Object-Oriented Programming

CSE703029

- Faculty of Computer Science
- Phenikaa University
- Lecture 3: Initialization & Cleanup

Slides adapted from Steven Roehrig

Initialization

- In “C”-style programming, structures were glued-together primitive types, and functions were separate.
- If a structure needed initialization, the programmer had to remember to do it.
- We often forgot…
- Just as bad, we also forgot to “clean up”

What Needs to be Initialized?

- A stream for file reading needs to be attached to the file.
- An array of Vectors needs to have the Vectors created (and themselves initialized).
- A Checkbox needs to have its state set, and perhaps be associated with an ActionListener.
- A Socket needs to have its IP address set.
- A Rectangle needs to have its dimensions and location set.
- Etc.

What If We Forget?

- Things don't act the way we expect them to!
- We only learn about problems at runtime.
- Maybe we don't find out until it's too late.
- Common culprits:
  - references that lead nowhere
  - garbage values

How Does Java Help?

- Java initializes all class member variables to zero whenever we create an object.
- Java allows us to write ***constructors***, special methods, one of which will be called on object creation.
- Java refuses to create an object (compile error) if we haven't provided the right kind of constructor.

Constructors

- A constructor method has the same name as the class. It has no return type.
- There can be many different constructors, each with a distinct *argument signature*.
- (This implies that overloaded methods are OK in Java.)
- You specify the particular constructor you want when you create an object.

Example Constructor-class Book

```
public class Book {
   String title;
   String author;
   int numPages;
   Book() { }        ; // default constructor
   public Book(String t, String a, int p) {
   title = t;
   author = a;
   numPages = p;
  }

  public static void main(String[] args) {
   Book myObj = new Book("a","b",2); //New book
   System.out.println(myObj.title);
  }
}
```

Example Constructor

```
class Book {
    String title;
    String author;
    int numPages;
    Book() { }                    // default constructor
    Book(String t, String a, int p) {
        title = t;
        author = a;
        numPages = p;
}
```

A *default constructor* has no arguments (but still has the same name as the class).

Making Books

- **Book uselessBook = new Book();**
  - title is an empty character sequence
  - author is an empty character sequence
  - numPages is 0
- **Book usefulBook = new Book("The TeXBook", "Donald Knuth", 483);**

Method Overloading

- Methods with the same name, but different sets of arguments.
- A natural idea (carWash the car? shirtWash the shirt? dogWash the dog? Nah…)
- Constructors can be overloaded; so can any function.
- This is OK, but not recommended:
  - void print(String s, int i)
  - void print(int i, String s)
- You can't overload on the return type alone.

Overloading With Primitives

- The compiler tries to find an exact match, but will promote ("widen") a value if necessary.

```
void doSomething(long l) { // whatever }
:
int i = 13;
doSomething(i);
```

- The compiler won't narrow without an explicit cast.

The Default Constructor

- "But, how come I didn't have to write constructors for the last homework?"
- The compiler will write one for you!
- But ***only*** if you haven't written any constructors at all (for this class).
- A *default constructor* has no arguments (but still has the same name as the class).

A Common Error

```
class Book {
    String title; String author; int numPages;
    Book(String t, String a, int n) {
        title = t; author = a, numPages = n;
    }
}
Book b = new Book();
```

- The compiler gives an error.
- Normally, you always provide a default constructor that does as much as possible (but not too much!).

The **this** Keyword

- The **this** keyword refers to the current object in a method or constructor.

The **this** Keyword

- A common "C" idiom:

  MusicFile f = new MusicFile("Yardbirds")

  play(&f, 4); // play the 4th track

- In object-oriented style, we want to "send a message" to an object, so in Java we say

  f.play(4);

- The compiler knows which object (**f** in this case) the method is being called for.
- The compiler sends this information to the method, in the form of a reference to **f**.

The **this** Keyword (cont.)

- If necessary, we can get a reference to the "current" object; it's called **this**.

```
public class Leaf {
    int i = 0;
    Leaf increment() {
        i++;
        return this;
    }
    void print() { System.out.println("i = " + i); }
    public static void main(String[] args) {
        Leaf x = new Leaf();
        x.increment().increment().increment().print();
    }
}
```

Other Uses of **this**

```
public class Flower {
    int petalCount = 0;
    String s = new String("null");
    Flower(int petals) { petalCount = petals; }
    Flower(String ss) { s = ss; }
    Flower(String s, int petals) {
        this(petals);
//!     this(s);                    // can't do it twice
        this.s = s;
    }
    Flower() { this("hi", 47); }          // default constructor
}
```

So, What Is A **static** Method?

- It's a method that belongs to the class but not to any instance.
- It's a method "with no **this**".
- You can't call non-**static** methods from within a **static** method.
- You can call a **static** method without knowing any object of the class.

Cleanup

- Java has a *garbage collector* that reclaims memory.
- If an object “can’t be reached” by a chain of references from a reference on the stack (or static storage), it is garbage.
- There is no guarantee that such an object will be garbage collected.
- Garbage collection is not like destruction (in the C++ sense).

Member Initialization

- Unitialized variables are a common source of bugs.
  - Using an unititialized variable in method gets a compiler error.
  - Primitive data members in classes automatically get initialized to “zero”.
- Is the initialized value (zero) any better than a “garbage value”?

Member Initialization (cont.)

- You can initialize in a class definition:

```
class Notebook {
    long ram = 1048576;
    String name = new String("IBM");
    float price = 1995.00;
    Battery bat = new Battery();
    Disk d;    // a null reference
    int i = f();
}
```

- This is *very* surprising to C++ programmers!

Constructors Again

- You can have both class initialization and constructor initialization:

```
class Counter {
   int i = 1;
   Counter() { i = 7; }
   Counter(int j) {  };
   ;
```

- The order of initialization follows the order of the initialization statements in the class definition.
- It's done before any constructor initialization, so it may be done twice (as **Counter** illustrates).

Static Member Initialization

- Same story; primitives get zero unless initialized, references get null unless initialized.
- Static initialized either
  - when the first object of the type is created, or
  - at the time of the first use of the variable.
- If you never use it, it's never initialized.

Java Encapsulation: Get and Set

- **Encapsulation:** to make sure that "sensitive" data is hidden from users.
- To achieve, must:
- declare class variables/attributes as **private**
- provide **public** **get** and **set** methods to access and update the value of a private variable

Get & Set Methods

- Get returns the variable value,
- Set sets the value.

Get & Set

```
public class Person {

    private String name; // private = restricted access

    // Getter

    public String getName() {
    return name; }

    // Setter

    public void setName(String newName) {
     this.name = newName; }

}
```

Add **toString()** to Class A

```
class A {
    int i;
    public String toString() {
        return new String("" + i);
        // or this:
        // return “” + i;
        // but not this:
        // return i;
    }
}
```

Example of a Simple **Time** Class

```
public class Time {
    int hour;
    int minute;
    int second;

    Time() { setTime(0, 0, 0); }
    Time(int h) { setTime(h, 0, 0); }
    Time(int h, int m) { setTime(h, m, 0); }
    Time(int h, int m, int s) { setTime(h, m, s); }
```

Time Class (cont.)

```
Time setTime(int h, int m, int s) {
    setHour(h);
    setMinute(m);
    setSecond(s);
    return this;
}

Time setHour(int h) {
    hour = (( h >= 0 && h < 24 ) ? h : 0 );
    return this;
}
```

Time Class (cont.)

```
Time setMinute(int m) {
    minute = (( m >= 0 && m < 60 ) ? m : 0 );
    return this;
}
Time setSecond(int s) {
    second = ((s >= 0 && s < 24 ) ? s : 0 );
    return this;
}

int getHour() { return hour; }
int getMinute() { return minute; }
int getSecond() { return second; }
```

Time Class (cont.)

```
   public String toString() {
      return (“” + ( hour == 12 || hour == 0 ) ? 12 : hour % 12 ) +
         ":" + ( minute < 10 ? "0" : "" ) + minute +
         ":" + ( second < 10 ? "0" : "" ) + second +
         ( hour < 12 ? " AM" : " PM" ) ;
   }
}
```

Time Class Driver

```
public class TestTime {
    public static void main(String[] args) {
        Time t1 = new Time();
        Time t2 = new Time(20, 3, 45);

      t1.setHour(7).setMinute(32).setSecond(23);
        System.out.println("t1 is " + t1);
        System.out.println("t2 is " + t2);
    }
}
```

Miscellaneous Topics: Recursion

- Joan Rivers says “I hate cleaning my house. Before I’m even finished I have to do it again!”

```
// Joan Rivers’ algorithm (pseudo-code)
cleanTheHouse() {
    static String message = “I’m ”
    message = message + “so ”
    shout(message + “tired of this!”)
    cleanTheHouse()
}
```

Recursion

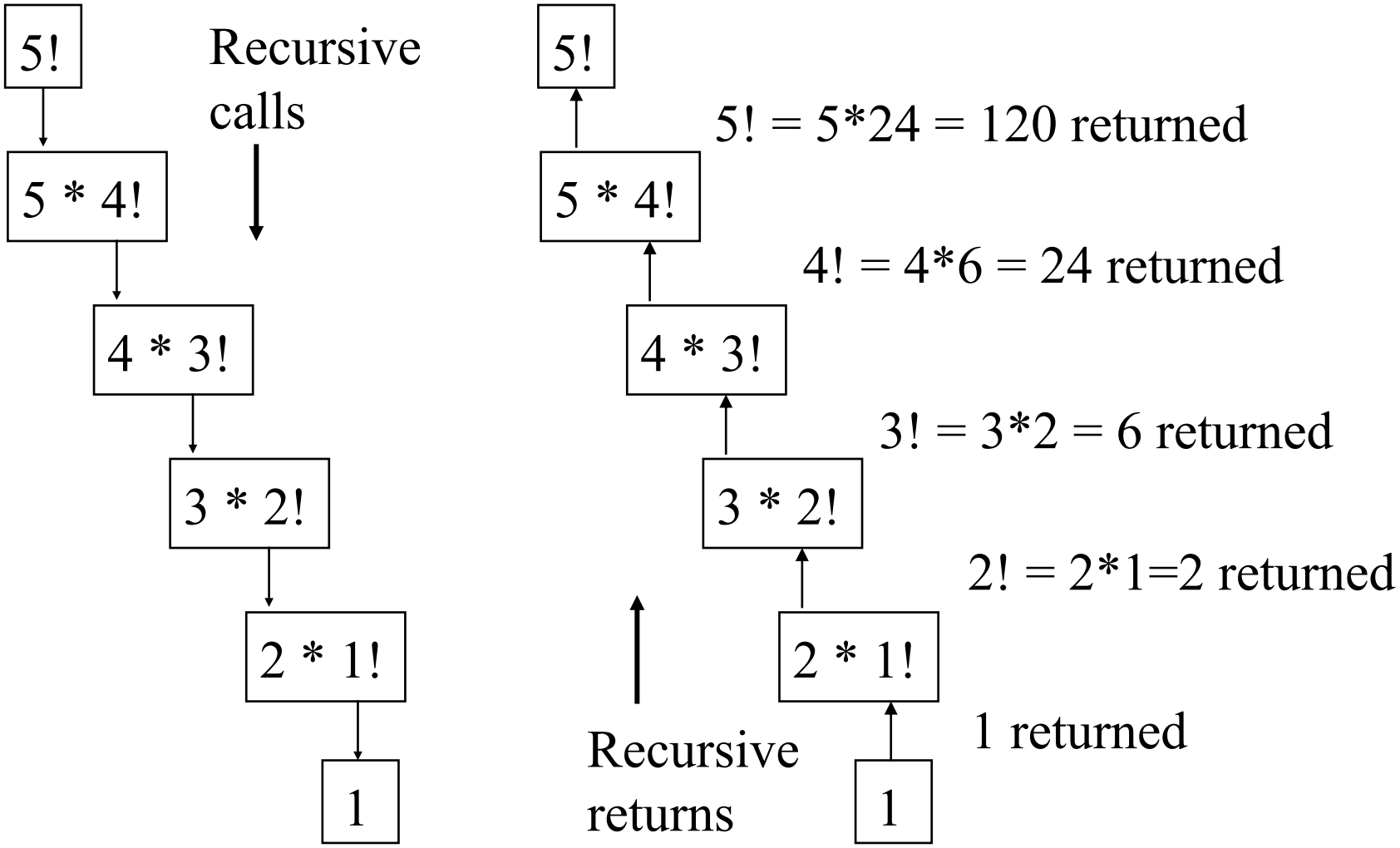
- A method that calls itself.
- At each call, new local variables are created.
- There must be a *stopping condition*! Joan doesn't have one…
- Often a natural way to express a problem.
- Iteration might be better, because of the overhead of function calls and extra storage.
- It's not always easy to convert recursion into iteration.

Recursion (cont.)

- Factorials are easy: n! = n(n-1)(n-2) · · x3x2x1

```
long factorial( long number) {
    if (number <= 1)    // base case
        return 1;
    else
        return number * factorial(number - 1);
}
```

Deitel & Deitel's Illustration



Variable-Length Argument Lists

```
//pseudo code
class A { int i; }
public class VarArgs {
    static void f(Object[] x) {
        for (int i = 0; i < x.length; i++)
            System.out.println(x[i]);
    }
    public static void main(String[] args) {
         f(new Object[] {
            new Integer(47), new VarArgs(),
            new Float(3.14), new Double(11.11) } );
        f(new Object[] {"one", "two", "three" }) ;
        f(new Object[] {new A(), new A(), new A() } );
    }
```

Variable-Length Argument Lists

- This prints

  47

  VarArgs@fee6172e

  3.14

  11.11

  one

  two

  three

  A@fee61874

  A@fee61873

  A@fee6186a