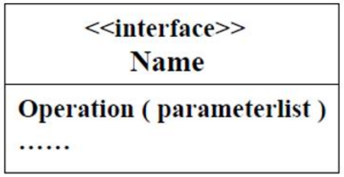
```
Class B {
  static void method2();
}
```

2 Interface and Enumerations

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
  - **Interface和Class: 定义和实现ADT**
  - 接口中只有方法的定义，没有实现
  - 接口之间可以继承与扩展
  - 一个类可以实现多个接口（从而具备了多个接口中的方法）
  - 一个接口可以有多种实现类





Java interfaces and classes

- **Interfaces vs. classes** **接口：确定ADT规约；类：实现ADT**
  - Interface: specifies expectations
  - Class: delivers on expectations (the implementation)
- **Classes *do* define types** **也可以不需要接口直接使用类作为ADT，既有ADT定义也有ADT实现**
  - Public class methods usable like interface methods
  - Public fields directly accessible from other classes
- **But prefer the use of interfaces** 实际中更倾向于使用接口来定义变量
  - 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
```

An Example

This violates the spec for `Set`, its immutability, so `ArraySet` is not a legal implementation of `Set`.

```
/** Represents an immutable set of elements of type E. */
   public interface Set<E> {
       /** make an empty set */
       public Set();
       /** @return true if this set contains e as a member */
       public boolean contains(E e);
       /** @return a set which is the union of this and that */
       public ArraySet<E> union(Set<E> that);
   }
```

Java interfaces can’t have constructors.

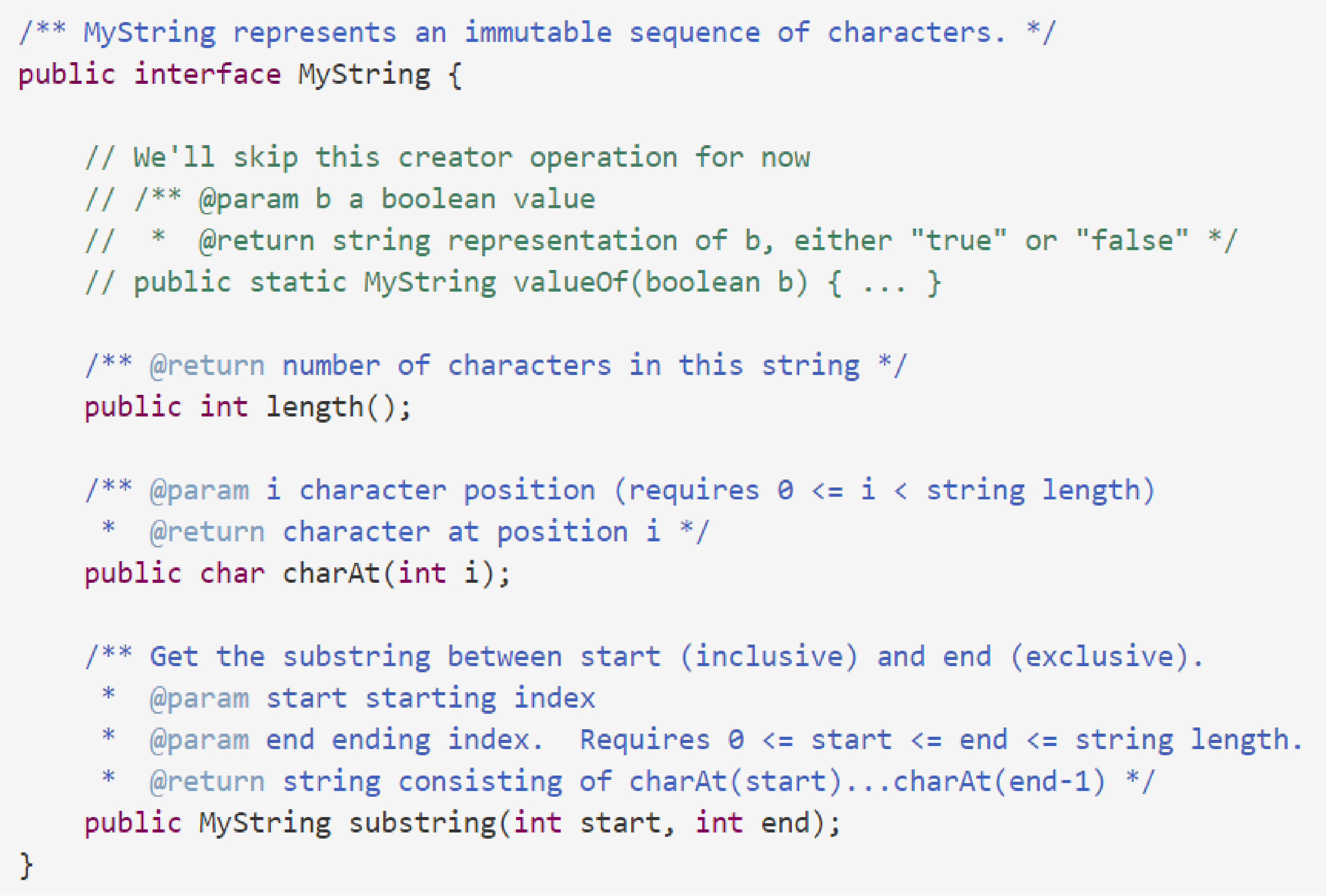
It isn’t representation-independent.

```
   /** Implementation of Set<E>. */
   public class ArraySet<E> implements Set<E> {
       /** make an empty set */
       public ArraySet() { ... }
       /** @return a set which is the union of this and that */
       public ArraySet<E> union(Set<E> that) { ... }
       /** add e to this set */
       public void add(E e) { ... }
   }
```

Java allowes classes to have more methods than the interface

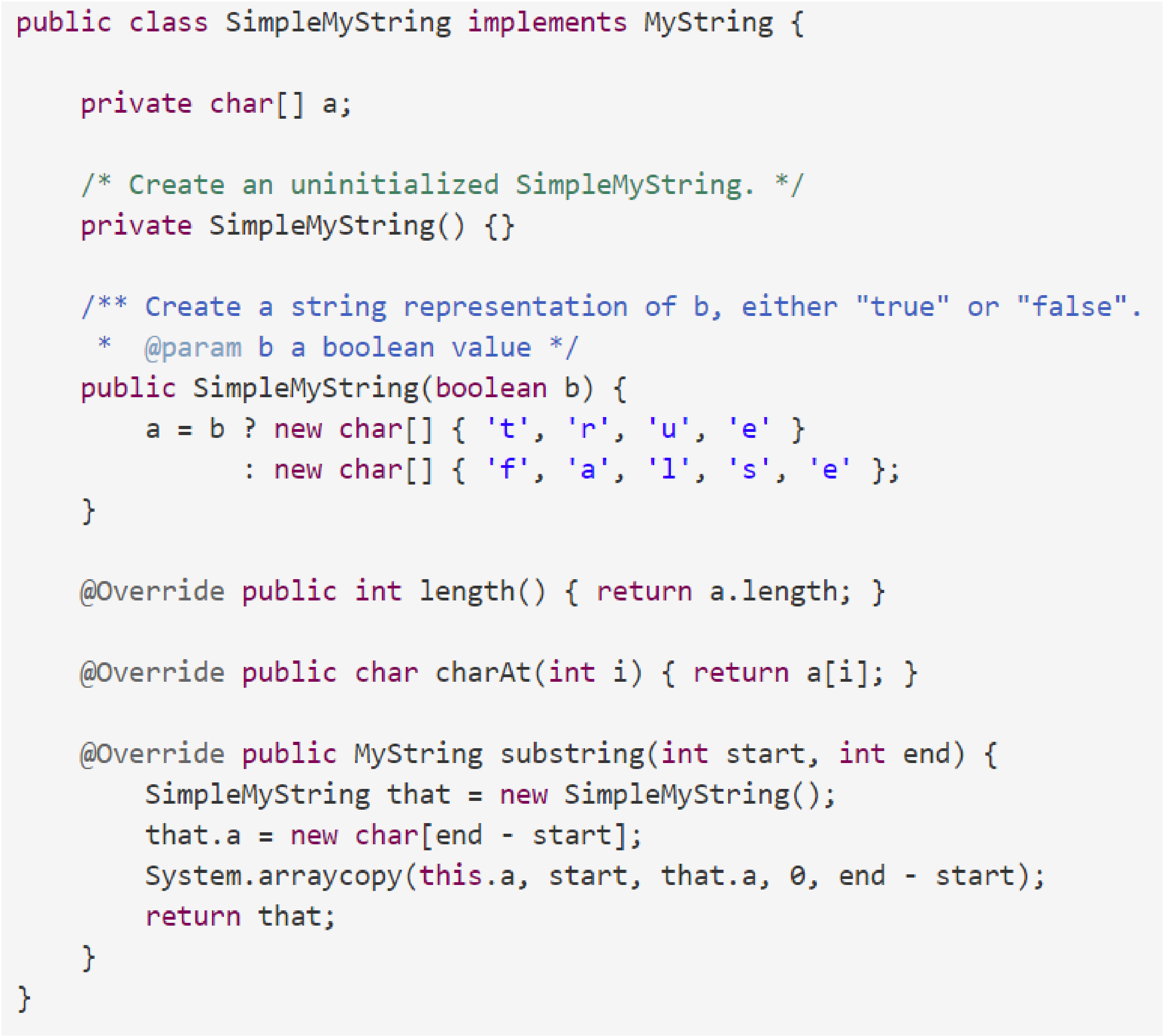
It’s missing the `contains()` method.

Using Interface+Class for ADT: `MyString`



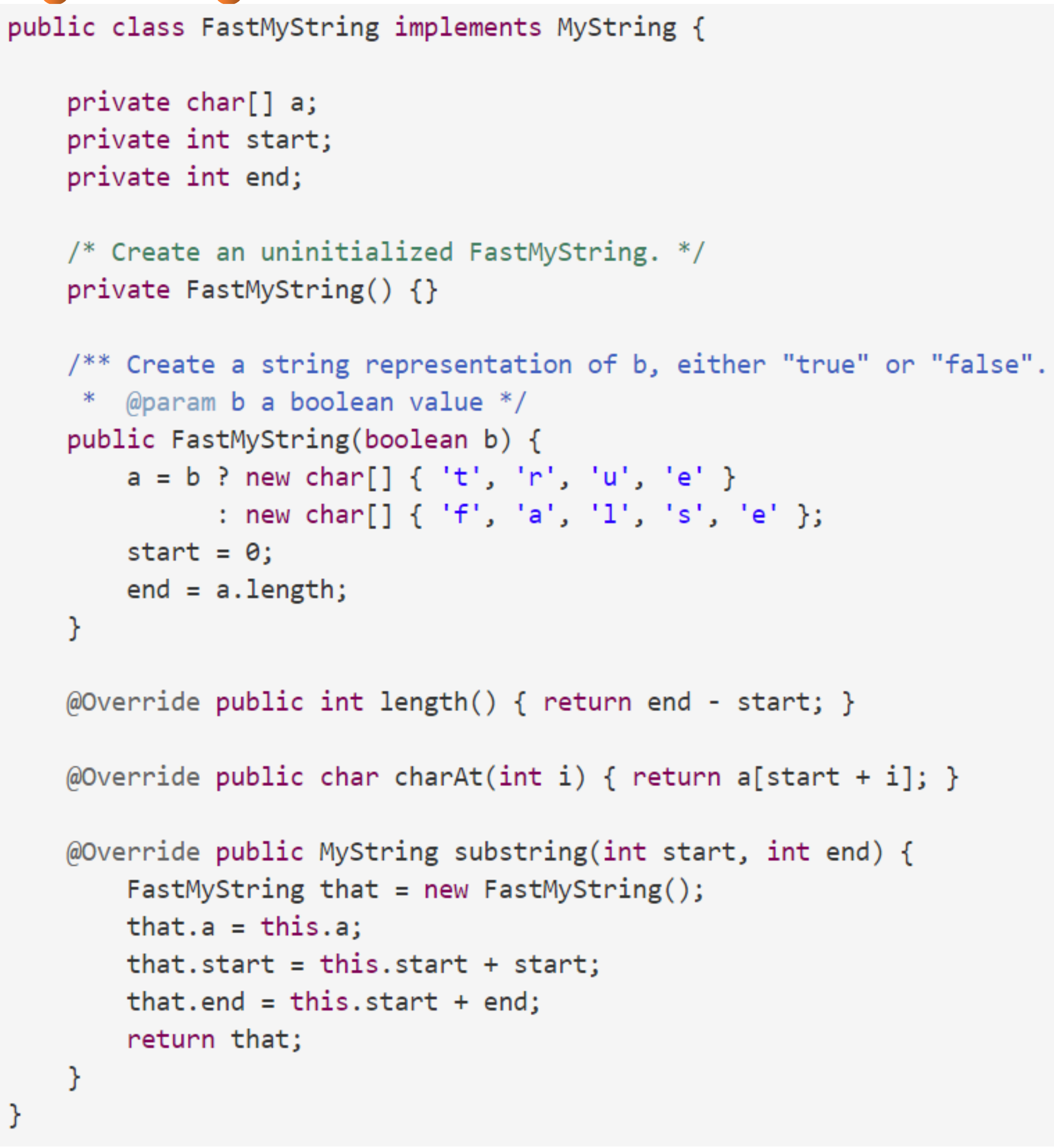
```java
/** MyString represents an immutable sequence of characters. */
public interface MyString {

    // 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`



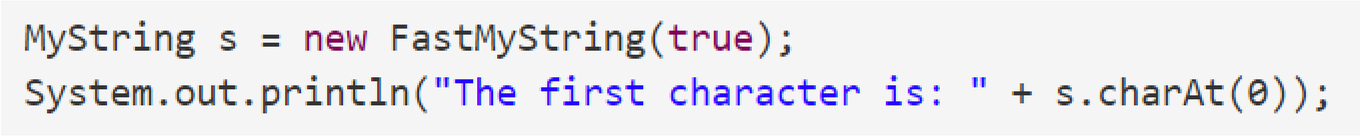
```java
public class SimpleMyString implements MyString {

    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

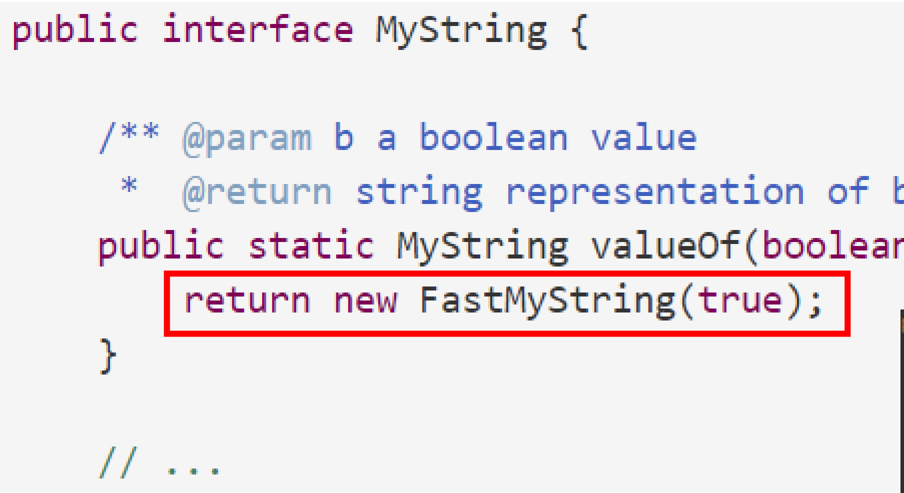


```
MyString s = new FastMyString(true);
System.out.println("The first character is: " + s.charAt(0));
```

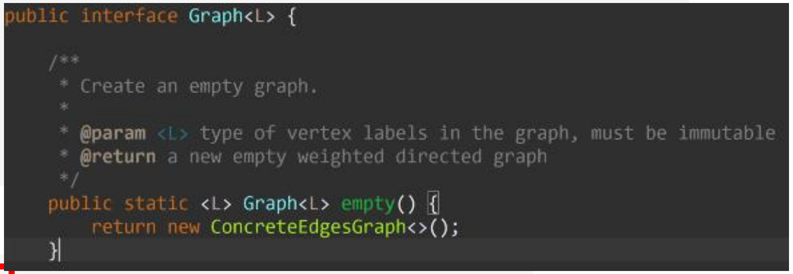
- **Problem: breaks the abstraction barrier**
  - Clients must know the name of the concrete representation class.
  - Because interfaces in Java cannot contain constructors, they must directly call one of the concrete class’ constructors.
  - The spec of that constructor won’t appear anywhere in the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

  - **打破了抽象边界，接口定义中没有包含constructor，也无法保证所有实现类中都包含了同样名字的constructor。**
  - **故而，客户端需要知道该接口的某个具体实现类的名字**

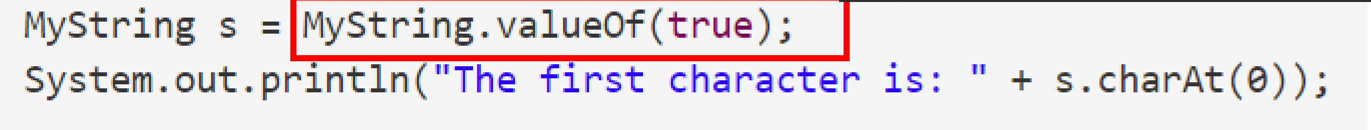
Using static factory instead of constructor



```
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);
    }

    // ...
```



```
public interface Graph<L> {

    /**
     * Create an empty graph.
     *
     * @param <L> type of vertex labels in the graph, must be immutable
     * @return a new empty weighted directed graph
     */
    public static <L> Graph<L> empty() {
        return new ConcreteEdgesGraph<>();
    }
```



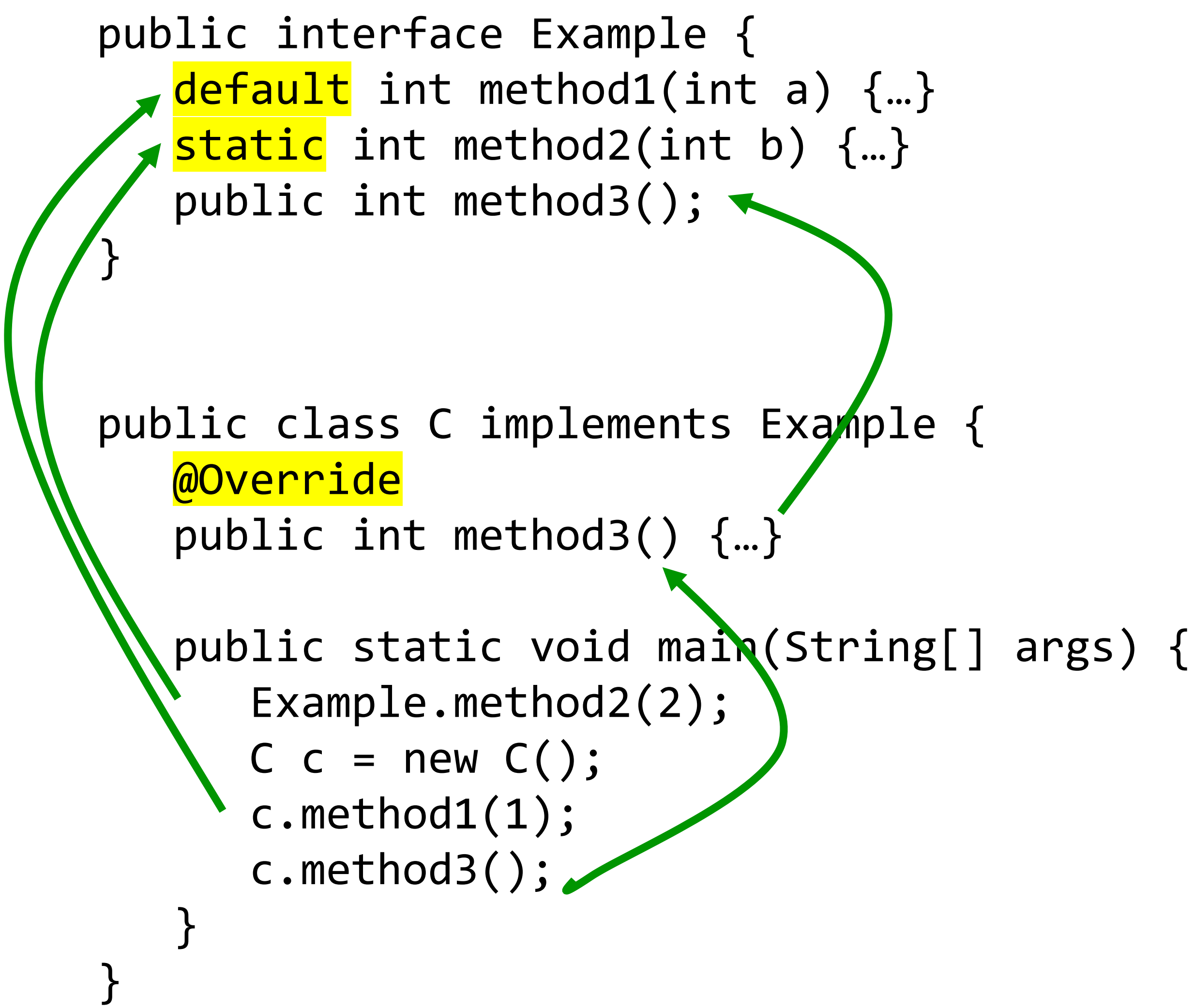
```
MyString s = MyString.valueOf(true);
System.out.println("The first character is: " + s.charAt(0));
```

From Java 8 interfaces are allowed to contain static methods.

Using default methods in an interface

- In a typical design based on abstractions, where an interface has one or multiple implementations, if one or more methods are added to the interface, all the implementations will be forced to implement them too. Otherwise, the design will just break down. **接口中的每个方法在所有类中都要实现，缺点是会导致部分方法的重复实现**
- `default` interface methods are an efficient way to deal with this issue. They allow us to add new methods to an interface that are automatically available in the implementations. Thus, there‘s no need to modify the implementing classes. 通过`default`方法，在接口中统一实现某些功能，无需在各个类中重复实现它。
- The most typical use of default methods in interfaces is to incrementally provide additional functionality to a given type without breaking down the implementing classes. default 方法的典型使用方式：以增量式的为接口增加额外的功能而不破坏已实现的类

Using default methods in an interface



```
public interface Example {
   default int method1(int a) {…}
   static int method2(int b) {…}
   public int method3();
}


public class C implements Example {
   @Override
   public int method3() {…}

   public static void main(String[] args) {
      Example.method2(2);
      C c = new C();
      c.method1(1);
      c.method3();
   }
}
```

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

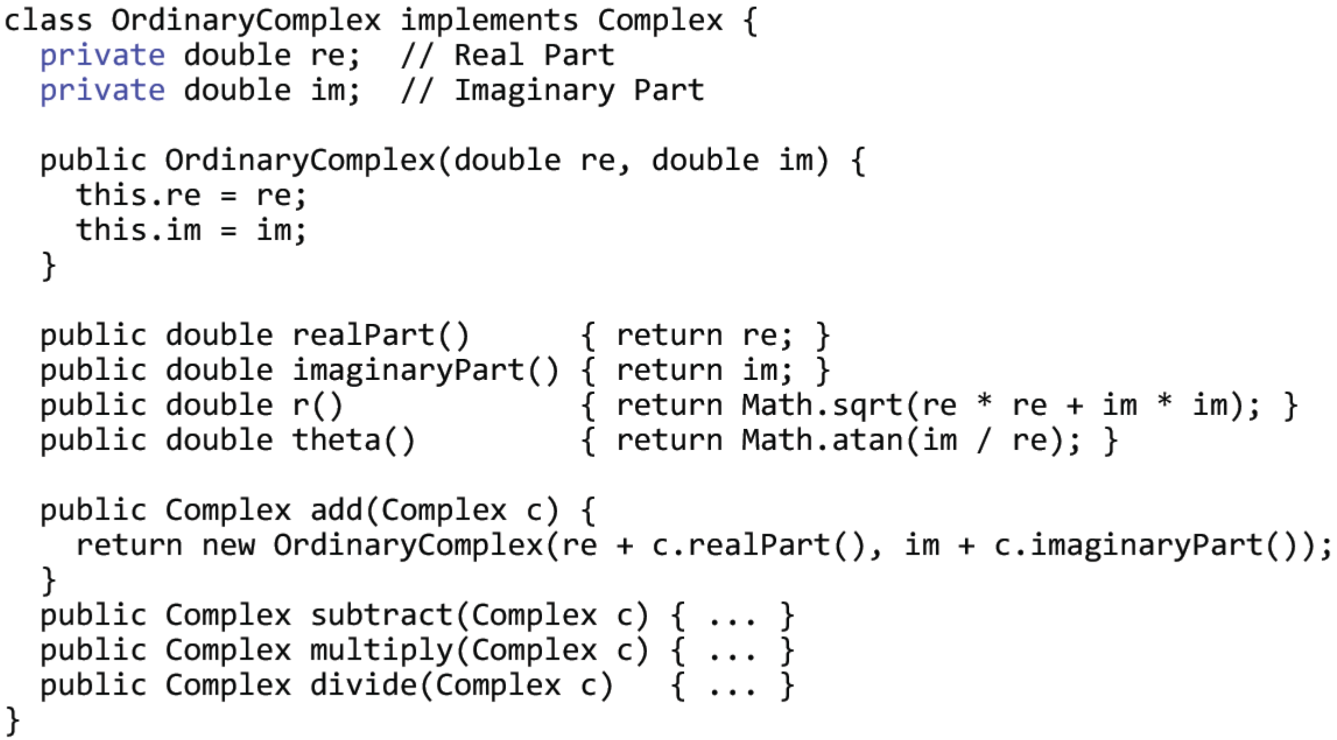
Information hiding with interfaces

- **Declare variables using interface type 使用接口类型声明变量**
- **Client can use only interface methods 客户端仅使用接口中定义的方法**
- **Fields not accessible from client code 客户端代码无法直接访问属性**

- **But this only takes us so far**
  - Client can access non-interface members directly
  - In essence, it's voluntary information hiding
  - 客户端仍然可以访问其他的非接口成员

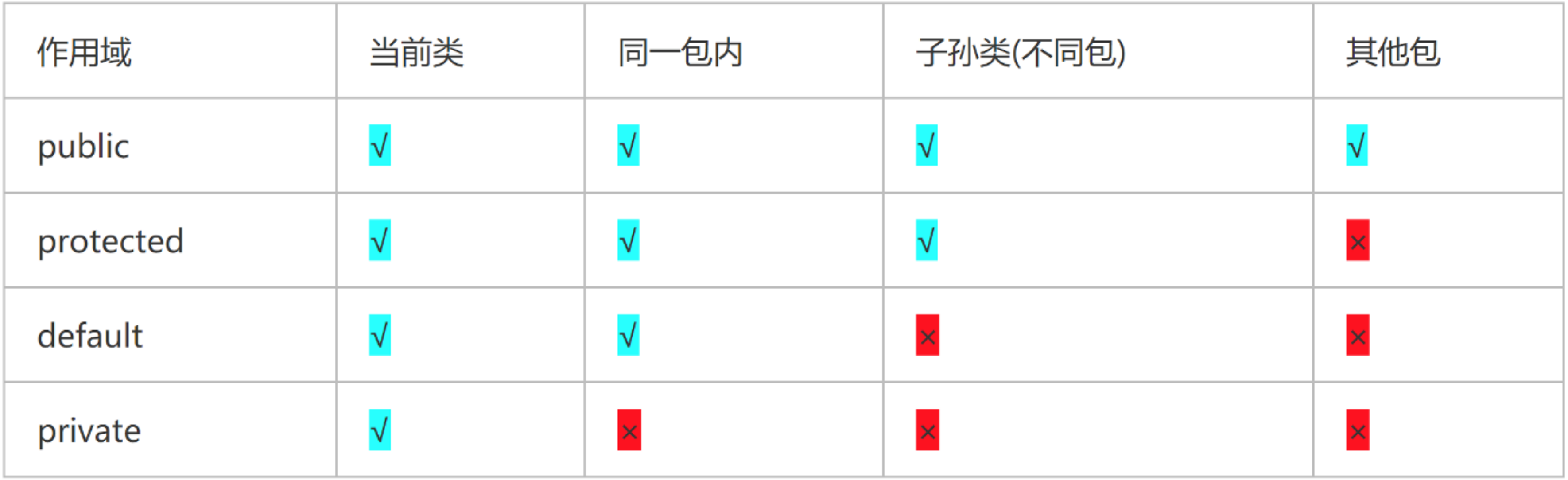
Visibility modifiers for members

- `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)   { ... }
}
```

Visibility modifiers for members



作用域	当前类	同一包内	子孙类(不同包)	其他包
public	√	√	√	√
protected	√	√	√	×
default	√	√	×	×
private	√	×	×	×

Best practices for information hiding

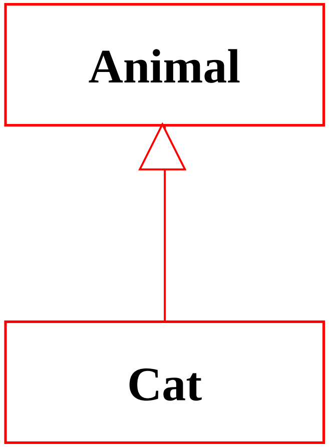
- **Carefully design your API**
- **Provide *only* functionality required by clients, and all other members should be `private`**
- **You can always make a `private` member `public` later without breaking clients**
  - But not vice-versa!

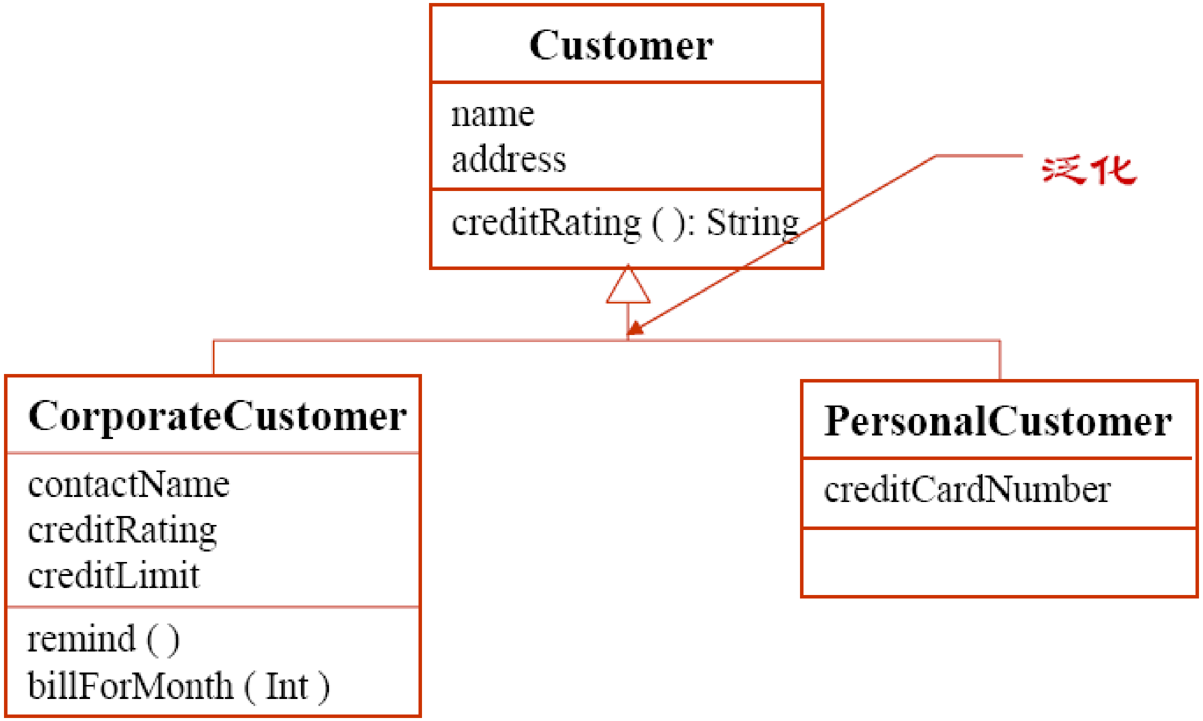
5 Inheritance and Overriding

Inheritance

- **Inheritance is for code reuse**
  - Write code once and only once
  - **Superclass features(public, protected) implicitly available in subclass**

```
Class A extends B
```



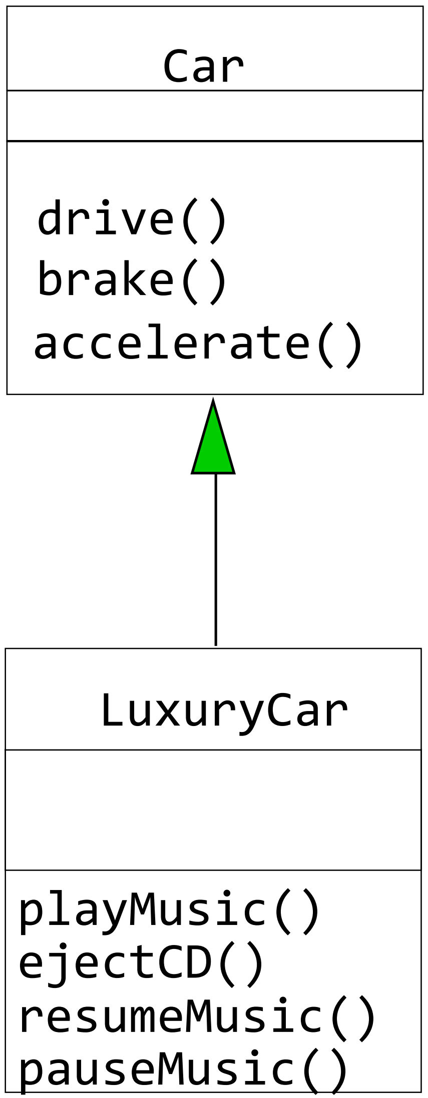


(1) Overriding

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 (严格继承：子类只能添加新方法，无法重写超类中的方法)**
  - 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;
     }
}
```

help() not overwritable

```
class Valve extends Device {
     Position s;
     public void setSerialNr(int n) {
        ...
     }
}
```

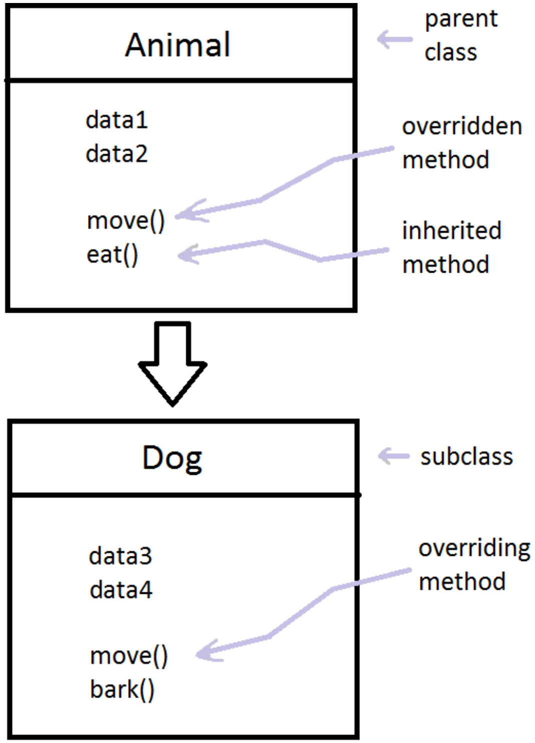
setSerialNr() overwritable

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 { ... }`

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 same name, same parameters or signature, and same return type. 重写的函数：完全同样的signature
  - The version of a method that is executed will be determined by the object that is used to invoke it. 实际执行时调用哪个方法，运行时决定。
  - If an object of a parent class is used to invoke the method, then the version in the parent class will be executed;
  - If an object of the subclass is used to invoke the method, then the version in the child class will be executed.



Example: Overriding a Method

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

父类型中的被重写函数体不为空：意味着对其大多数子类型来说，该方法是可以被直接复用的。

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

对某些子类型来说，有特殊性，故重写父类型中的函数，实现自己的特殊要求

Rewriteable Methods are set to empty

如果父类型中的某个函数实现体为空，意味着其所有子类型都需要这个功能，但各有差异，没有共性，在每个子类中均需要重写。

```
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

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.**

```
class Thought {
    public void message() {
        System.out.println("Thought.");
    }
}

public class Advice extends Thought {
    @Override  // @Override annotation in Java 5 is optional but helpful.
    public void message() {
        System.out.println("Advice.");
        super.message();  // Invoke parent's version of method.
    }
}

Thought parking = new Thought();
parking.message();  // Prints "Thought."

Thought dates = new Advice();
dates.message();  // Prints “Advice. \n Thought."
```

重写之后，利用super()复用了父类型中函数的功能，并对其进行了扩展

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… */ }
```

Constructor call must be the first statement in a constructor

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 override the operations of a superclass with completely new meanings.** 重写的时候，不要改变原方法的本意
- **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!!**

Tips for overriding a method

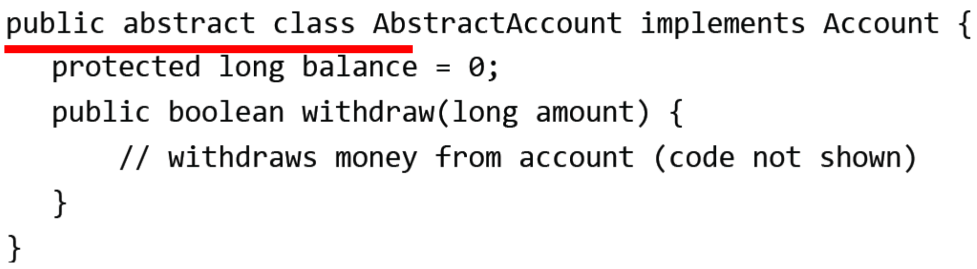
- **If you want to override a method:**
  - Make sure signatures match **签名保持一致**
  - Use `@Override` so compiler has your back**(compiler 会检查覆盖方法和被覆盖的方法签名是否完全一致)**
  - *Do* copy-and-paste declarations (or let IDE do it for you)
  - The visibility can keep the same or increased, but not reduced.

```
class Book{
    protected int getPrice(){
        return 30;
    }
}

public class ComputerBook extends Book{
       int getPrice(){ return 10 }  //error
}
```

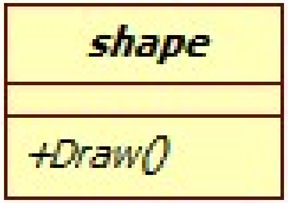
(2) Abstract Class

Abstract Methods and Abstract Classes

- **Abstract method:**
  - A method with a signature but without an implementation (also called abstract operation) **只有定义没有实现**
  - Defined by the keyword `abstract`
- **Abstract class:**
  - A class which contains **at least one** abstract method is called abstract class
  - **It cannot be instantiated! 抽象类不能实例化(不能用new 生成对象)**
  - Before a class derived from an abstract class can be instantiated, all abstract methods of its parent classes must be implemented by some class in the derivation chain. **继承某个抽象类的子类在实例化时，所有父类中的抽象方法必须已经实现**



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



An example of abstract class

```
abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}
```

如果某些操作是所有子类型都共有，但彼此有差别，可以在父类型中设计抽象方法，在各子类型中重写

```
class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

所有子类型完全相同的操作，放在父类型中实现，子类型中无需重写。

有些子类型有而其他子类型无的操作，不要在父类型中定义和实现，而应在特定子类型中实现。

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`

Abstract Methods and Abstract Classes

- **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 多态、子类型、重载

(1) Three Types of Polymorphism

Three Types of Polymorphism (多态)

- **Ad hoc polymorphism (特殊多态):** 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 (功能重载)**. **一个方法可以有多个同名的实现(方法重载)**
- **Parametric polymorphism (参数化多态):** 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(泛型编程)**
- **Subtyping (also called subtype polymorphism or inclusion polymorphism 子类型多态、包含多态):** when a name denotes instances of many different classes related by some common superclass.**一个变量名字可以代表多个类的实例(子类型)**

(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

- **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.**

- 重载：多个方法具有同样的名字，但有不同的参数列表或返回值类型
- 价值：方便**client**调用，**client**可用不同的参数列表，调用同样的函数

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** **静态多态**
  - A function call is resolved using the 'best match technique', i.e. the function is resolved depending upon the argument list. **根据参数列表进行最佳匹配**
  - Static type checking in function calls **静态类型检查**
  - The determination of which of these methods are used is resolved at compile time. **在编译阶段时决定要具体执行哪个方法 (static type checking)**
  - 与之相反，overridden methods则是在run-time进行dynamic checking！

Overloading rules

- **Rules in function overloading: the overloaded function must differ either by the arity or data types**
  - MUST change the argument list. 不同的参数列表
  - CAN change the return type. 相同/不同的返回值类型
  - CAN change the access modifier. 相同/不同的**public/private/protected**
  - CAN declare new or broader checked exceptions. 相同/不同的异常
  - A method can be overloaded in the same class or in a subclass. 可以在同一个类内重载，也可在子类中重载

Legal Overloads

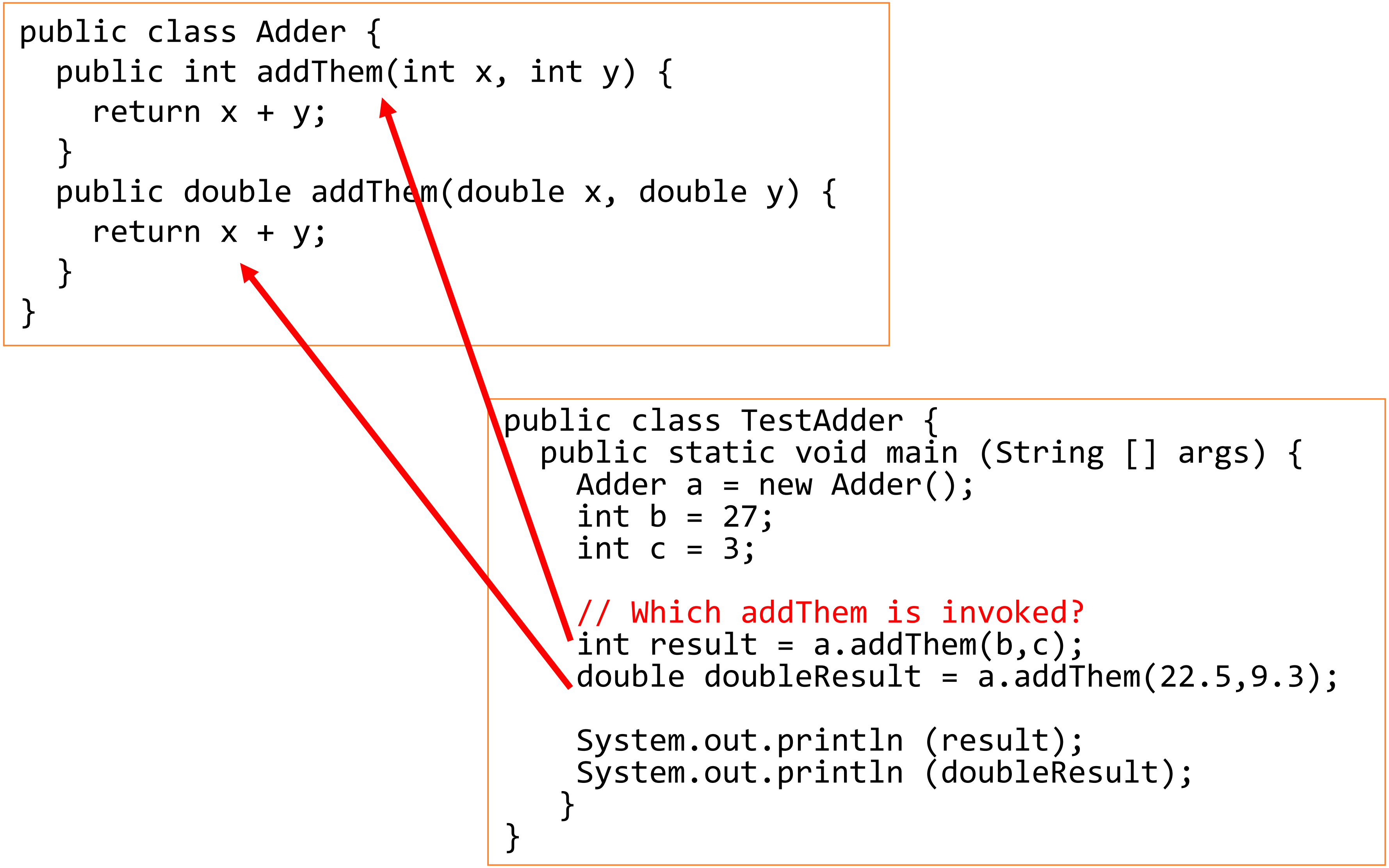
```
public void changeSize(int size,
             String name, float pattern) { }
```

- **Which of 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){ }`
  - `public void changeSize(int length, String pattern, float size){ }`

    The same signature!
  - `public boolean changeSize(int size, String name, float pattern) {  }`

    No changes on parameters

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

Overload也可以发生在父类和子类之间

```
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);
   }
}
```

“Animal”

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	

Animal a = new Animal();
a.eat();	

Horse h = new Horse();
h.eat();	

Animal ah = new Horse();
ah.eat();	

Horse he = new Horse();
he.eat("Apples!");	

Animal a2 = new Animal();
a2.eat("Carrots");	

Animal ah2 = new Horse();
ah2.eat("Carrots");	

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("Carrots");**
	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();     Compile: Cannot Instantiate the
   a. vocalize();                        type Animal
2. Dog d = new Dog();           Woof!
   d.vocalize();
3. Animal b = new Cow();        Moo!
   b.vocalize();
4. b.moo();                     Compile: The method moo() is
                                         undefined for the type
                                         Animal
```

Overriding vs. Overloading

```java
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);
  }
}
```

```java
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 重写时父类和子类中的方法具有相同的签名
  - When a method in a derived class has a different signature from the method in the base class, that is overloading 签名不同时则为重载
  - Note that when the derived class overloads the original method, it still inherits the original method from the base class as well 子类重载了父类的方法后，子类仍然继承了被重载的方法

Overriding vs. Overloading

详见第9章协变

	Overloading	Overriding
Argument list	**Must change**	**Must not change**
Return type	**Can Change**	**Not change or subtype**
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.** **参数多态性是指方法针对多种类型时具有同样的行为（这里的多种类型应具有通用结构），此时可使用统一的类型变量表达多种类型**
  - 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. **在运行时根据具体指定类型确定具体类型(编译成class文件时，会用指定类型替换类型变量"擦除")**
  - This is what is called "**Generics (泛型)**" in Java.

Parametric polymorphism

- **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. 泛型编程是一种编程风格，其中数据类型和函数是根据待指定的类型编写的，随后在需要时根据参数提供的特定类型进行实例化。

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.***

泛型编程围绕“从具体进行抽象”的思想，将采用不同数据表示的算法进行抽象，得到泛型化的算法，可以得到复用性、通用性更强的软件。

Generics in Java 类型变量

- A **type variable(类型变量)** is an unqualified(无限制的) identifier.
- **Using <>, the diamond operator, to help declare type variables.**
- **For example:**
  - `public interface List<E>`
  - `public class Entry<KeyType, ValueType>`
  - `List<Integer> ints = new ArrayList<Integer>();`
- **They are introduced by generic class declarations, generic interface declarations, generic method declarations** 使用泛型变量的三种形式：泛型类、泛型接口和泛型方法

Generics in Java 泛型类

- A **class** is generic if it declares one or more type variables. 类中如果声明了一个或多个泛型变量，则为泛型类
  - These type variables are known as the **type parameters(类型参数)** of the class. 这些类型变量称为类的类型参数
  - 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.

```
class ClassName <T,E,…>{
  private T var;
  .....
}
```

Example

```
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();
```

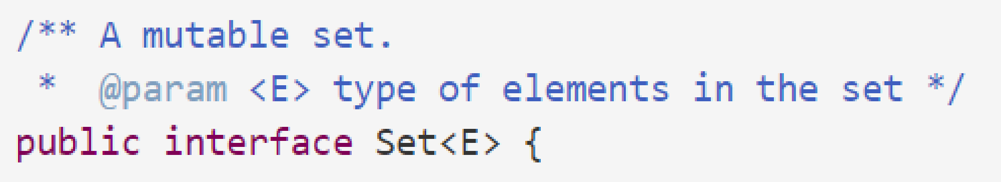
Generics in Java –泛型接口

- 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.

```
interface InterfaceName <T,E,…>{
  public T methodName();
  .....
}
```

Another example: Java Set

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

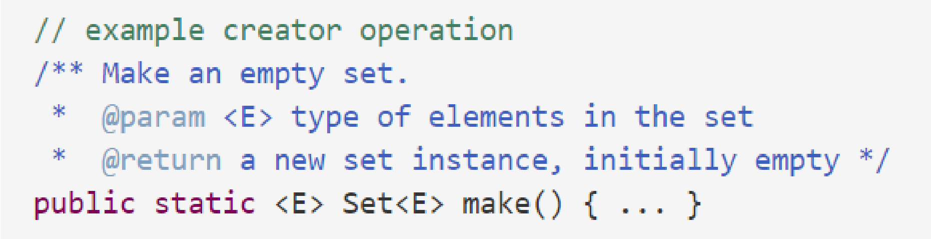


```
/** A mutable set.
 *  @param <E> type of elements in the set */
public interface Set<E> {
```

- **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> .**

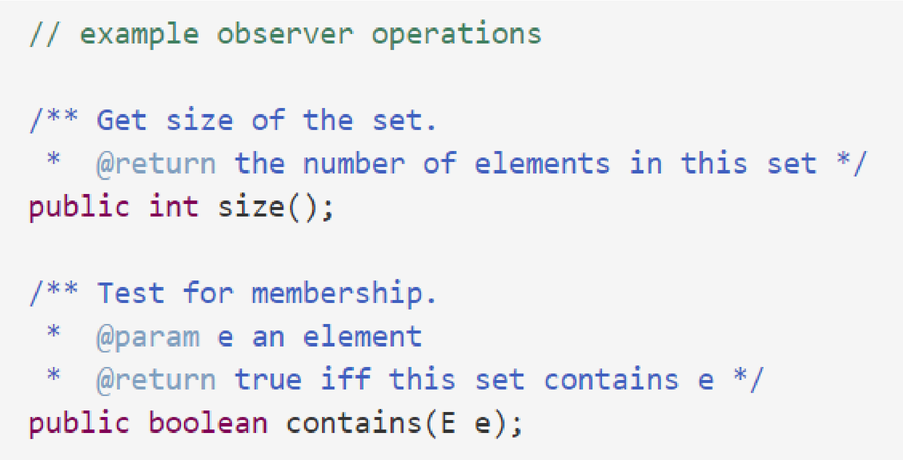
Another example: Java Set

- **Creator**



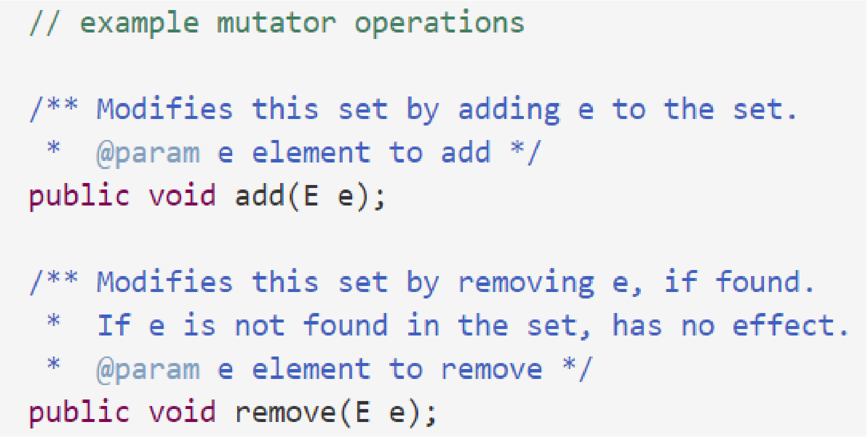
```
// example creator operation
/** Make an empty set.
 *  @param <E> type of elements in the set
 *  @return a new set instance, initially empty */
public static <E> Set<E> make() { ... }
```

- **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);
```

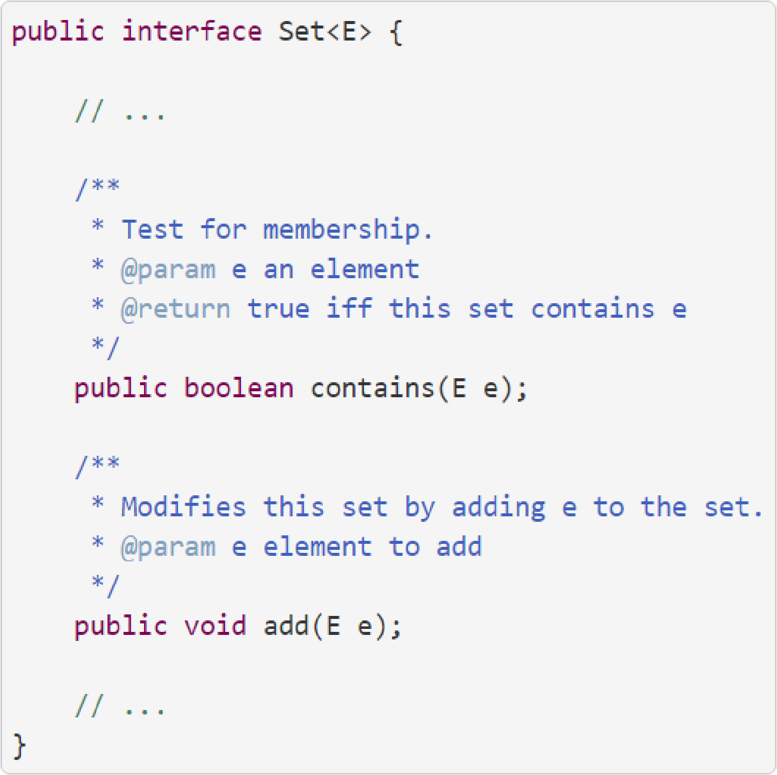
- **Mutator**



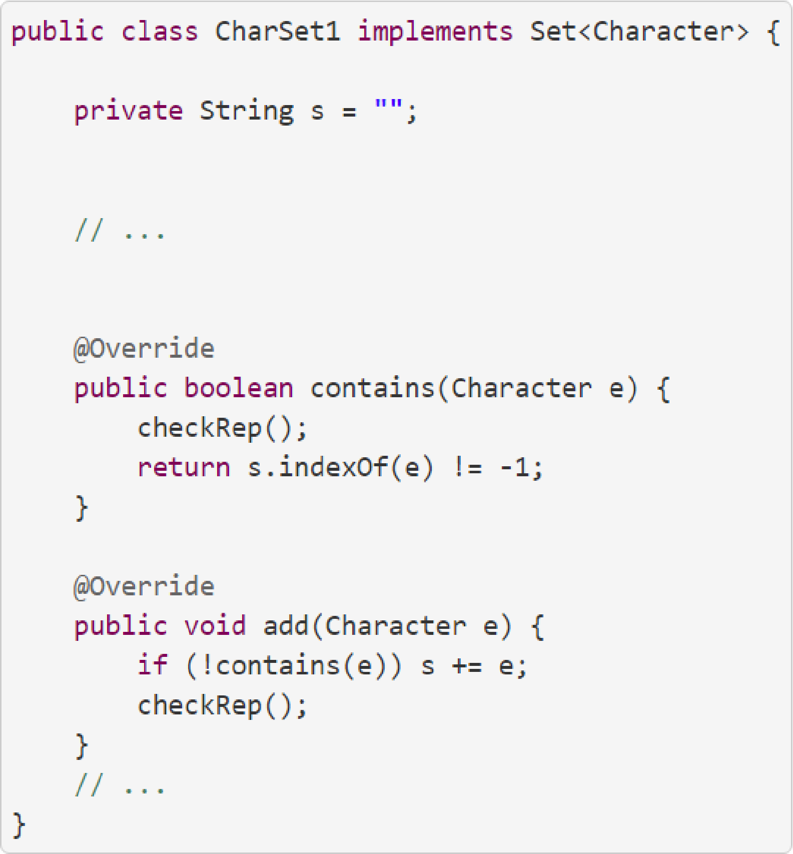
```
// example mutator operations

/** 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 –泛型接口

- **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`. 泛型接口，非泛型的实现类



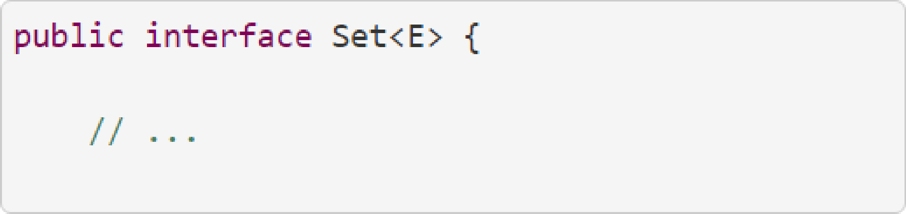
```
public interface Set<E> {

    // ...

    /**
     * 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);

    // ...
}
```



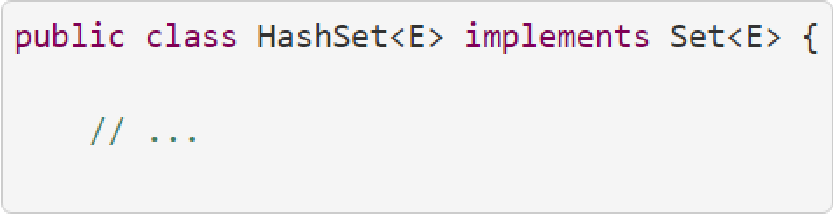
```
public class CharSet1 implements Set<Character> {

    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.** 泛型接口，泛型的实现类
  - 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> {

    // ...
```

Generics in Java-泛型方法(选学)

- 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.

```
public <T> T genericMethod(Class<T> tClass) {
        T instance = tClass.newInstance();
        return instance;
}
```

注：

泛型类/接口，是在实例化类的时候指明泛型的具体类型
泛型方法，是在调用方法的时候指明泛型的具体类型

Generics in Java泛型方法(选学)

- **Generic method in normal class** 普通类中的泛型方法

```
public class GenericTest {
   public void normalMethod(…){ …}

   public <T> T genericMethod(…){
        T var;
         ….;
    }
}
```

Generics in Java 泛型方法(选学)

- **Generic method in generic class 泛型类中的泛型方法**

```
class GenericTest<T>{
        //下面的T同所在类的类型变量一致，show1不是泛型方法
        public void show1(T t){
             System.out.println(t.toString());
         }

        //下面的E是新的类型变量，只适用于此方法，show2是泛型方法
        public <E> void show2(E t){
             System.out.println(t.toString());
        }

        //下面的T是新的类型变量，同类的类型变量无关（即使名字一样）
        //show3是泛型方法
        public <T> void show3(T t){
             System.out.println(t.toString());
        }
}
```

Generics in Java泛型方法(选学)

- **Generic method in generic class** 泛型类中的泛型方法

```
public static void main(String[] args){

      GenericTest<String> genericTest = new GenericTest<>();

      genericTest.show1("genericTest!"); //succeed, "genericTest!"
      genericTest.show1(Integer.valueOf("1")); //compile error

      genericTest.show2(Integer.valueOf("1")); //succeed, 1
      genericTest.show2(person);  //succeed, maybe name of person

       genericTest.show3(Integer.valueOf("1")); //succeed, 1
      genericTest.show3(person);  //succeed, maybe name of person
}
```

Generics in Java泛型方法(选学)

- **Generic method and static method**
  - Static method cannot use generic variable defined in class level. 静态方法不能使用所在泛型类定义的泛型变量
  - If static methods are to use generics, they must also be defined as generic methods. 如果静态方法要使用泛型的话，必须将静态方法也定义成泛型方法。

```
/**
* Create an empty graph.
*
* @param <L> type of vertex labels in the graph, must be immutable
* @return a new empty weighted directed graph
*/
public static <L> Graph<L> empty() {
  ……
}
```

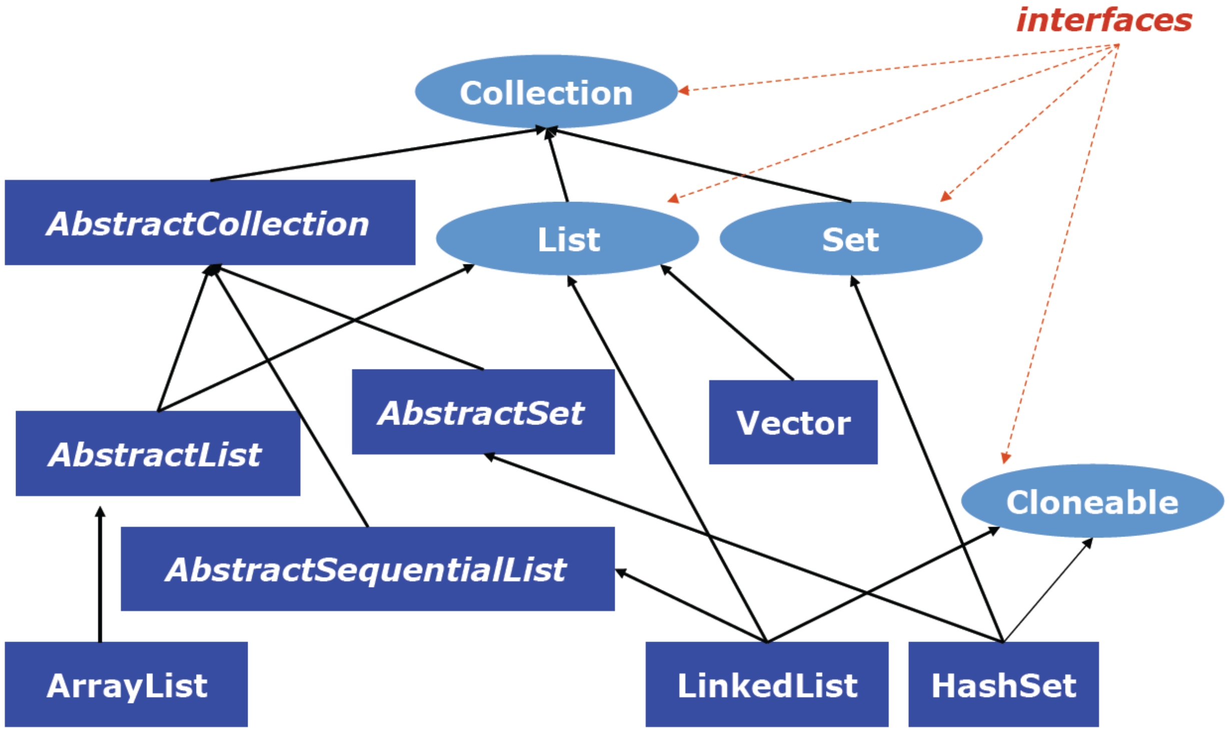
Some Java Generics details

- **Can have multiple type parameters**
  - e.g., `Map<E, F>`, `Map<String, Integer>`
- **Wildcards 通配符**
  - `List<?> list = new ArrayList<String>();`
  - `List<? extends Animal>`
  - `List<? super Animal>`
- **Generic type info is erased (i.e. compile-time only)**
  - Cannot use `instanceof` to check generic type 运行时泛型消失了！
- **Cannot create Generic arrays**
  - `Pair<String>[] foo = new Pair<String>[42];` // won't compile

(4) Subtyping Polymorphism

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. 一个类只有一个父类，但可以实现多个接口

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.**
- 子类型的规约不能弱化超类型的规约。

Subtype polymorphism

- **Subtype polymorphism: Different kinds of objects can be treated uniformly by client code**

子类型多态：不同类型的对象可以统一的处理而无需区分

- **Each object behaves according to its type (e.g., if you add new kind of account, client code does not change)** 从而隔离了“变化”

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

- **Liskov Substitution Principle (LSP):**
  - If S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e. an object of type T may be substituted with any object of a subtype S) without altering any of the desirable properties of T.

➔ Section 9 Reusability

10 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 in 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

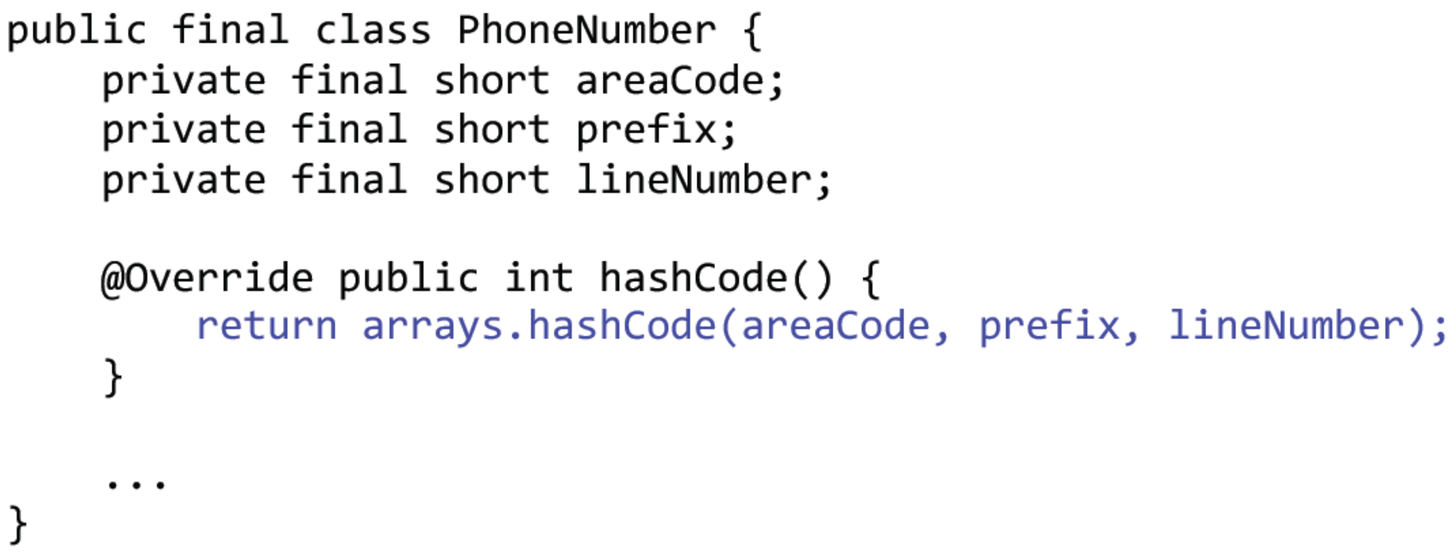
```
public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    @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?

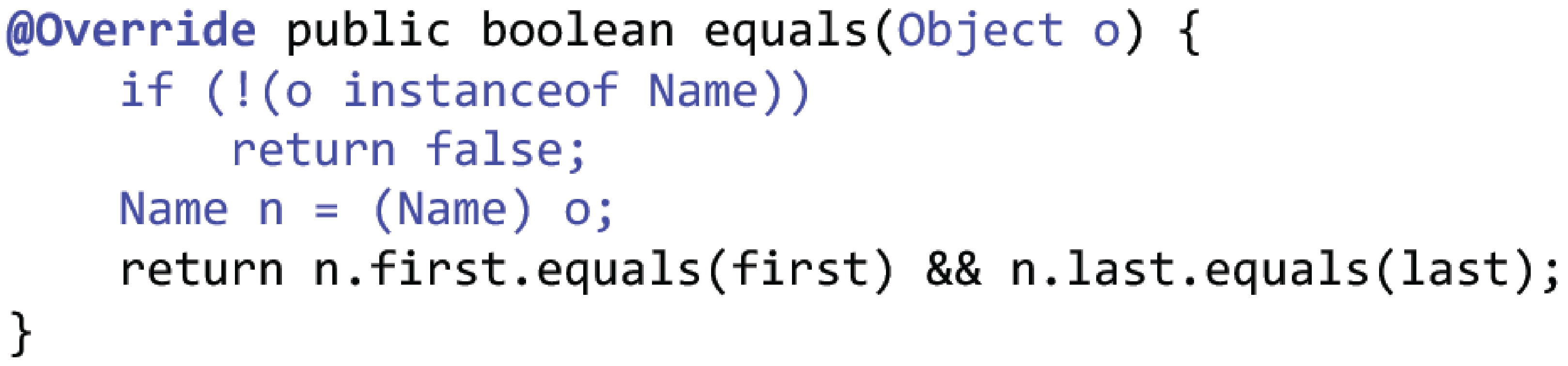
```
public class Name {
    private final String first, last;
    public Name(String first, String last) {
        if (first == null || last == null)
            throw new NullPointerException();
        this.first = first; this.last = last;
    }
    public boolean equals(Name o) {
        return first.equals(o.first) && 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);
}
```

11 Designing good classes

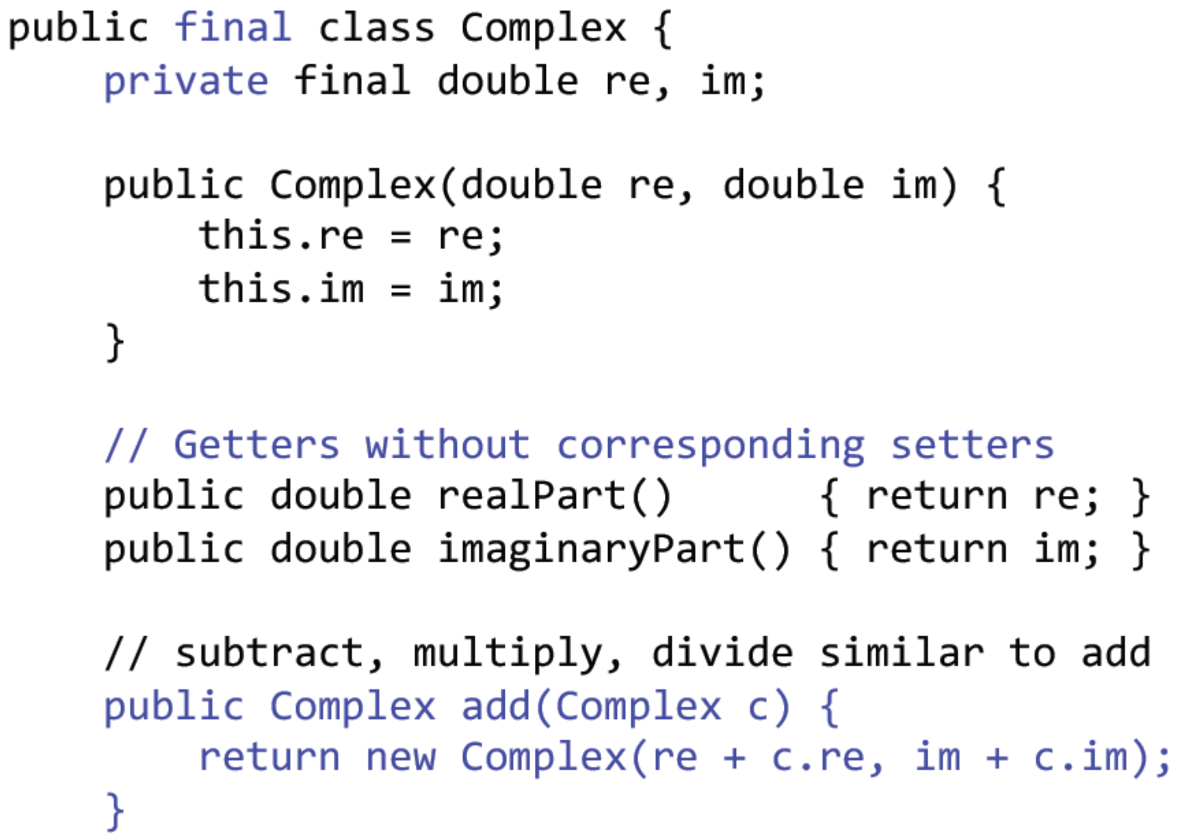
Advantages of immutable classes

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

How to write an immutable class

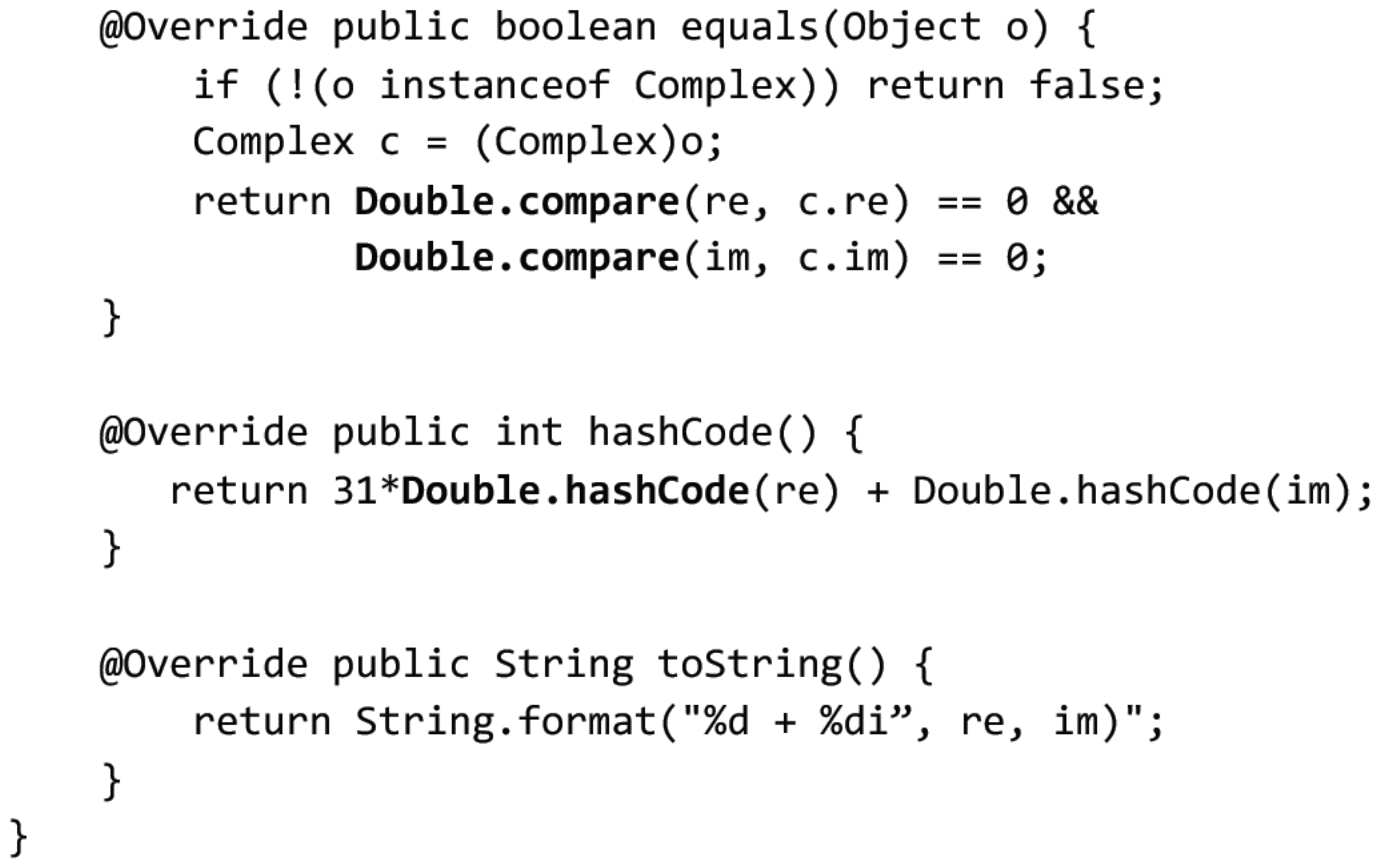
- Don’t provide any mutators
- Ensure that no methods may be overridden
- Make all fields `final`
- Make all fields `private`
- Ensure security of any mutable components (avoid rep exposure)
- **Implement** `toString()`, `hashCode()`, `clone()`, `equals()`, **etc.**

Immutable class example



```
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)";
    }
}
```

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

12 History of OOP

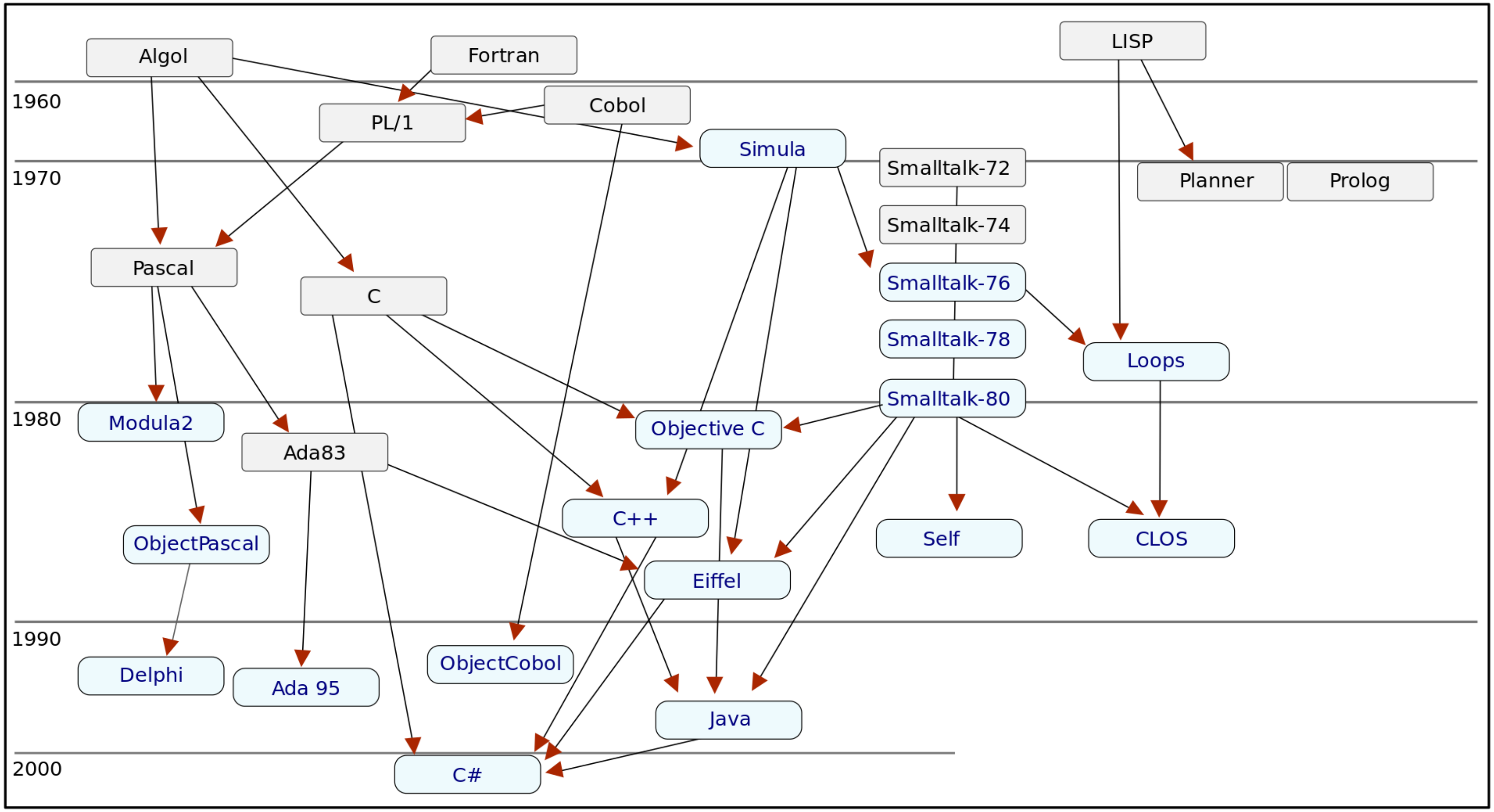
Simulation and the origins of OO programming

- **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

- **Criteria of Object-Orientation**
- **Basic concepts: object, class, attribute, method, and interface**
- **Distinct features of OOP**

  Encapsulation and information hiding

  Inheritance and overriding

  Polymorphism, subtyping and overloading

  Static and Dynamic dispatch
- **Some important `Object` methods in Java**
- **To write an immutable class**
- **History of OOP**
- **Summary**

The end

May 10, 2022