ETH Zürich
Chair of Software Engineering

# Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

## Java: introduction to object-oriented features

# Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

## Java classes and objects

# Classes and objects

- The basic encapsulation unit is the class
  - as in every object-oriented language
- A class is made of a number of features (or members)
  - instance variables (attributes, fields)
  - methods
- Classes and features have different levels of visibility
- Objects are class instances
  - and classes are sets of objects
    - or blueprints for creating objects
  - constructors are special methods to create new objects
  - in Java, objects are automatically destroyed when no longer referenced (garbage collection)
    - no destructors, but optional finalize methods

# A simple class example

```
package ch.ethz.inf.se.javacsharpindepth;
/**
 * @author John H. Doe
 */
public class MainClass {
          // 'main' must be all lowercase
     public static void main (String[] args) {
          Game myGame = new Game();
          System.out.println("Game starts!");
          myGame.startGame();
     }
}
```

# Attributes (instance variables)

- Relate to a class instance
- Declared within the class curly brackets, outside any method
- Visible at least within the class scope, within any method of the class
- Automatically initialized to the default values
  - `0` or `0.0` for numeric types, '`\u0000`' for chars, **`null`** for references, **`false`** for booleans

# Methods (instance methods)

- Relate to an instance and are declared within the class curly brackets

- May have arguments
- Must have return type (possibly **`void`**)

```
boolean test(int i, boolean b){
      // some stuff here
      return true;
}
```

- Constructors are “special” (more on this later)

# Information hiding

Attribute and method visibility "modifiers":

- `public`: visible everywhere
- `protected`: visible in the same package and in subclasses (wherever they are)
- (*): visible in the same package
- `private`: visible only in the class in which it is defined

Class visibility

- Top level classes can only have default or public visibility
- Nested classes can have any chosen visibility level
  - (except for inner classes: see later)

(*) No keyword for "package" visibility: it's the default

# The `static` modifier

When applied to non-local variables and methods

- Relates to a specific class, not to a class instance
- Shared by every object of a certain class
- Accessed without creating any class object

```
MyClass.myStaticAttribute
MyClass.myStaticMethod()
```

The `static` modifier does not apply to top-level classes in Java

# Constructors

- Same name as the class
- No return type (not even **`void`**)
- An argumentless constructor is provided by default if no other constructor is explicitly given

# Local variables

- Declared within a method’s scope
(denoted by curly brackets)

- Visible only within the method’s scope

- De-allocated at method end

- Not automatically initialized
  - warning if no explicit initialization is given

# The keyword this

Refers to the current object

```
public class Card {

    private int value;

    public int getValue() {
        return value;
    }
    public void setValue(int value) {
        this.value = value;
    }
}
```

# Nested classes

A class defined inside another class, that may access its private data. (Nested is the opposite of “top-level”.)

Variants of nested classes

- Inner class: non-static nested class
  - can reference the outer class instance
  - there’s a one-to-one correspondence between instances of the containing and inner class
- `static` nested class
  - no references to the outer class (non-static) instance
- Anonymous (inner) class: inner class without a name, defined in the middle of a method or initialization block
  - no visibility specifiers allowed
- Local (inner) class: inner class with a name, defined in the middle of a method or initialization block
  - no visibility specifiers allowed

# Anonymous inner class example

```java
public void start(int num) {
      // ActionListener is an interface
      ActionListener listener = new ActionListener()
      // anonymous inner class starts here
      {
         public void actionPerformed(ActionEvent e) {
            // reaction code here; may refer to num
         }
      }; // anonymous inner class ends here
      // other code here
}
```

Which design pattern does this example suggest?

# Anonymous inner class example

```
public void start(int num) {
      // ActionListener is an interface
      ActionListener listener = new ActionListener()
      // anonymous inner class starts here
      {
         public void actionPerformed(ActionEvent e) {
            // reaction code here; may refer to num
         }
      }; // anonymous inner class ends here
      // other code here
}
```

This is an instance of the observer design pattern

# Method overloading

- Using the same name with different argument list
  - list can differ in length, argument type, or both
- Example: constructors
- Method signature: name + arguments list
  - The return type is not part of the signature
- Tip: overloading may reduce readability: don’t abuse it

# Method overloading with subtypes

When a method name is overloaded with argument types that are related by inheritance, method resolution selects the "closest" available type.

Example: `Student` is a subtype of `Person`

```
class X {
        // v1
      void foo (Person p) { }
        // v2
      void foo (Student p) { }
 }
X x = new X();
x.foo(new Person()); // Executes v1
x.foo(new Student()); // Executes v2
```

# Method overloading with subtypes

When a method name is overloaded with argument types that are related by inheritance, method resolution selects the "closest" available type.

Example: `Student` is a subtype of `Person`

```
 class Y { void foo (Person p) { ... } }
 class Z { void foo (Student p) { ... } }

Y y = new Y();
y.foo(new Person()); // OK
y.foo(new Student()); // OK

Z z = new Z();
z.foo(new Person()); // Error
z.foo(new Student()); // OK
```

# Operator overloading

- No custom operator overloading is possible
- Only “+” for **`String`** is overloaded at language level

```
System.out.println(
        “Custom operator overloading ” +
        “would have been nice…” )
```

# Method argument passing

- All the primitive types are passed by value
  - Inside the method body we work with a local copy
  - We return information using the `return` keyword
- (Object) Reference types are passed by value too, but:
  - What is passed by value is the reference (i.e., an object address)
  - Consequently, a method can change the state of the object attached to the actual arguments through the reference

# Variable number of arguments

To pass a variable number of arguments to a method:

- Use a collection (including arrays)
- From Java 5.0: varargs arguments "..."

```
public void write(String ... someStrings) {
      for (String aString : someStrings) {
          System.out.println(aString);
      }
}
```

- This is just syntactic sugar for an array
  - You can pass an array as actual
- The varargs parameter must be the only one of its kind and the last one in the signature

# Block initializers (a.k.a. initialization blocks)

- Similar to “anonymous” method bodies
  - without signature and return type, only curly brackets and possibly the `static` modifier
- The code within them is executed during initialization
- Can be `static` or non-static
- Useful to perform some computation before the constructors are invoked
  - Factor out code common to multiple constructors
  - Initialize `final static` variables

# Finalizer methods

The `Object` class includes a method:

```
protected void finalize()
```

which can be overridden in any class.

The `finalize` method is called just before garbage collection

- May never be called, if an object is not collected
- No real-time guarantee that the object is collected right after finalize is executed

What's for: do some final clean-up upon object disposal

- E.g.: resources not properly released beforehand

It is not meant for general release of resources

- Files and other I/O resources have "close/destroy" methods, which should be called explicitly

# Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

## Inheritance, polymorphism, and dynamic dispatching

# Inheritance

- We can explicitly “extend” from one class only
  - Otherwise, every class implicitly extends `Object`
- Public and protected inherited fields and methods are available in the heir.
- Package-visible (no visibility specifiers) inherited members are visible only in heirs within the same package.

# Overriding and dynamic dispatching

- Overriding: method redefinition in a subclass
- Overriding rule:
  - (before Java 5.0) overriding method must have the same signature and return type as in the superclass
  - (from Java 5.0) overriding method must have the same signature as in the superclass and a covariant return type of the superclass

- Annotation **`@Override`** avoids compiler warning
- Dynamic dispatching applies
- The keyword **`final`** prevents overriding in subclasses
- Overriding cannot reduce the visibility of a method
  - e.g.: from **`public`** to **`private`**
- No overriding for **`static`** methods

# Covariant return types example

In Java 5.0 the return type of an overridden method can be a subtype of the base method’s return type.

```
class Account { ... }

class SavingsAccount extends Account { ... }

class AccountManager {
      public Account GetAccount() { ... }
}

class SavingsAccountManager extends AccountManager {
      public SavingsAccount GetAccount() { ... }
}
```

# Casting and Polymorphism

Casting is C++/Java/C# jargon to denote polymorphic assignments.

- Let S be an ancestor of T (that is, T $\rightarrow^*$ S)
  - Upcasting: an object of type T is attached to a reference of type S
  - Downcasting: an object of type S is attached to a reference of type T

```
class Vehicle;
class Car extends Vehicle;
Vehicle v =(Vehicle)new Car(); // upcasting
Car c = (Car)new Vehicle(); // downcasting
```

# Casting in Java

- Upcasting is implicit
  - For primitive types, upcasting means assigning a “smaller” type to a “larger” compatible type
    - `byte` to `short` to `int` to `long` to `float` to `double` (`long` to `float` may actually lose precision)
    - `char` to `int`
  - For reference types, upcasting means assigning a subtype to a supertype, that is:
    - a subclass to superclass
    - an implementation of an interface X to that interface X
    - an interface X to the implementation of an ancestor of X
- Downcasting must be explicit
  - can raise runtime exceptions if it turns out to be impossible
- No casts are allowed for reference types outside the inheritance hierarchy

# The `instanceof` keyword

- The **`instanceof`** keyword performs runtime checking of the dynamic type of a reference variable
  - Syntax: **`aVariable instanceof aType`**
  - Is the object attached to **`aVariable`** compatible with **`aType`**?
    - Compatible means of **`aType`** or one of its subtypes

# Shadowing

Variables with the same name and different (but overlapping) scopes:

- A local variable shadows an attribute with the same name: use **`this`** to access the attribute
- A subclass attribute shadows a superclass attribute with the same name
- Polymorphism does not apply
  - if a reference is superclass type and attached object is subclass type, the superclass variable is used
- Tip: avoid if possible (it may decrease readability)

# The `final` modifier

- **`final`** class
  - Cannot be inherited from

- **`final`** attribute, argument, or local variable
  - It’s a constant: cannot be redefined and must be initialized
  - (If it’s a reference: the object state can change)
  - **`final static`** attributes can only be initialized by block initializers
  - **`final`** (non-static) attributes can be set only once, and must be set by every constructor of the class (whenever initializers haven’t already set them).
  - Style tip: constant names are capitalized

- **`final`** method
  - Cannot be overridden

ETH Zürich
Chair of Software Engineering

# Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

## The object creation process

# The keyword `super`

- Enables invocation of a superclass method from within an overriding method in a subclass
- Can be used to explicitly invoke a constructor of the superclass (see next example)

# Chained constructors

Any constructor implicitly starts by executing the argumentless constructor of the parent class, unless:

- A specific constructor of the superclass is invoked using `super(...)`
- Another specific constructor of the same class is invoked using `this(...)`
- If used, `super(...)` or `this(...)` must be the first instruction

# Chained constructors: example

```
public class CreatureCard extends Card {

    int value;
    public CreatureCard(String name){
        super(name);
        // class-specific initializations
        value = 7;
    }

    public CreatureCard(int value){
        this("Big Monster");
        // class-specific initializations
        this.value = value;
    }
}
```

# Object creation process

```
MyClass obj = new MyClass();
```

(`static` members are initialized before)

- `new` allocates memory for a `MyClass` instance (all attributes, including inherited ones)
- initializes all attributes to default values

If constructor references `super` (explicitly or by default):

1. Recursive call to constructor of superclass
2. Execute `MyClass`'s initializers in their textual order
3. Execute constructor body

If constructor references `this` (another constructor X):

1. Recursive call to other constructor X
2. Execute rest of originally called constructor body

# Object creation process: example

```java
public class Person {
      int age = 1;
}

public class Student extends Person {
  { age = 6; }
  double gpa = age/2;
  public Student() { gpa += 1.0; }
}

Person p1 = new Person();         // age = 1
Person p2 = new Student();        // age = 6, gpa = 4.0
```