Inheritance 2

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Wolfgang.Schreiner@risc.jku.at
http://www.risc.jku.at

Method Overriding

Methods in Superclasses

```
public class Article
{
  public int number;
  public String title;
  public int price;
  ...
  public void printInfo()
  {
    System.out.print("Article '" + title);
    System.out.print( "' (" + number + ") costs ");
    System.out.println(price/100 + "." + price%100 + " Euro.");
  }
 }
```

Method `printInfo` may be used by any subclass of `Article`.

Application

Book and CD may use `printInfo`.

```
Book book = new Book(...);
CD cd = new CD(...);
book.printInfo();
cd.printInfo();
```

- Problem: `printInfo` is too general.
  - Books and CDs have special information that should be printed as well.

We want special versions of `printInfo` for the subclasses.

Method Overriding

A method declaration may be overridden in a subclass.

```
public class Book extends Article
{ public String author;
  public String publisher;
  public String ISBN;
  ...
  public void printInfo()
  { System.out.print("Article '" + title);
    System.out.print( "' (" + number + ") costs ");
    System.out.println(price/100 + "." + price%100 + " Euro.");
    System.out.print("This is a book written by " + author);
    System.out.print(" and published by " + publisher);
    System.out.println(" with ISBN number " + ISBN + ".");
  }
}
```

Application

```
Book book = new Book(...);
book.printInfo();
```

- Method `printInfo` of class `Book` is called.
  - The corresponding method declaration in class `Article` has been overridden.

Problem: a lot of code has been duplicated in the new declaration.

Calling Methods in Superclasses

```
public class Book extends Article
{
  public String author;
  public String publisher;
  public String ISBN;
  ...
  public void printInfo()
  {
    super.printInfo();
    System.out.print("This is a book written by " + author);
    System.out.print(" and published by " + publisher);
    System.out.println(" with ISBN number " + ISBN + ".");
  }
}
```

`super` refers to `this` object as an object of the superclass.

Calling Methods in Superclasses

```
public class Book extends Article
{
  ...
  public void printInfo() { ... }
  ...
  public void printArticleInfo()
  {
    super.printInfo();
  }
}

Book book = new Book(...);
book.printArticleInfo();
```

Any method may refer to an overridden method.

Shadowing Variables

```
public class Article
{
  public int price;
  ...
}

public class Book extends Article
{
  public String price; // shadows int declaration
  ...
}
```

Do not shadow variable declarations of superclasses.

Final Methods and Classes

We may prevent unwanted extensions.

- A `final` method can not be overridden:

```
public final type method(...)
{
  ...
}
```

- A `final` class is not the parent of any other class:

```
public final class Class
{
  ...
}
```

The compiler can create more efficient code for method calls.

Abstract Classes and Frameworks

Abstract Classes

- A class `C` can serve two purposes:
  - For creating objects: `c = new C(...)`
  - For inheritance: `public class D extends C { ...}`
- Sometimes we want a class only for the second purpose.
  - Class provides functionality for use and extension by subclasses.
  - Class itself is not intended for instantiation.

Abstract classes are only used for extension by other classes.

Example

A method demands that an object provides a String representation.

```
public static void printNice(Printable o)
{
  System.out.println("-- begin object ------------------ ");
  System.out.println(o.getString());
  System.out.println("-- end object -------------------- ");
}

public abstract class Printable
{
  public abstract String getString();
}
```

A concrete subclass of the abstract class provides the implementation.

Example

```
public class MyInteger extends Printable
{
  public int i;
  public MyInteger(int i) { this.i = i; }

  public String getString()
  {
     return Integer.toString(i);
  }
}
```

```
MyInteger i = new MyInteger(7);      Printable i = new MyInteger(7);
printNice(i);                        printNice(i);
```

Abstract Classes versus Interfaces

Abstract classes are in some respect similar to interfaces.

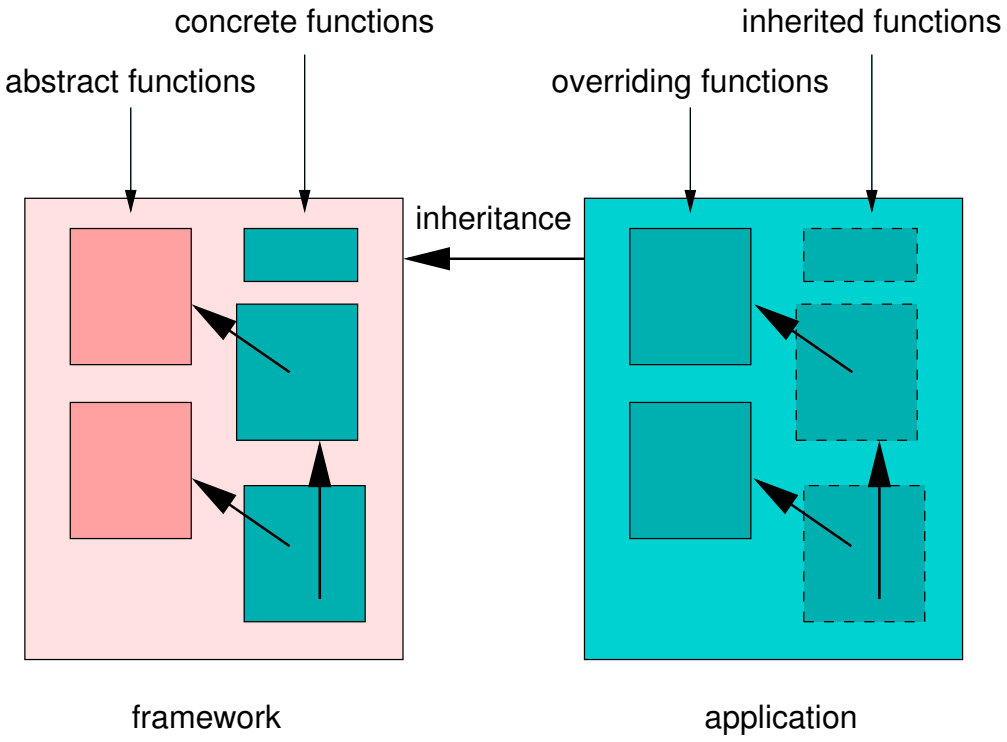
- Two main differences:
  - A class may only inherit from one abstract class but it may implement multiple interfaces.

```
public class B extends A { ... }
public class B implements I1, I2, ... { ... }
```

  - In an interface, all methods are abstract; in an abstract class (a framework), some methods may be not abstract.
    - Non-abstract methods provide skeleton functionality to subclasses.
    - Abstract methods are "gaps" to be filled by subclasses.
      - Subclass must fill skeleton with "flesh".

Use abstract classes, if you also want to provide some functionality.

Framework



A framework is an abstract class that provides some functionality.

Frameworks

```
public abstract class Printable
{
  public abstract String getString();
  public void printNice()
  {
    System.out.println("-- begin object ------------------ ");
    System.out.println(getString());
    System.out.println("-- end object -------------------- ");
  }
}

Printable i = new MyInteger(7);
i.printNice();
```

MyInteger provides `getString` and inherits `printNice`.

Frameworks

A framework may give default implementations of methods.

```
public abstract class Printable
{
  public String getString()
  {
    return "This object has no string representation."
  }

  public void printNice(Printable o)
  {
    System.out.println("-- begin object ------------------ ");
    System.out.println(getString());
    System.out.println("-- end object -------------------- ");
  }
}
```

Example: The `Applet` Class

```
java.lang.Object
  |
  +--java.awt.Component
       |
       +--java.awt.Container
            |
            +--java.awt.Panel
                 |
                 +--java.applet.Applet
```

```
public class Name extends Applet
{
   public void paint(Graphics graph)
   {
     paint operations on graph
   }
}
```

The Component Class

```
public abstract class Component implements ...
{
   ...
   public void update(Graphics g)
   {
     if (...)
     {
       g.setColor(getBackground());
       g.fillRect(0, 0, width, height);
       g.setColor(getForeground());
     }
     paint(g);
   }

   public void paint(Graphics g) { }
}
```

Visibility Modifiers

Visibility Modifiers

Classes, fields, and methods may be hidden from users.

- No visibility modifier: "default visibility"
  - Entities are visible to classes within the current package.
    - No class in other package can refer to these entities.
- Visibility modifiers: change visibility status.
  - `public`, `private`, `protected`.
    - Multiple visibility levels.

Use visibility modifiers to protect entities from unintended use.

Class Visibility

```
public class Class
{
  ...
}
```

Class becomes visible outside the current package.

Field and Method Visibility

Three different protection levels.

1. `private`:
   - Field/method can be only used within the current class.
2. `protected`:
   - Field/method can be used within the current class and its subclasses.
   - Field/method can be used by all classes in the current package.
3. `public`:
   - Field/method can be used without restriction by anyone.

Rules of Thumb

How are the visibility modifiers typically used?

- Make all classes `public`
  - Unless you want to restrict access to the current package.
- Make all class fields `private`
  - Fields are local to the class.
- Make all methods that are part of the class specification `public`.
  - Methods provide functionality of interest.
- Make all other methods `private`.
  - Methods are for internal use only.

Good rules in most situations.

Typical Class Structure

```
// class can be used by any other class
public class Class
{
  // the private fields
  private type field;

  // the public constructors
  public Class (...) { ... }

  // the public methods for the user
  public type method (...) { ... }

  // the private methods for the class
  private type method (...) { ... }
}
```