Constructors/Destructors in Java

Prof. Chris Jermaine
cmj4@cs.rice.edu

Constructors/Destructors in Java

- All about allocating resources before an object is used
- And freeing resources when an object is done
- Will cover constructors first
  - Which are done right in Java and make a lot of sense
- And then cover destructors
  - Which are not and don’t

Constructors

- We’ve seen ‘em
  - — Code that gets automatically called when memory for an object is allocated
- But we’ll discuss in a bit more detail

Default Values

- At object allocation
  - — Java assigns each member var its default value
- If you want, you can give an explicit initialization

```
class Foo {
    private int a = 12;
    private IDoubleVector b = new DenseDoubVector (2, 0);
    private double c;
    ...
}
```

- Initialized in order of declaration
  - — And initialized before any constructor code is executed
  - — Note: statics only initialized once, at first creation of object of that type

Providing Initialization Code

- Can have a block of code that is always run before constructor

```
class Foo {
   int a;
   {
      System.out.println (“This code’ll run before the”);
      System.out.println (“constructor.”);
   }
   Foo () {
      System.out.println (“Here is the constructor.”);
   }
}
...
Foo bar = new Foo ();
```

- What does this do?
- Note: can have block labeled “static”... what happens then?

Calling Superclass Constructors

- The default (no-param) superclass const. is automatically called
  - Invoked before anything else is done to the subclass
  - This can cause a chain of invocations, all the way back to “Object”
  - If you want another constructor, use call to “super”
  - Must be the first statement in a named constructor

```
class Foo extends Bar {
    {
        System.out.println (“Hi mom!”);
    }
    Foo () {
        super (2);
        System.out.println (“Here is the constructor.”);
    }
} ...
Foo bar = new Foo ();
```

  - What does this do?

I Think Java Does This Just About Right

- Except that there’s no really easy way to force a subclass...
  - — To call a particular, parameterized constructor
- Ex:

```
class AChecker {
   private int xPos;
   private int yPos;
   protected AChecker (int initX, int initY) {}
}

class BlackChecker extends AChecker {
   public BlackChecker () {
      System.out.println (“I screwed, up, why?”);
   }
}
```

— What’s the best thing you can do here to prevent problems?

Destructors

- A “destructor” is a piece of code called when an object dies
- One weird thing about Java
  - It lacks destructor support in the language
  - Does have “finalize()” inherited from “Object”, but that’s something else
  - I couldn’t believe this when I first learned Java...
- Why did the Java leave out destructors?

Destructors

- A “destructor” is a piece of code called when an object dies
- One weird thing about Java
  - — It lacks destructor support in the language
  - — Does have “finalize()” inherited from “Object”, but that’s something else
  - — I couldn’t believe this when I first learned Java...
- Why did the Java leave out destructors?
  - — Presumably, they thought, “Java’s garbage collected”...
  - — Destructors are for writing code that frees memory when an object is dead
  - — So we don’t need them!
- What’s wrong with this argument?

Destructors

- A “destructor” is a piece of code called when an object dies
- One weird thing about Java
  - — It lacks destructor support in the language
  - — Does have “finalize()” inherited from “Object”, but that’s something else
  - — I couldn’t believe this when I first learned Java...
- Why did the Java leave out destructors?
  - — Presumably, they thought, “Java’s garbage collected”...
  - — Destructors are for writing code that frees ~~memory~~ **Resources** when an object is dead
  - — So we don’t need them! **Not true!**
- What’s wrong with this argument?
  - — Local memory is not the only resource! What are some others?

Destructors

- A “destructor” is a piece of code called when an object dies
- One weird thing about Java
  - It lacks
  - Does have “finalize()” inherited from “Object”, but that’s something else
  - I couldn’t believe this when Scott first told me...
- Why did the Java leave out destructors?
  - Presumably, they thought, “Java’s garbage collected”...
  - Destructors are for writing code that frees ~~memory~~ **Resources** when an object is dead
  - So we don’t need them! **Not true!**
- What’s wrong with this argument?
  - Local memory is not the only resource! What are some others?
  - Secondary storage, server connection, network connection, device driver memory

So What Does a Good Programmer Do?

- Lack of a destructor is an attack on encapsulation!
- Why? You suddenly have to worry...
    - — Does this class have to deal with freeing some resource?
    - — When that resource might be unimportant to the class’ interface!
    - — If so, make sure to call “freeResource” routine
    - — Example of commonly suggested workaround:

```
NetworkConnection temp;
try {
     .... // code that deals with temp here
} finally {
     temp.freeResource ();
}
```

- — Java 7 even has a “try-with-resources” shortcut for this

But Even This Is Not Very Good

```
NetworkConnection temp = new NetworkConnection ();
try {
     .... // code that deals with temp here
} finally {
     temp.freeResource ();
}
```

- What’s the problem?

But Even This Is Not Very Good

```
NetworkConnection temp = new NetworkConnection ();
try {
     .... // code that deals with temp here
} finally {
     temp.freeResource ();
}
```

- What’s the problem?
  - — What if inside the “try” code you create a reference to “temp”?
  - — No easy way to deal with this, except to be careful, and never alias a resource
  - — OR could eschew aliases entirely (but that’s not the “Java way of programming”)
- It’s easy for me to stand here and be snarky
  - — And complain about Java (why not just admit failure, add a real destructor?)
  - — But this is a very serious issue
  - — Only way around it is being as careful as possible with resources!

Final Note

- What’s up with “Object.finalize ()”?
  - — Called by garbage collector “if and when” JVM determines no more references
- After call to “finalize”, JVM can discard the object
- But no guarantees. This code never prints “done” on my machine:

```
class Bar {
    Bar () {
        System.out.println (“new Bar!”);}
    protected void finalize () {
        System.out.println (“done”);}
}
...
public static main (String [] args) {
    System.out.println (“Inside main.”);
    Bar x = new Bar ();
}
```