Course of Programming in Java

by Łukasz Stafiniak

Email: lukstafi@gmail.com, lukstafi@ii.uni.wroc.pl
Web: www.ii.uni.wroc.pl/~lukstafi

The Java Programming Language

Chapter 2: Classes and Objects

Chapter 3: Extending Classes

Chapter 6: enums

by Ken Arnold, James Gosling, David Holmes

Classes

public | abstract final | class Body {

public protected private | static | final | long idNum;

```
    public String name;
    public CelestialBody orbits;
    public static long nextID = 0;
```

public protected private | static | abstract final | String showOrbit() {

final | String s = (orbits == null ?

```
                    name : name + " around " + orbits.name);
    }
}
...
        Body mercury; ...
```

Class instance (i.e. object) initialization

the following are all valid field initializers (can occur inside class body):

```
double zero = 0.0;                  // constant
double sum = 4.5 + 3.7;             // constant expression
double zeroCopy = zero;             // field
double rootTwo = Math.sqrt(2); // method invocation
double someVal = sum + 2*Math.sqrt(rootTwo); // mixed
```

If a field is not initialized a default initial value is assigned to it depending on its type:

Type	Initial Value
`boolean`	`false`
`char`	`'\u0000'`
`byte, short, int, long`	`0`
`float, double`	`+0.0`
object reference	`null`

Static fields

we could print the value of `nextID` as follows:

```
System.out.println(Body.nextID);
```

A static member may also be accessed using a reference to an object of that class, such as

```
System.out.println(mercury.nextID);
```

Final fields

If a `final` field does not have an initializer it is termed a blank final.

Definitely (un)assigned analysis.

When you decide whether a field should be `final`, consider three things:

- Does the field represent an immutable property of the object?
- Is the value of the field always known at the time the object is created?
- Is it always practical and appropriate to set the value of the field when the object is created?

Creating objects

```
Body sun = new Body(); // implicitly provided constructor

sun.idNum = Body.nextID++;
sun.name = "Sol";
sun.orbits = null; // in solar system, sun is middle

Body earth = new Body();
earth.idNum = Body.nextID++;
earth.name = "Earth";
earth.orbits = sun;
```

ugly...

Constructors

```
class Body {
    public long idNum;
    public String name = "<unnamed>";
    public Body orbits = null;
    private static long nextID = 0;

    Body() {
        idNum = nextID++;
    }
}
```

or better, add:

```
Body(String bodyName, Body orbitsAround) {

    this();
    name = bodyName;
    orbits = orbitsAround;
}
```

Now the allocation code is much simpler:

```
Body sun = new Body("Sol", null);

Body earth = new Body("Earth", sun);
```

Initialization blocks

```
class Body {

    public long idNum;
    public String name = "<unnamed>";
    public Body orbits = null;

    private static long nextID = 0;

    {
        idNum = nextID++;
    }

    public Body(String bodyName, Body orbitsAround) {
        name = bodyName;
        orbits = orbitsAround;
    }
}
```

- Most useful when you are writing anonymous inner classes that can't have constructors.

- useful to define a common piece of code that all constructors execute. While this could be done by defining a special initialization method, say `init`, the difference is that such a method would not be recognized as construction code and could not, for example, assign values to blank final fields.

- *Use sparingly*.

Static initialization

```
class Primes {

    static int[] knownPrimes = new int[4];

    static {
        knownPrimes[0] = 2;
        for (int i = 1; i < knownPrimes.length; i++)
            knownPrimes[i] = nextPrime();
    }
    // declaration of nextPrime ...
}
```

Methods

Methods with Variable Numbers of Arguments

For example, in

```
public static void print(String... messages) {
    // ...
}
```

`messages` is a `String[]`.

```
...

   print(“Any”, “number”, “of”, “arguments”); ...
```

Return multiple values

- return references to objects that store the results as fields,
- take one or more parameters that reference objects in which to store the results
- return an array that contains the results

```
public class Permissions {

    public boolean canDeposit,
                   canWithdraw,
                   canClose;
}
```

Here is a method that fills in the fields to return multiple values:

```
public class BankAccount {

    private long number;    // account number
    private long balance;   // current balance (in cents)

    public Permissions permissionsFor(Person who) {
        Permissions perm = new Permissions();
        perm.canDeposit = canDeposit(who);
        perm.canWithdraw = canWithdraw(who);
        perm.canClose = canClose(who);
        return perm;
    }

    // ... define canDeposit et al ...
}
```

Parameter values

```
class PassByValue {
    public static void main(String[] args) {
        double one = 1.0;

        System.out.println("before: one = " + one);
        halveIt(one);
        System.out.println("after:  one = " + one);
    }

    public static void halveIt(double arg) {
        arg /= 2.0;     // divide arg by two
        System.out.println("halved: arg = " + arg);
    }
}
```

The following output illustrates that the value of `arg` inside `halveIt` is divided by two without affecting the value of the variable `one` in `main`:

```
before: one = 1.0

halved: arg = 0.5
after:  one = 1.0
```

```
class PassRef {
    public static void main(String[] args) {
        Body sirius = new Body("Sirius", null);

        System.out.println("before: " + sirius);
        commonName(sirius);
        System.out.println("after:  " + sirius);
    }
    public static void commonName(Body bodyRef) {
        bodyRef.name = "Dog Star";
        bodyRef = null;
}}
```

This program produces the following output:

```
before: 0 (Sirius)

after:  0 (Dog Star)
```

"The Java programming language does not pass objects by reference; it passes object references by value."

Exercise 1. What will happen when we pass a `Double` argument above?

Overloading methods

```
public boolean orbitsAround(Body other) {

    return (orbits == other);
}
public boolean orbitsAround(long id) {
    return (orbits != null && orbits.idNum == id);
}
```

extremely poor use of overloading:

```
public static void print(String title) {
    // ...
}
public static void print(String title, String... messages) {
    // ...
}
public static void print(String... messages) {
    // ...
}
```

Given the invocation

```
print("Hello"); // which print ?
```

unique match when ignoring *varargs* methods.

In contrast, this invocation is ambiguous and results in a compile-time error:

```
print("Hello", "World");  // INVALID: ambiguous invocation
```

Importing static member names

hyperbolic tangent (tanh): (actually, `tanh` belongs to `Math`)

```
static double tanh(double x) {
    return (Math.exp(x) - Math.exp(-x)) /
           (Math.exp(x) + Math.exp(-x));
}
```

Using:

```
import static java.lang.Math.exp;
```

we can rewrite our example more clearly as:

```
static double tanh(double x) {
    return (exp(x) - exp(-x)) /
           (exp(x) + exp(-x));
}
```

A static import statement must appear at the start of a source file, before any class or interface declarations.

Enums

Enum constants (instances of the enum class, calling `new` on `E` forbidden):

```
enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
```

```
Suit currentSuit = ... ;
if (currentSuit == Suit.DIAMONDS) ... ;
```

Every enum type `E` has two static methods that are automatically generated for it by the compiler:

```
public static E[] values()
```

```
public static E valueOf(String name)
```

Constant-specific behavior

```
enum ChessPiece {
    PAWN { // you can't forget to handle the case for a specific value
        Set<Position> reachable(Position current) {
            return ChessRules.pawnReachable(current);
        }
    },
    ROOK {
        Set<Position> reachable(Position current) {
            return ChessRules.rookReachable(current);
        }
    },
    // ... anonymous subclasses of ChessPiece
    QUEEN {
        Set<Position> reachable(Position current) {
            return ChessRules.queenReachable(current);
        }
    };
    // declare the methods defined by this enum
    abstract Set<Position> reachable(Position current);
}
```

Extending Classes

The ability to extend classes interacts with the access control mechanisms to expand the notion of **contract** that a class presents. Each class can present two different contracts: one for users of the class and one for extenders of the class. Both of these contracts must be carefully designed.

With class extension, **inheritance of contract** and **inheritance of implementation** always occur together. However, you can define new types independent of implementation by using interfaces.

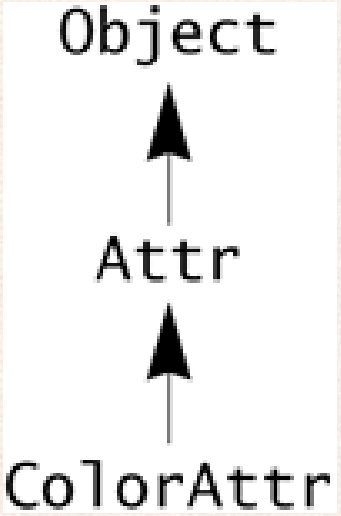
```
public class Attr {

    private final String name;
    private Object value = null;

    public Attr(String name) {
        this.name = name;
    }

    public Attr(String name, Object value) {
        this.name = name;
        this.value = value;
    }

    public String getName() {
        return name;
    }

    public Object getValue() {
        return value;
    }
```

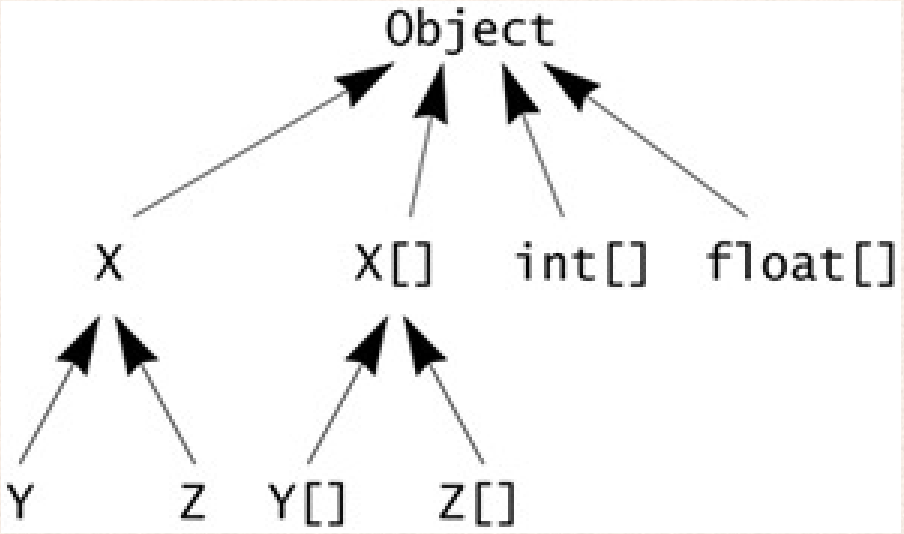
```
    public Object setValue(Object newValue) {
        Object oldVal = value;
        value = newValue;
        return oldVal;
    }

    public String toString() {
        return name + "='" + value + "'";
    }
}
```

```
class ColorAttr extends Attr {
    private ScreenColor myColor; // the decoded color

    public ColorAttr(String name, Object value) {
        super(name, value); // let the superclass ‘‘honor its contract’’
        decodeColor();
    }
    public ColorAttr(String name) {
        this(name, "transparent");
    }
    public ColorAttr(String name, ScreenColor value) {
        super(name, value.toString());
        myColor = value;
    }
    public Object setValue(Object newValue) {
        // do the superclass's setValue work first
        Object retval = super.setValue(newValue);
        decodeColor();
        return retval;
    }
```

```
    /** Set value to ScreenColor, not description */
    public ScreenColor setValue(ScreenColor newValue) {
        // do the superclass's setValue work first
        super.setValue(newValue.toString());
        ScreenColor oldValue = myColor;
        myColor = newValue;
        return oldValue;
    }

    /** Return decoded ScreenColor object */
    public ScreenColor getColor() {
        return myColor;
    }

    /** set ScreenColor from description in getValue */
    protected void decodeColor() {
        if (getValue() == null)
            myColor = null;
        else
            myColor = new ScreenColor(getValue());
    }
}
```

Single-rooted hierarchy:





Initialization

When an object is created, memory is allocated and cleared for all its fields, including those inherited from superclasses.

After this, construction has three phases:

1. Invoke a superclass's constructor.

   If not given, the implicit constructor `super()` is used.

2. Initialize the fields using their initializers and any initialization blocks.

3. Execute the body of the constructor.

Any methods can be called.

Overriding

- **overloading**: add a new method, same name, different signatures
- **overriding**: "replace" a method, same name and signatures

  old method can still be accessed, e.g. `super.setValue(newValue)`
- **hiding** a field (superclass methods keep using the old field)

A method can be overridden only if it is accessible (e.g. not `private`).

Static methods can only be hidden like fields, but anyway they should be accessed via the name of its declaring class (unaffected).

```
class SuperShow {
    public String str = "SuperStr";
    public void show() {
         System.out.println("Super.show: " + str);
    }
}
class ExtendShow extends SuperShow {
    public String str = "ExtendStr";
    public void show() {
         System.out.println("Extend.show: " + str);
    }
    public static void main(String[] args) {
        ExtendShow ext = new ExtendShow();
        SuperShow sup = ext;
        sup.show();
        ext.show();
        System.out.println("sup.str = " + sup.str);
        System.out.println("ext.str = " + ext.str);
    }
}
```

There is only one object, but we have two variables containing references to it: One variable has type `SuperShow` (the superclass) and the other variable has type `ExtendedShow` (the actual class). Here is the output of the example when run:

```
Extend.show: ExtendStr

Extend.show: ExtendStr
sup.str = SuperStr
ext.str = ExtendStr
```

```
class Base {
    /** return the class name */
    protected String name() {
        return "Base";
    }
}

class More extends Base {
    protected String name() {
        return "More";
    }
    protected void printName() {
        Base sref = (Base) this;

        System.out.println("this.name()  = " + this.name());
        System.out.println("sref.name()  = " + sref.name());
        System.out.println("super.name() = " + super.name());
    }
}
```

Although `sref` and `super` both refer to the same object using the type `Base`, only `super` will ignore the real class of the object. Here is the output of `printName`:

```
this.name()  = More

sref.name()  = More
super.name() = Base
```

Type Compatibility and Conversion

A subclass instance can be used in place of a superclass parameter or assigned to a superclass variable.

A superclass instance can be explicitly cast to any of its subclasses.

```
if (sref instanceof More)
    mref = (More) sref;
```

If the compiler can tell that a cast is incorrect then a compile time error can occur. If the compiler cannot ascertain that the cast is correct at compile time, then a run time check will be performed. If the run time check fails because the cast is incorrect, then a `ClassCastException` is thrown.

TO BE CONTINUED...

Thank You