320341 Programming in Java

JACOBS
UNIVERSITY

Fall Semester 2014

Lecture 8: Interfaces and Inner Classes

Instructor: Jürgen Schönwälder

Slides: Bendick Mahleko

Objectives

The objective of this lecture is to

- Introduce interfaces and their use in Java

Overview

JACOBS UNIVERSITY

An interface can be characterized as follows:

- Specifies ***what a class can do, without specifying how***
- An interface says,
  "this is what all classes implementing this interface will look like"
- An interface is used to establish a ***protocol*** between classes

An interface gives the form of a class

- ❑ Method names
- ❑ Argument lists
- ❑ Return types

- An interface provides no method bodies

Overview

JACOBS UNIVERSITY

What interfaces do NOT provide?

- Interfaces never have instance fields
- Methods are never implemented in interfaces

Instance fields: supplied by the implementing class(es)

Method implementation: implementations provided the implementing class (es)

Interface Creation

The Java keyword interface is used to create an interface

interface fields are implicitly **static** & **final**

```
interface Instrument {
    // compile-time constants
    int i = 5;  // static and
    final                                   // no
    method definitions allowed

    void play();   // methods are automatically public
    String what();
    void adjust();
}
```

i) Methods are implicitly **public**
ii) Make interface methods **public** in implementing class

- Add public keyword before interface keyword to define an interface in a file of the same name
- An **Implementing** class defines how the interface works

Interface Creation

JACOBS UNIVERSITY

Two steps to implement an interface

1. Declare that a class intends to implement an interface
   - *Use the implements keyword*

2. Supply definitions for ALL methods in the interface
   - Implement each method in the interface
   - In one or more methods is left out, an error message will be generated

Interface Creation

The Java keyword interface is used to create an interface

interface fields are implicitly **static** & **final**

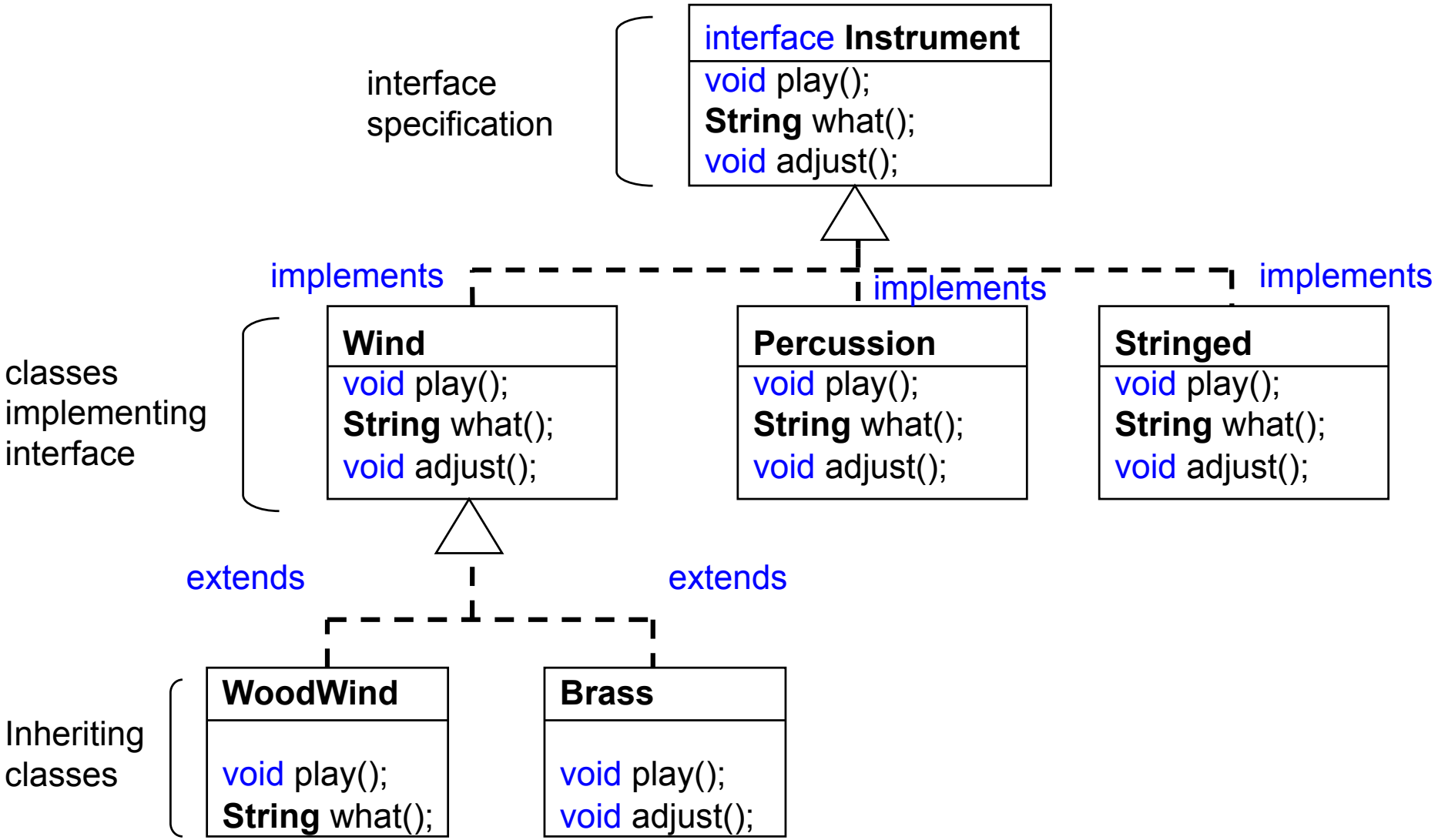
```
interface Instrument {
    // compile-time constants
    int i = 5;  // static and
    final                                   // no
    method definitions allowed

    void play();   // methods are automatically public
    String what();
    void adjust();
}
```

i) Methods are implicitly **public**
ii) Make interface methods **public** in implementing class

- Add public keyword before interface keyword to define an interface in a file of the same name
- An **Implementing** class defines how the interface works

Example



Implementing interfaces

JACOBS UNIVERSITY

Use keyword implements

How to declare a class that implements an interface

Implementing class **must implement all interface methods**!

Class implementing an interface is like any other class e.g., can be extended

```
class Wind implements Instrument   {
        public void play() { System.out.println(„Wind.play()“); }
        public String what() {  return „Wind“;   }
        public void adjust()  {  }
}
        …

class Brass extends Wind {
        public void play() { System.out.println(„Brass.play()“); }
        public void adjust(){ System.out.println(„Brass.adjust()“); }
}
```

Properties of interfaces

JACOBS UNIVERSITY

- Interfaces are not classes
- Interfaces cannot be instantiated

```
Instrument x = new Instrument();   // ERROR
```

- It is legal to declare interface variables

```
Instrument inst; // OK
```

- *An interface object must reference an object of a class that implements it*

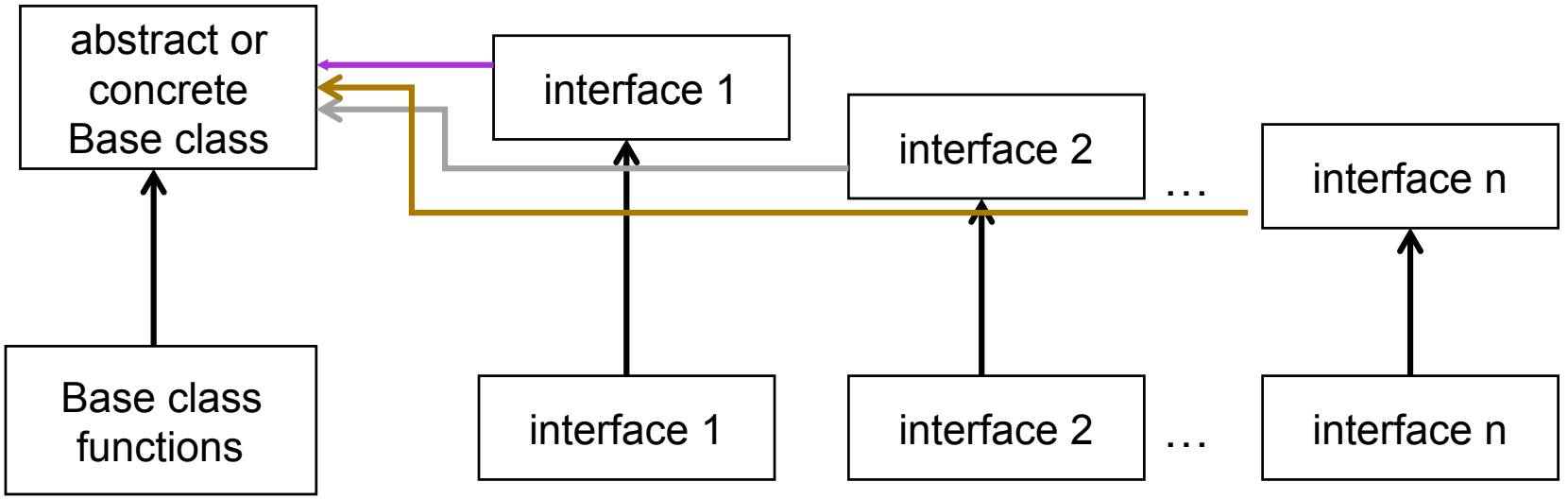
```
Instrument inst = new Wind(…); // OK provided Wind
                                                  // implements
                                                  Instrument
```

- Use instanceof to check if the object implements an interface

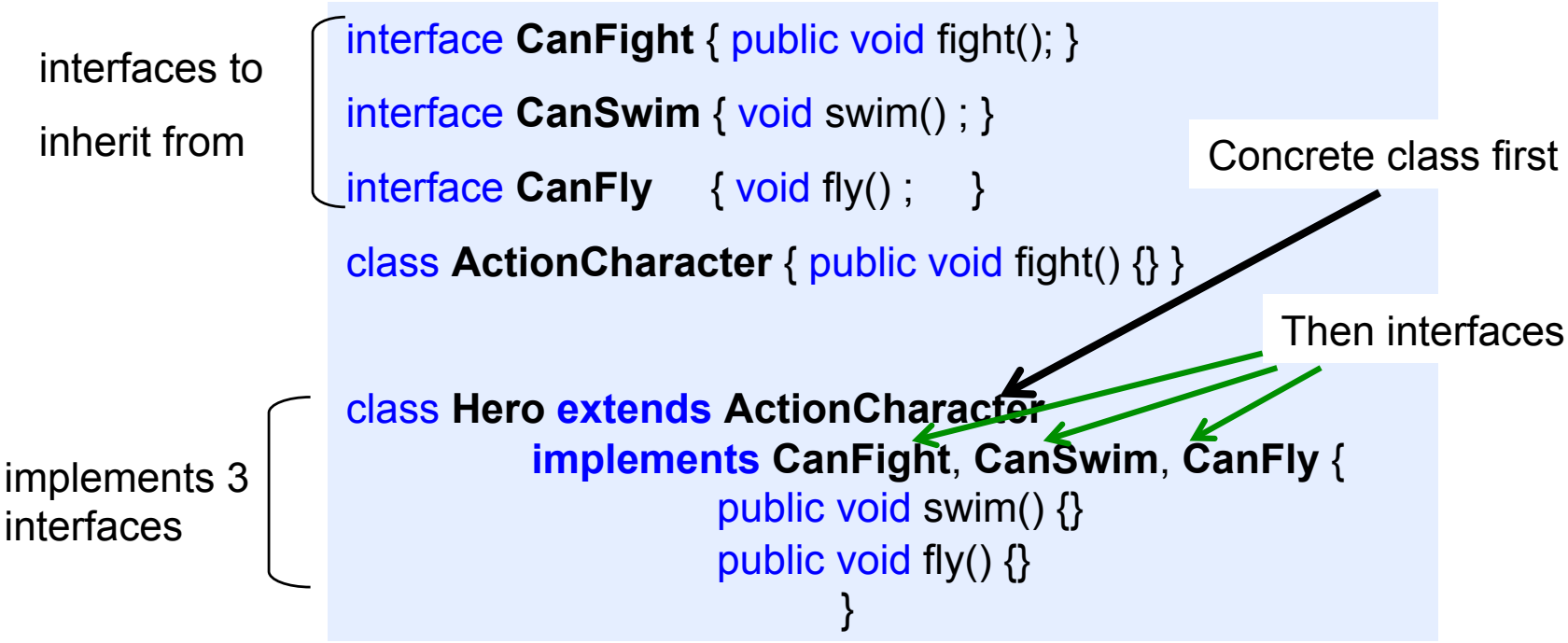
```
if (anObject instanceof Instrument) {…}
```

“Multiple Inheritance” in Java

The connotation “an **x** is an **a** and a **b** and a **c**” is called *multiple inheritance* in C++



Example



- All method definitions must be given by implementing classes

Example

Take different interface references as parameters

Hero object is **upcast to different base types**

```
public class Adventure {
        static void t (CanFight x)    { x.fight();  }
        static void u (CanSwim x)     { x.swim(); }
        static void v (CanFly x)      { x.fly();  }
        static void w (ActionCharacter x) { x.fight(); }

        public static void main(String [] args) {
                Hero h = new Hero();
                t(h); // treat as CanFight
                u(h); // treat as CanSwim
                v(h); // treat as CanFly
                w(h); // treat as ActionCharacter
                }
}
```

- **Hero** object has been upcast to each of the three interfaces
- Therefore interfaces are more than just “pure abstract classes”

Name Collision

Interfaces with same method names

```
interface I1 { void f(); }

interface I2 { int f(int i) ; }

interface I3 { int f() ;     }

class C { public int f() {return 1;} }

class C2 implements I1, I2  {
            public void f() {}
            public int f (int i)  { return 1;} //
overloaded                          }
                                       class C3 extends C
implements I2  {                                        public
int f(int i) { return 1; } // overloaded                  }
```

Methods differ only by return type

```
class C5 extends C implements I1  {}
interface I4 extends I1, I3 {}
```

- Mixing-up overriding, implementation and overloading
- **Avoid using the same method names in different interfaces that are intended to be combined!!**

Interface Inheritance

Interfaces can extend other interfaces (interface inheritance)

An interface can extend other interfaces

New interface with more methods created

```
interface Monster {
        void menace();
}

interface DangerousMonster extends Monster  {
        void destroy() ;
}

interface Lethal {
        void kill() ;
}

interface Vampire extends DangerousMonster, Lethal {
        void drinkBlood();
}

class DragonZilla implements DangerousMonster {
        public void destroy ()  { }
        public void menace() { }
}
```

Only possible with interfaces

Grouping Constants

JACOBS UNIVERSITY

The interface is a convenient tool for creating a group of constants

- *Any fields put in an interface are automatically static and final*
- *The fields in an interface are automatically public*

Group of constants – automatically public, static, final

```
public interface Months {
  int
    JANUARY = 1, FEBRUARY = 2, MARCH = 3,
    APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
    AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
    NOVEMBER = 11, DECEMBER = 12;
}
```

The values are referenced for example by **Months.JANUARY** after importing **Month** into you class

Initializing Fields in Interfaces

JACOBS UNIVERSITY

Fields defined in interfaces are automatically static and final

- The fields cannot be assigned “blank finals”
- The fields must be assigned initial values (can be nonconstant expr.)
- Static fields are initialized when the class is first loaded

- *The fields are not part of the interface, but are stored in static storage area for that interface*

Initializing Fields in Interfaces

JACOBS UNIVERSITY

interface fields: automatically
public, static & final
must be assigned initial values

Initial values can be nonconstant expressions

```
public interface RandVals {
 int rint = (int)(Math.random() * 10);
            long rlong = (long)(Math.random() * 10);
            float rfloat = (float)(Math.random() * 10);
            double rdouble = Math.random() * 10;
}
```

Interfaces and Abstract Classes

JACOBS UNIVERSITY

Similarities between abstract classes and interfaces

- Both used to specify services and defer their implementation to implementing classes
- Both abstract classes and interfaces cannot be instantiated to create objects

Differences between abstract classes and interfaces

- Abstract classes can contain non-abstract methods but all methods in interfaces have no definition/ implementation

- A class can inherit from only one abstract superclass, but a class can implement any number of interfaces – multiple inheritance of interfaces in Java is legal

Inner Classes

JACOBS UNIVERSITY

What are inner classes?

- An inner class is a *class defined inside another class*

- Inner classes allow to group classes that logically belong together
  - ❑ This gives you better control of visibility

- Inner classes are distinct from *composition*

Inner Classes

JACOBS UNIVERSITY

Benefits of inner classes

1. Access data in the scope where it is defined (including private data)
2. Inner classes can be hidden from other classes in the same package
3. Anonymous inner classes are handy for defining call-backs

Example

Create inner class by placing the class definition inside a surrounding class

Inner class **Contents**

Inner class **Destination**

Inner classes used inside method

```
public class Parcel1 {
        class Contents {
            private int i = 11;
            public int value() { return i; }
        }
        class Destination {
                private String label;
                Destination(String whereTo) {
                        label = whereTo;
                }
                String readLabel() { return label; }
        }
        // Using inner classes looks just like
        // using any other class, within Parcel1:
        public void ship(String dest) {
                Contents c = new Contents();
                Destination d = new Destination(dest);
                System.out.println(d.readLabel());
        }
  …
}
```

Example

Returning reference to inner class

```
public class Parcel2 {
        class Contents {
                private int i = 11;
                public int value() { return i; }
        }
        class Destination {
                private String label;
                Destination(String whereTo) {
                        label = whereTo;
                }
                String readLabel() { return label; }
        }
        public Destination to(String s) {
                return new Destination(s);
        }
        public Contents cont() {
                return new Contents();
        }
```

Outer class's method returning **Destination** reference

Outer class's method returning **Contents** reference

Referencing Inner Classes

- The type of object is specified as **OuterClassName.InnerClassName**
- Example

Referencing inner class objects

```
public class Parcel2 {
        …
        public static void main(String[] args) {
          Parcel2 p = new Parcel2();
          p.ship("Tanzania");
          Parcel2 q = new Parcel2();

          // Defining references to inner classes:
          Parcel2.Contents c = q.cont();
          Parcel2.Destination d = q.to("Borneo");
        }
      }
```

Inner Classes and Upcasting

JACOBS UNIVERSITY

Hiding the interface implementation

- Define common interfaces in their own files

Destination & **Contents** represent interfaces available to the client programmer

```
// Destination.java file
public interface Destination {
        String readLabel();
}

// Contents.java file
 public interface Contents {
        int value();
}
```

Inner Classes and Upcasting

JACOBS UNIVERSITY

Hiding the interface implementation cont…

PContents is private, only **Parcel3** can access it

PDestination is protected, so only **Parcel3**, classes in **Parcel3** package and inheritors of **Parcel3** can access it

Client programmer has restricted access to class members

```
public class Parcel3 {
    private class PContents implements Contents {
      private int i = 11;
      public int value() { return i; }
    }
    protected class PDestination implements Destination {
        private String label;
        private PDestination(String whereTo) {
              label = whereTo;
        }
        public String readLabel() { return label; }
    }
    public Destination dest(String s) {
      return new PDestination(s);
    }
    public Contents cont() {
      return new PContents();
    }
}
```

Inner Classes and Upcasting

JACOBS UNIVERSITY

Can't downcast to private (protected) inner class, unless you're an inheritor

```
class Test {
    public static void main (String[] args) {
      Parcel3 p = new Parcel3();
      Contents c = p.cont();
      Destination d = p.dest("Tanzania");
      // Illegal -- can't access private class:
      //! Parcel3.PContents pc = p.new PContents();
    }
}
```

private inner classes allow the class designer

- To completely prevent type-coding dependencies
- To completely hide details about implementation

Note: Normal (non-inner) classes cannot be made private or protected — only public or "friendly."

Inner Classes in Methods and Scopes

JACOBS UNIVERSITY

Inner classes can be created

- Inside methods
- In arbitrary scope

Why nest classes inside methods?

- When implementing an interface to create and return a reference
- When solving a complicated problem and want to aid your solution

Inner Classes in Methods and Scopes

*A class that is nested within a method or block is called a **local inner class***

No access specifier

PDestination:

- is part of *dest* method
- is not accessible outside *dest*
- is a valid object after *dest* returns

```
public class Parcel4 {
      public Destination dest(String s)  {
        class PDestination implements Destination {
            private String label;
            private PDestination(String whereTo) {
            label = whereTo;
            }
            public String readLabel() { return label; }
        } // end of PDestination
        return new PDestination(s);
      } // end of dest
      public static void main(String[] args) {
            Parcel4 p = new Parcel4();
            Destination d = p.dest("Tanzania");
      }
}
```

Inner Classes in Methods and Scopes

JACOBS UNIVERSITY

Nesting a class within a Scope

No access specifier

TrackingSlip :

- Gets compiled along with other classes
- Is not available outside defined scope

```
public class Parcel5 {
  private void internalTracking(boolean b) {
    if (b) {
        class TrackingSlip {
            private String id;
            TrackingSlip(String s) {
                id = s;
            }
            String getSlip() { return id; }
        }
        TrackingSlip ts = new TrackingSlip("slip");
        String s = ts.getSlip();
    } //end if
    // Can't use it here! Out of scope:
    //! TrackingSlip ts = new TrackingSlip("x");
  }
  public void track() { internalTracking(true); }
  ….
}
```

Overview

Local inner classes are never declared with an access specifier

Local inner classes' scope is restricted to the block in which they are declared

Local inner classes are completely hidden

Anonymous Inner Classes

JACOBS UNIVERSITY

We can go a step further than local inner classes

- To make a **single object of the local inner class**, we don't need to give the class a name

- Such a class is called an ***anonymous inner class***

- An anonymous inner class cannot have constructors
  - ❑ Construction parameters are given to the superclass constructor

Example

JACOBS UNIVERSITY

Creates an object of an anonymous class that's inherited from **Contents**

```
public class Parcel6 {
    public Contents cont() {
            return new Contents() {
                private int i = 11;
                public int value() { return i; }
            }; // Semicolon required in this case
    }
    public static void main(String[] args) {
        Parcel6 p = new Parcel6();
        Contents c = p.cont();
    }
}
```

```
// Destination.java file
public interface Destination {
        String readLabel();
}

// Contents.java file
 public interface Contents {
        int value();
}
```

- The returned reference is automatically upcast to a **Contents** reference

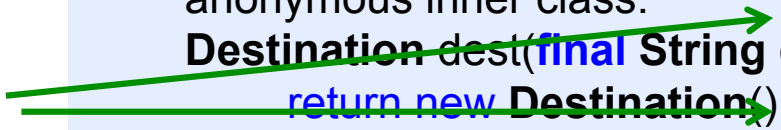
- The **anonymous inner-class** syntax is a shorthand for:

```
class MyContents implements Contents {
        private int i = 11;
        public int value() { return
        i; }                        }
                            return new MyContents();
```

Anonymous Inner Classes

Anonymous Inner Classes Initialization

An outside object used in the anonymous class must be final



```
public class Parcel8
        {                                                    // Argument
        must be final to use inside                                  //
        anonymous inner class:                    public
        Destination dest(final String dest) {
                return new Destination() {
                        private String label = dest;
                        public String readLabel() {
                                return label;
                                } // end of readlLabel
                                        };

         } // end of dest

         public static void main(String[] args) {
                Parcel8 p = new Parcel8();
                Destination d = p.dest("Tanzania");
        }
} // end of Parcel8
```

Anonymous Inner Classes

JACOBS UNIVERSITY

The link to the outer class

- The object of an inner class has a link to the enclosing object that created it

- The object of an inner class can *access members of the enclosing object*

- The outer class reference is denoted

```
OuterClassName.this
```

Anonymous Inner Classes

JACOBS UNIVERSITY

Thus

- *Inner classes have access rights to all the elements in the enclosing class*

- The inner class keeps a reference to the object of its enclosing class

  - The *(hidden) reference* is used to select members of the enclosing class

  - *An inner class object is created only in association with the object of the enclosing class*

Anonymous Inner Classes

JACOBS UNIVERSITY

General syntax

```
new SuperType (construction parameters) {
        inner class methods and data

    }
```

- **SuperType** can be an interface AND
- The *inner class* implements the interface OR

- **SuperType** can be a class (e.g., abstract class) and
- The *inner class* extends the class

Example

Link to outer class

end(), current() and *next()* refer to *obs* which isn't part of **SSelector** but a private field in the enclosing class

```
interface Selector {
    boolean end();
    Object current();
    void next();
}
```

```
public class Sequence {
      private Object[] obs;
      private int next = 0;
      public Sequence(int size) {
        obs = new Object[size];
      }
     …
      private class SSelector implements Selector {
          int i = 0;
          public boolean end() {
            return i == obs.length;
          }
          public Object current() {
                    return obs[i];
          }
          public void next() {
                    if (i < obs.length) i++;
          }
      }
    …
    }
```

Static Inner Classes

If you don't need a connection between the inner class object and the outer class object, then you can make the inner class static

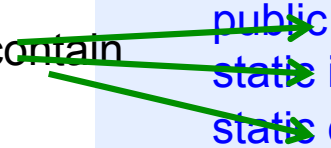
static inner class means

- No outer-class object is required to create an object of a static inner class
- An outer-class object can't be accessed from an object of a static inner class
- Non-static inner classes cannot have static data, **static** fields, or **static** inner classes, but static inner classes can have all these

Example

JACOBS UNIVERSITY

Static inner classes can contain other static elements



```
public class Parcel10 {
  private static class PContents implements Contents {
        private int i = 11;
        public int value() { return i; }
  }
  protected static class PDestination implements Destination {
        private String label;
        private PDestination(String whereTo) {
              label = whereTo;
        }
        public String readLabel() { return label; }
        public static void f() {}
        static int x = 10;
        static class AnotherLevel {
              public static void f() {}
              static int x = 10;
        }
  }
  …
}
```

Example cont…

No object of **Parcel10** is necessary

- Use the normal syntax for selecting a static member to call the methods that return references to **Contents** and **Destination**

```
public static Destination dest(String s) {
      return new PDestination(s);
 }
 public static Contents cont() {
      return new PContents();
 }
 public static void main(String[] args) {
      Contents c = cont();
      Destination d = dest("Tanzania");
 }
}
```

Static Inner Classes

JACOBS UNIVERSITY

Static inner classes inside interfaces

- A static inner class can be part of an interface

```
interface IInterface {
        static class Inner {
                                int i, j, k;
                                                public
        Inner() {}
        void f() {}
        }
}
```

- The static inner class is only placed inside the namespace of the interface, thus does not violate interface rules

Creating Instances of Inner Classes

JACOBS UNIVERSITY

```
public class Parcel11 {
    class Contents {
        private int i = 11;
        public int value() { return i; } }
            class Destination {
                private String label;
                Destination (String
    whereTo) {                          label =
    whereTo;                            }

    String readLabel() { return label; }
    }
    public static void main(String[] args) {
    Parcel11 p = new Parcel11();
    Parcel11.Contents c = p.new Contents();
    Parcel11.Destination d = p.new
                        Destination("Tanzania");
        }
    }
```

Must use instance of outer class to create an instances of the inner class

Tells **p** object to create **Contents** and **Destination** objects respectively

Multiply-Nested Classes

JACOBS UNIVERSITY

Classes can be deeply nested

Class with three levels of nesting

```
class MNA {
        private void f() {}
        class A {
                private void g() {}
                public class B {
                        void h() {
                                g();
                                f();
                        }
                }
        }
}
public class MultiNestingAccess {
        public static void main(String[] args) {
        MNA mna = new MNA();
        MNA.A mnaa = mna.new A();
        MNA.A.B mnaab = mnaa.new B();
        mnaab.h();
        }
}
```

.new Syntax for creating objects in nested classes

In **MNA.A.B**, methods **g( )** and **f( )** are callable without any qualification

Inner Class Identifiers

JACOBS UNIVERSITY

- Every class produces a .class file that holds information about how to create objects of this type

- Inner classes also produce a .class file

- Inner classes are named ***name of the enclosing class, followed by a '$', followed by the name of the inner class***

- Example

```
WithInner$Inner.class
```

Reading Assignment

- Horstmann, C. S. & Cornell, G. (2013) Core Java(TM) 2 (Vol. I), Chapter 6. Prentice Hall, 9th Edition.