ETH Zürich
Chair of Software Engineering

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Java: advanced object-oriented features

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Packages

Packages

Classes are grouped in packages

- A hierarchical namespace mechanism
- Map to file system pathnames
  - one public top-level class per file
  - one package per directory
- Influence class visibility (according to modifiers)
- Even if a default anonymous package exists, it is customary to define the package explicitly:
  - E.g.:

    `ch.ethz.inf.se.java.mypkg`
- Tip: notice the useful name convention

The statements `package` and `import`

- `package` declares a package
- Classes from external packages generally need to be imported using `import`
- Classes from `java.lang` are automatically imported
- `*` makes available all classes in a package, but not those in sub-packages, so that they can be used without writing their fully-qualified names (as if they were declared in the current importing package)
  - If there is another class with the same name in the current importing package, you still need a fully-qualified name to bypass the shadowing by the local declaration

```
package ch.ethz.inf.se.java.mypkg;
import java.util.Set; // Only class Set
import java.awt.*;
import java.awt.event.*;
```

`static` imports

Introduced in Java 5.0

You can use imported `static` members of a class as if they were defined (also as `static` members) in the current class

```
import static java.lang.Math.*;
   …
double r = cos(PI * theta);
```

- When to use: for frequent access to static members of another class (avoids duplication or improper inheritance).
- Issue: where does a method come from? (Traceability)
- Tip: do not abuse!

Core packages in Java 7.0

- **`java.lang`**
  (basic language functionalities, fundamental types, automatically imported)
- **`java.util`** (collections and data structures)
- **`java.io`** and **`java.nio`**
  (old/new file operations API. **`nio`** improved in Java 7)
- **`java.math`** (multi-precision arithmetic)
- **`java.net`** (networking, sockets, DNS lookup)
- **`java.security`** (cryptography)
- **`java.sql`** (database access: JDBC)
- **`java.awt`** (native GUI components)
- **`javax.swing`**
  (platform-independent rich GUI components)

ETH Zürich
Chair of Software Engineering

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Abstract classes and interfaces

Abstract classes and interfaces

A method may or may not have an implementation

- if it lacks an implementation, it is abstract

A class whose implementation is not complete is also called abstract

- but even a fully implemented class can be declared abstract

Interfaces are a form of fully abstract classes

- they enable a restricted form of multiple inheritance
- they can only contain methods (signatures) and constants (`static final` fields)

Abstract classes and methods

- An **`abstract`** class cannot be directly instantiated
- An **`abstract`** method cannot be directly executed
- If a class has an **`abstract`** method, the class itself must be **`abstract`**
- An **`abstract`** class cannot be **`final`**
- A **`static`** method cannot be **`abstract`**
- A constructor cannot be **`abstract`**
- Useful for conceptualization and partial implementations

Interfaces

- Declared using **`interface`** instead of **`class`**
- Equivalent to a fully **`abstract`** class
  - you don't use the keyword **`abstract`** in an **`interface`**
- A way to have some of the benefits of multiple inheritance, with little hassle (e.g., selecting implementations)
- A class may **`implement`** one or more interfaces
- An interface can **`extend`** one or more interfaces

Interface use

- For typing, implementing an interface is essentially equivalent to extending a class: polymorphism applies
- All interface methods are implicitly **`abstract`** and **`public`**
- All interface attributes are implicitly **`public`**, **`static`**, and **`final`** (must be set by initializers once and for all)
- Useful for design: specify what, not how
- Tip: use interfaces to have more flexible designs (but attributes are rarely appropriate in interfaces).

Method name clash in multiple interfaces

Two interfaces `I1` and `I2` may define two methods with the same name and signature.

If a class `C` a extends both `I1` and `I2`, it must provide only one implementation of the method, that applies to both interfaces. In other words, the two methods are merged.

```
interface I1 {  int checksum(); }
interface I2 {  int checksum(); }

class C extends I1, I2 {
      int checksum() {
        return this.hashCode();
      }
}
```

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

“Special” classes and features

The String class

- Sequences of Unicode characters
- Immutable class: no setters
- If initialized upon creation as in:

```
String s = "Test";
```

  - Exists in the "string pool" in the stack
  - Uses shared memory
  - No duplicates
- `java.lang.StringBuilder` class provides mutable strings

Object comparison: `equals`

```
public boolean equals(Object obj) {
      return (this == obj);
}
```

- The default semantics compares addresses

- We can provide a different semantics by overriding
  - Implementation should be an equivalence relation
    - Reflexive, symmetric, transitive
  - For any non-null reference variable **`x`** it should be:

    `x.equals(null) == false`

Class `Object`: hash code

```
public int hashCode()
```

Returns distinct integers for distinct objects. Its specification:

- required:
  `o1.equals(o2)` implies `o1.hashCode() == o2.hashCode()`
- as much as possible:
  `o1.equals(o2)` iff `o1.hashCode() == o2.hashCode()`

Overriding `equals()` in descendants does not guarantee to give the right semantics to `hashCode()` as well.

In general, it may be necessary to explicitly override `hashCode()`, so that equal objects have equal hash codes.

Class `Object`: string representation

```
public String toString() {
    return getClass().getName() + "@" +
    Integer.toHexString(hashCode());
}
```

- Tip: all descendants should override this method
- Tip: the result should be a concise and informative representation

Arrays

- Arrays are objects
  - but with the familiar syntax to access them
- Operator [ ] to access components
- The only available attribute is **`length`**
- All components must a have a “common” type
  - a common ancestor in the inheritance hierarchy
- Array components are automatically initialized to defaults

Array use

```
// declaration
int[] iArray;
// definition: size given
iArray = new int[7];
// declaration with definition
Vehicle[] v = new Vehicle[8];
// polymorphic array (Car, Truck --> Vehicle)
v[0] = new Car();
v[1] = new Truck();
// using initializers
double[] dArray = {2.4, 4.5, 3.14, 7.77};
Vehicle[] v1 = {new Car(), new Truck()};
```

Multidimensional arrays

Multidimensional arrays in Java are just arrays of arrays

3-dimensional array, declaration only:

```
int [][][] threeDim;
```

Declaration with initialization:

```
// For 0 ≤ i < 4: twoDim[i] == null
int [][] twoDim = new int[4][];
// For 0 ≤ i < 4: twoDim[i] is array {0, 0}
int [][] twoDim = new int[4][2];
```

Jagged array: different components have different size:

```
int [][] jagged = {{3, 4, 5}, {6, 7}};
```

Enumerated types

Denote a finite set of values

```
enum TypeName {VALUE_1, ..., VALUE_N};
```

Within the type system, **TypeName** is a class that extends class **Enum** and has **N** distinct static constants

```
TypeName aValue = TypeName.VALUE_2;
```

By default, each **VALUE_k** is printed as its own name; to have a different representation, override **toString()**

A variable of **enum** type can also be **null**

An **enum** class can have constructors, attributes, and methods, with some restrictions w.r.t. a full-fledged class

Enumerated type example

```
enum EventStatus  {
      APPROVED("A"), PENDING("P"), REJECTED("R");

      private String shortForm;

// constructor must be private: not directly callable
      private EventStatus(String shortForm) {
            this.shortForm = shortForm;
      }

      public String getShortForm(){
            return shortForm;
      }
}
```

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Assertions and contracts

Contracts

Contracts are specification elements embedded in the program text. They typically extend the syntax of Boolean expressions in the language. Here's an example with Eiffel syntax.

```
class BankAccount

  balance: INTEGER

  deposit (amount: INTEGER)
    require amount > 0                      // precondition
    do balance := balance + amount
    ensure balance > old balance end // postcondition

invariant
  balance >= 0                            // class invariant
end
```

Contracts: preconditions

The precondition of a method **`M`** specifies requirements that every call to **`M`** must satisfy. It is the caller's responsibility to ensure that the precondition is satisfied.

```
ba: BankAccount
create ba                    // object creation

ba.deposit (120)             // valid call: 120 > 0
ba.deposit (-8)              // invalid call: -8 < 0
```

Contracts: postconditions

The postcondition of a method **M** specifies conditions that hold whenever an invocation to **M** terminates. **M**'s body is responsible to ensure that the postcondition is satisfied.

```
ba: BankAccount
create ba                       // object creation
// assume 'balance' is 20

ba.deposit (10)
      // postcondition ok: 30 > 20
ba.deposit (MAX_INTEGER)
      // postcondition violation if balance
         silently overflows into the negatives
```

Contracts: class invariants

The class invariant of a class `C` constrains the states that instances of the class can take. The class invariant's semantics is a combination of the semantics of pre- and postcondition: the class invariant must hold upon object creation, right before every qualified call to public members of `C`, and right after every call terminates.

```
ba: BankAccount
create ba                      // object creation
// class invariant must hold

// class invariant must hold
ba.deposit (10)
// class invariant must hold
```

Assertions

Java doesn't natively support contracts, but offers assertions: checks that can be executed anywhere in the code:

```
assert boolean-expr [:"message"];
```

- If evaluates to true, nothing happens
- If evaluates to false, throw `AssertionError` and display "message"

- Assertion checking is disabled by default
- Can be enabled at VM level, with different granularities
  - `java -ea MyClass` (`-da` to disable)
  - `java -esa MyClass` (for system classes assertions)
  - `java -ea:mypkg... -da:mypkg.subpkg MyClass` ("`...`" means: do the same for subpackages)

- Available since Java 1.4

Contracts as assertions

We can use assertions to render the semantics of contracts:

```
public class BankAccount {

   int balance = 0;

   void deposit(int amount) {
     int old_balance = balance;
     assert amount > 0 : "Pre violation";
     balance += amount;
     assert balance > old_balance :"Post violation";
    }
}
```

No explicit support for class invariants

- Can we render their semantics with **assert**?

JML: Java Modeling Language

- JML offers full support for contracts, embedded through Javadoc-like annotations

```
public class BankAccount {
    int balance = 0;
    /*@ requires amount > 0;
      @ ensures balance > \old(balance);
      @*/
    void deposit(int amount) {
      balance += amount;
    }
    //@ invariant balance >= 0;
}
```

- JML is not part of the standard Java platform, and hence requires specific tools to process the annotations
- Documentation and resources: http://www.jmlspecs.org