

Programming Paradigms

159.272

Object Lifecycle

Amjed Tahir
a.tahir@massey.ac.nz

Original author: Jens Dietrich

Readings

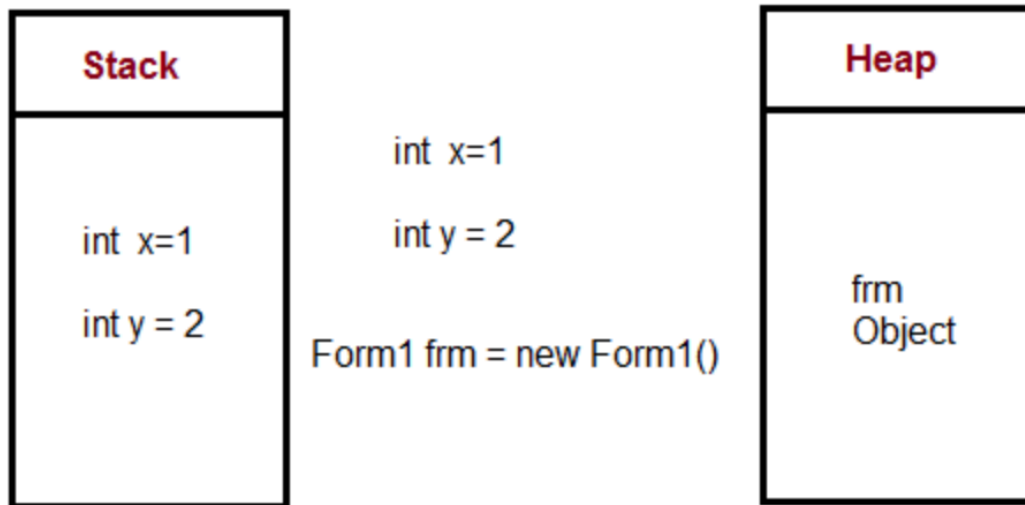
1. Java Tutorial, trail: Learning the Java Language
<http://docs.oracle.com/javase/tutorial/java/index.html>

Overview

- java memory architecture
- constructors
- referencing other constructors
- objects as graphs
- garbage collection
- out of memory and stack overflow errors
- finalize
- weak and soft references
- class lifecycle
- class loaders
- static blocks
- making objects persistent
- cloning objects

Java Memory Architecture

- Java allocates two types of memory: **heap** and **stack**
- **Stack** is used for static memory allocations
- **Heap** is used for dynamic memory allocations.
- objects are stored in the **heap**
- method invocations are stored in the **stack**



Java Memory Architecture

- once a method returns, the particular section of the stack is cleared
- the size of both stack and heap size can be customised
- for instance, starting Java with "java -Xmx2g" will set the maximum heap size to 2 GB

Eclipse init file with Xms and Xmx values

```
eclipse.ini — ~/eclipse/java-neon/Eclipse.app/Contents/Eclipse
eclipse.ini
--
--launcher.appendVmargs
13 -vm
14 /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/jre/bin
15 -vmargs
16 -Dosgi.requiredJavaVersion=1.8
17 -XX:+UseG1GC
18 -XX:+UseStringDeduplication
19 -XstartOnFirstThread
20 -Dorg.eclipse.swt.internal.carbon.smallFonts
21 -Dosgi.requiredJavaVersion=1.8
22 -Xms512m
23 -Xmx4096m
24 -XX:PermSize=1024m
25 -XX:MaxPermSize=4096m
26 -Xdock:icon=../Resources/Eclipse.icns
27 -XstartOnFirstThread
28 -Dorg.eclipse.swt.internal.carbon.smallFonts
29 -Declipse.p2.max.threads=10
30 -Doomph.update.url=http://download.eclipse.org/oomph/updates/milestone/latest
31 -Doomph.redirection.index.redirection=index:/->http://git.eclipse.org/c/oomph/org.e
32
33
```

Constructors

- objects are created using **constructors**
- constructors have no explicit return type
- the name of the constructor **MUST BE** the name of the class it instantiates
- constructors can have parameters and exceptions
- there can be multiple constructors, however, their signature (parameter types) must differ
- this is called **constructor overloading**
- if no constructor is defined, the compiler will assert that there is an **implicit constructor** copied from the super class (usually this means no parameters)

```
Class Student{  
  
    Public Student (){}  
}
```

Constructors ctd

- constructors are usually public, but can also have private, protected or package visibility (encapsulation - to be discussed later)
- the main role of constructors is to initialise the state (instance variables / properties) of an object

Constructor Example

```
public class Student {  
    private String name = null;  
    private String firstName = null;  
    public Student () {  
        this.name = "";  
        this.firstName = "";  
    }  
    public Student (String n,String fn) {  
        this.name = n;  
        this.firstName = fn;  
    }  
}
```

in the constructor, the instance variables are initialised

this refers to the object that is being constructed

there can be multiple constructors with different sets of parameter (types) (aka different signatures)

Referencing Other Constructors

- other constructors within the class can be invoked using the following syntax: `this (..)`
- this supports good programming style: duplication (copy and paste) can be avoided (more on good programming in 159251)
- classes have **superclasses** and **inherit state** (instance variables) from superclasses
- if the superclass is not explicitly defined, `java.lang.Object` is the implicit superclass
- this means that the inherited state should be initialised by constructors in the superclass
- these constructors can be referenced with `super (..)`
- the invocation of `super (..)` must be the first statement in the constructor

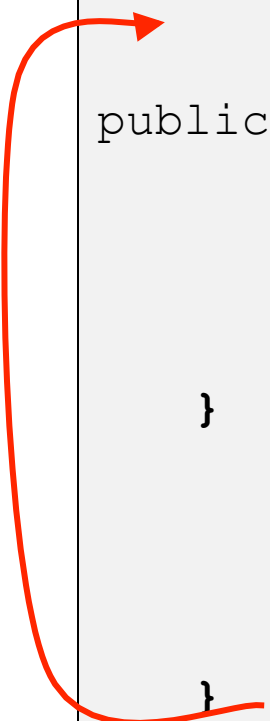
Referencing this() Example

```
public class Student {  
    private String name = null;  
    private String firstName = null;  
    public Student () {  
        this("", "");  
    }  
    public Student (String n, String fn) {  
        this.name = n;  
        this.firstName = fn;  
    }  
}
```

this invokes the second constructor with two empty strings as parameters

Referencing super() Example

```
public class Object {  
    public Object () {...}  
    ...  
public class Student {  
    private String name = null;  
    private String firstName = null;  
    public Student () {  
        this("", "");  
    }  
  
    public Student (String n,String fn) {  
        super();  
        this.name = n;  
        this.firstName = fn;  
    }  
}
```



invokes the constructor from the (implicit)
super class java.lang.Object

Freeing Memory

- Java does not have **destructors** (unlike C++)
In C++, data is freed with `delete`, `delete[]` or `free`
- the JVM automatically frees memory using a so-called **garbage collector (GC)** for short)
- this is mostly **transparent** for the programmer (i.e., the programmer does not have to worry about this)
- we explore the GC using some experiments

Monitoring the JVM

- the JVM has a great tool that can be used to monitor running applications: the **VisualVM**
- we will use VisualVM to check how much heap space is used by our Java program, and screen capture the respective charts
- to run VisualVM, start `jvisualvm` in the bin folder of the JDK

In windows: e.g., `C:\Program Files\Java\jdk1.7.0\bin\jvisualvm.exe`

in Mac e.g., `users/Library/Frameworks/JavaVm.Frameworks/version \current\command\jvisualvm`

From Command/Terminal: use `jvisualvm`

- the VisualVM documentation can be found here:

<http://docs.oracle.com/javase/6/docs/technotes/guides/visualvm/index.html>

The Object Graph

- the GC works on an object graph
- in this graph, objects are nodes, and references to other objects are edges
- reference:
 - via fields of objects
 - If object A had field F with value (object) B, then A references B

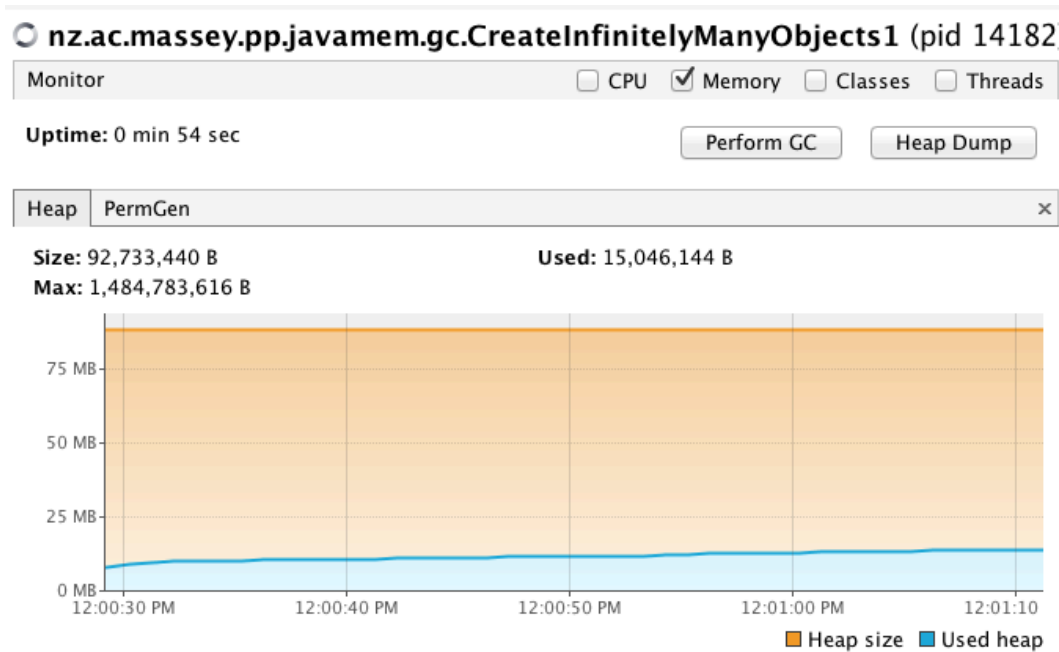
Creating Infinitely Many Objects 1

```
public static void main(String[] args) throws Exception {  
    while (true) {  
        for (int i=0;i<1000000;i++) {  
            Object obj = new Object();  
        }  
  
        // break 100 ms between, allows JVM to clean up (GC !)  
        Thread.sleep(100);  
    }  
}
```

this keeps creating objects forever

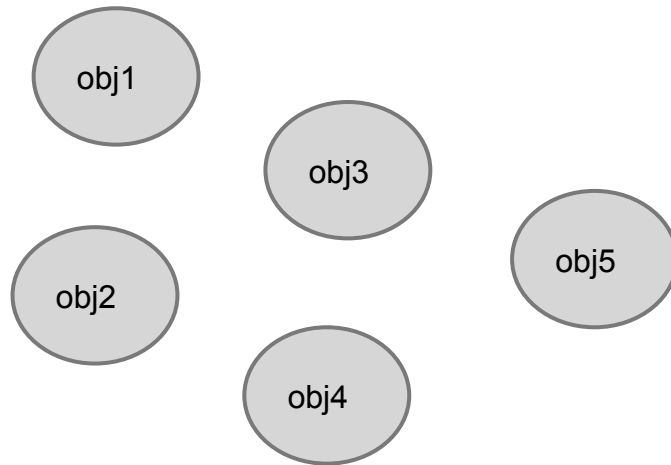
Note that this project will never terminate - you have to kill it manually (e.g., by pressing the stop button in Eclipse)

Creating Infinitely Many Objects 1 ctd



- heap usage stays flat
- the GC can easily remove objects quickly as they are not referenced (and *if nobody knows them, nobody can use them: they become garbage*)

Creating Infinitely Many Objects 1 ctd



- the object graph only consists of disconnected objects
- i.e., there are no edges (references between the objects created)

Creating Infinitely Many Objects 2

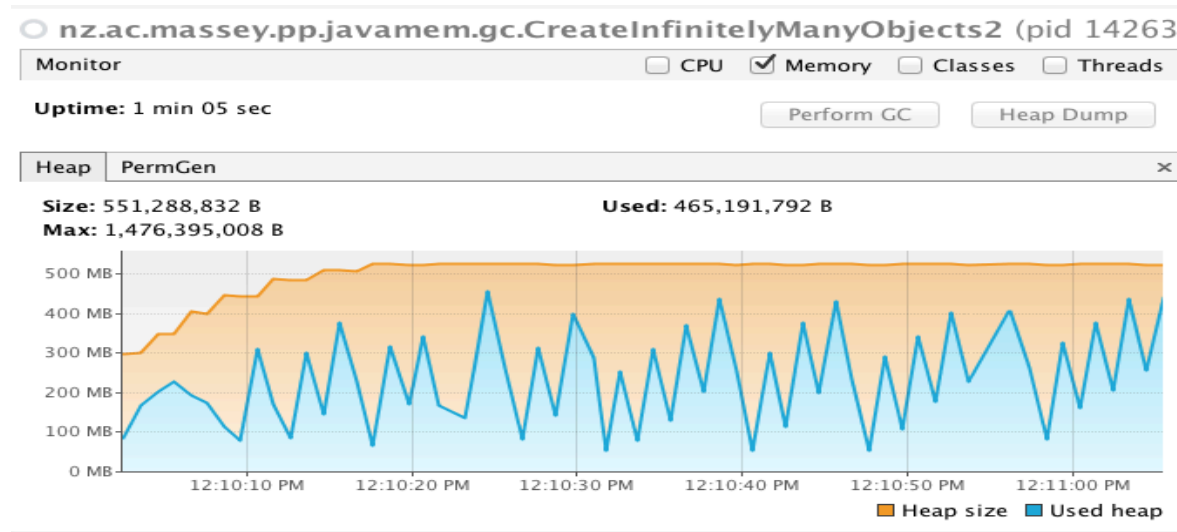
```
import java.util.*;
...
public static void main(String[] args) throws Exception {
    while (true) {
        List list = new ArrayList();
        for (int i=0;i<1000000;i++) {
            list.add(new Object());
        }

        // break between, allows JVM to clean up (GC
        Thread.sleep(100);
    }
}
```

in each iteration,
a new list is
created, the
reference to the
list is lost!

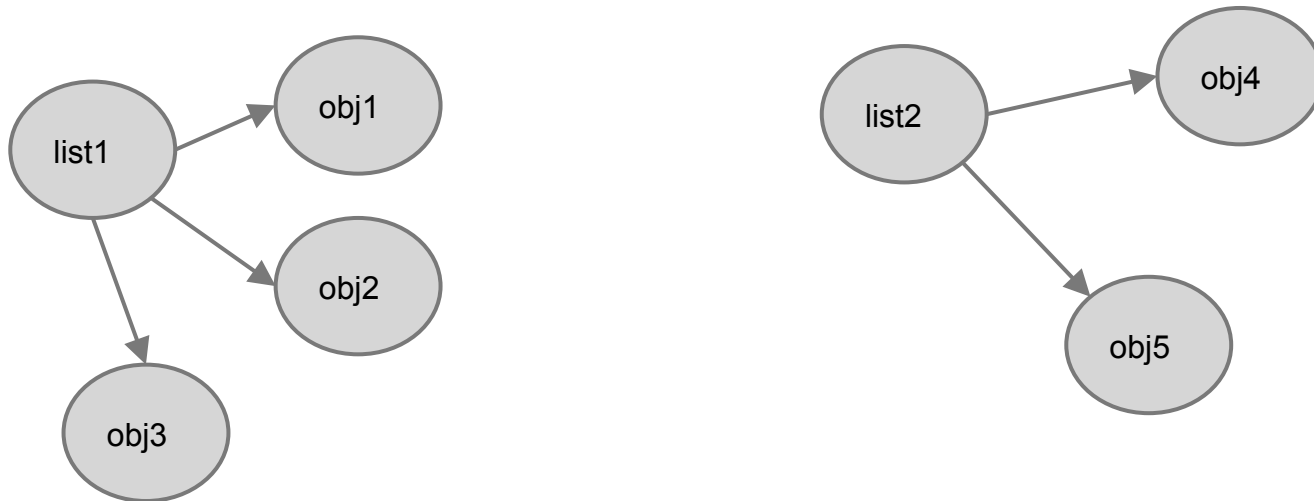
now the objects are actually
referenced (i.e., can still be
used) by a list object

Creating Infinitely Many Objects 2 ctd



- the application is still not running out of memory
- the reason is that the lists referencing the objects are themselves not referenced (i.e., they are garbage) - the reference to a list is lost when the block has been executed

Creating Infinitely Many Objects 2 ctd



- the object graph now has edges as well: the lists reference their elements
- however, the lists are not referenced themselves
- this means that indirectly, all objects in the list are garbage
- i.e., the GC works **recursively**
- it is apparent that this is much more difficult: the GC works in stages (first removing lists, then elements), and the heap allocation chart shows this

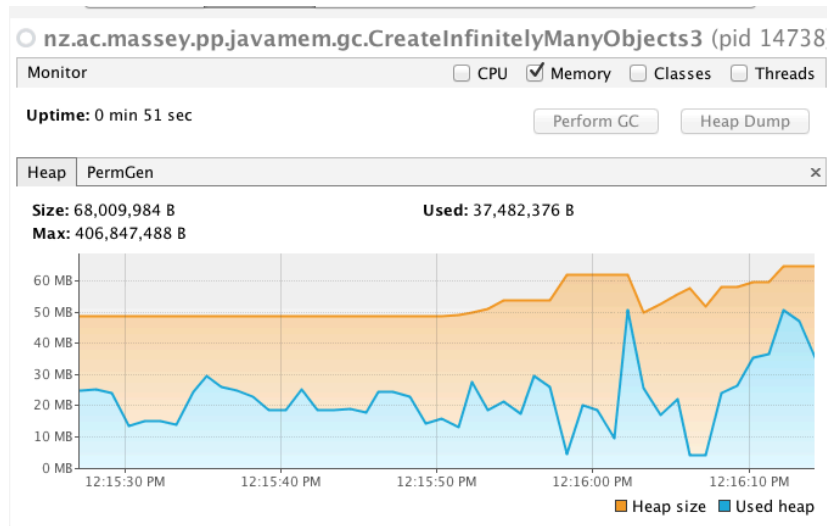
Creating Infinitely Many Objects 3

```
public class ObjectInList {  
    public List container = null;  
}  
...  
public static void main(String[] args) throws Exception  
{  
    while (true) {  
        List list = new ArrayList();  
        for (int i=0;i<1000000;i++) {  
            ObjectInList nextObject = new ObjectInList();  
            nextObject.container = list;  
            list.add(nextObject);  
        }  
        Thread.sleep(100);  
    }  
}
```

now not only the lists reference the objects, but the objects also reference the respective list

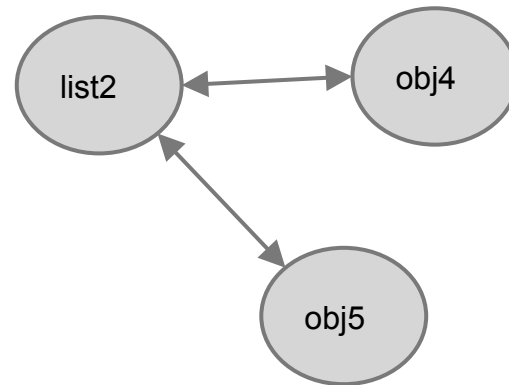
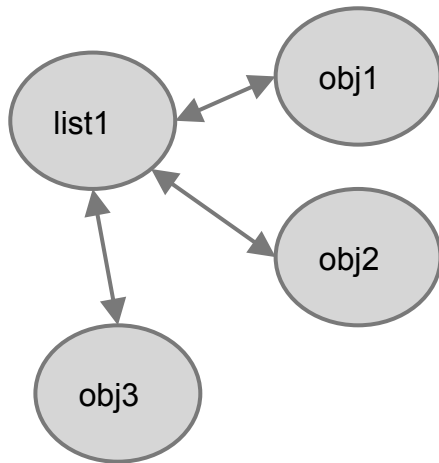
this creates a **cyclic** dependency between objects

Creating Infinitely Many Objects 3 ctd



- surprisingly, this still works
- the GC can detect and handle **cyclic dependencies**, and will correctly remove the lists and all contained objects from memory

Creating Infinitely Many Objects 3 ctd



- the object graph now has cycles: the lists reference their elements, and the elements reference the lists
- however, nobody references these cycles

Creating Infinitely Many Objects 4

```
public static void main(String[] args) throws Exception {
```

```
    List list = new ArrayList();
```

```
    while (true) {
```

```
        for (int i=0;i<1000000;i++) {
```

```
            list.add(new Object());
```

```
        }
```

```
        // break between, allows JVM to clean up (GC !)
```

```
        Thread.sleep(100);
```

```
    }
```

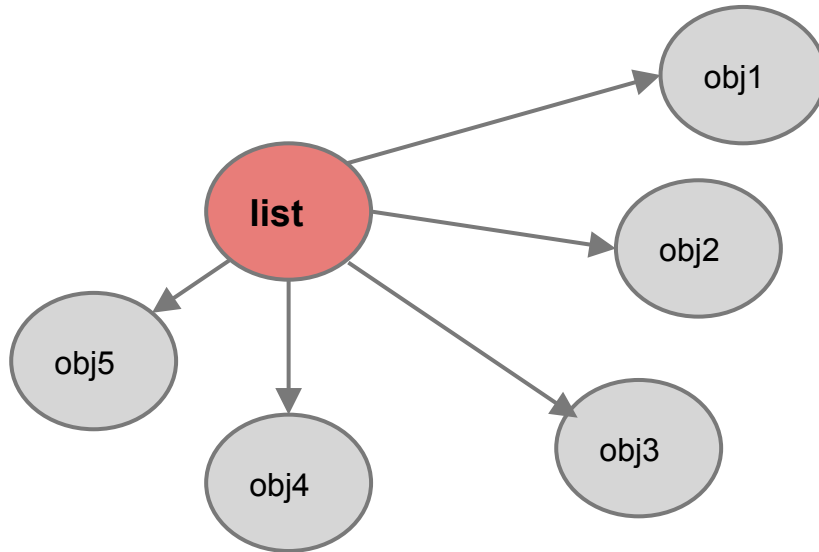
```
}
```

now there is only one list that is referenced within this method (in the stack!) - this will grow to an extent that forces the application out of memory

Creating Infinitely Many Objects 4 ctd

- this program runs out of memory
- i.e., the JVM terminates (crashes) with an **OutOfMemoryError**
- from the GC point of view, the one list created is still being used, and therefore none of the elements of the list can be garbage collected

Creating Infinitely Many Objects 4 ctd



- now every object created is referenced by one list
- note that this list is defined outside the main (while) loop
- i.e., objects created in all iterations will go into the same list !

StackOverflowErrors

```
public class TriggerStackOverflowError {  
    public static void main(String[] args) {  
        foo();  
    }  
    static void foo() {  
        foo();  
    }  
}
```

- there is a second type of "out of memory error" in Java - stack overflows
- whenever a method is invoked, information is added to the stack
- in particular when recursion is used without termination condition, the JVM runs out of stack space and returns with a **StackOverflowError**

Summary: GC in Java

- the GC works on an object graph
- the GC looks for nodes (objects) in this graph that are not reachable from static references or references on the stack (values of variables in currently executed methods)
- if such objects are found, they are removed and the respective section of the heap is freed
- the the process is repeated recursively
- the GC can also detect and handle cycles ("strongly connected components")

Class Lifecycle

- classes have a lifecycle as well
- if a class is being used for the first time, it is loaded and checked
- use of a class means either:
 - instantiation, or
 - use of a static member (variable or method)
- if the class has a **static block**, then this code is executed before the class is used once
- the static block acts like a "class constructor"

Static Block Example

```
public class MyClass {  
    static {  
        System.out.println("MyClass loaded");  
    }  
    ...  
}
```

Class Loaders

- class loaders are objects responsible for loading classes
- classes can be loaded from files, archives, network resources and memory locations
- if the same class is loaded with different class loaders, the resulting classes are treated as different
- classes can also be unloaded
- this has important applications in enterprise computing
- classloading is outside the scope of this paper, and will be discussed in other papers

Making Objects Persistent

- often it is required to "save objects", i.e. to make objects **persistent** so that they can be restored later
- this means that some objects and all other objects they reference must be saved
- this is called **object serialisation**: encode an object graph to a sequential structure that can be written to the file system
- Java has two solutions for object serialisation on board: binary serialisation and XML-based serialisation

Making Objects Persistent ctd

- serialisation is also used to transfer objects over the network
- technologies based on this are CORBA and RMI (outside the scope of this paper)
- in enterprise applications, objects are often saved by storing them as rows in relational databases
- this is rather complex, and often additional **object-relational mapping (ORM)** software (such as Hibernate) is used

Cloning Objects

- with constructors, objects are created "from scratch"
- another way to instantiate object is by **cloning** (copying) existing objects
- cloning means: creating new objects with the *same* state
- that usually means that the clone has the following properties:
 - it is of the same type as the original
 - the clone and the original are different objects (!=)

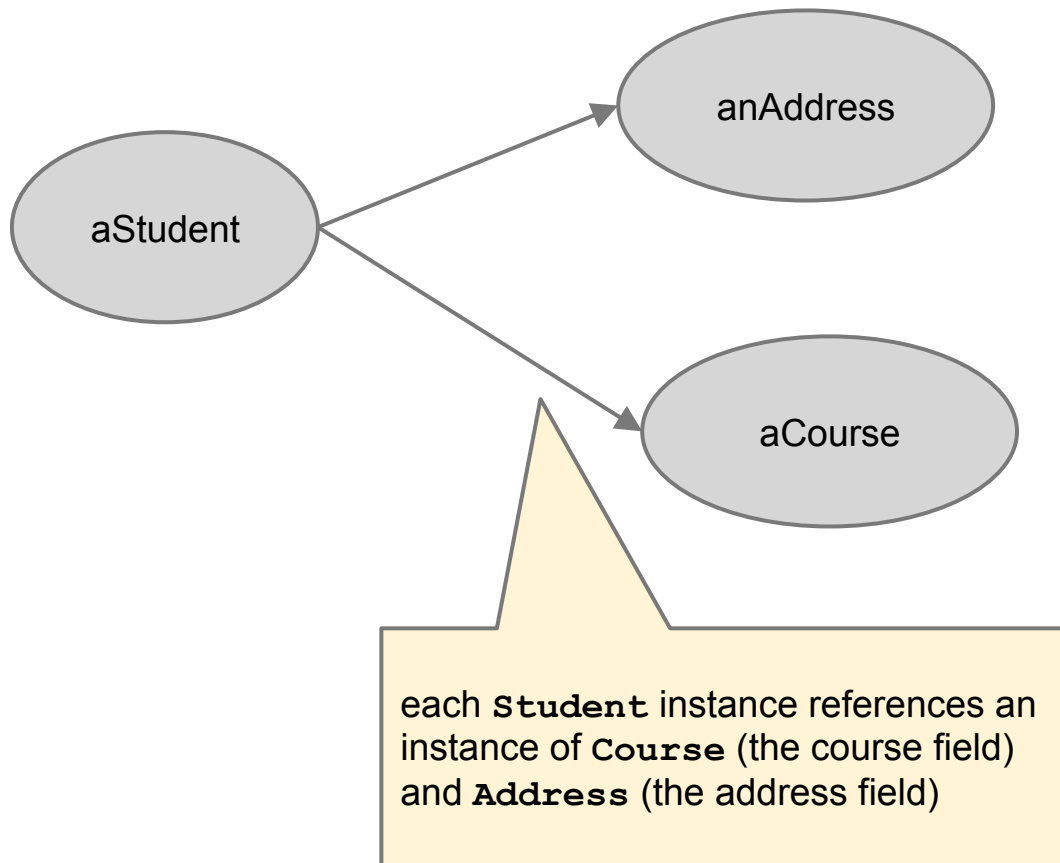
Cloning Objects ctd

- there is no direct built-in low level support for cloning in Java (although serialisation could be used for this)
- cloning is done by implementing a **clone()** method
- often, this is done by implementing a special **Cloneable** interface (interfaces will be discussed later)

Clone Example

```
public class Student {  
    public int id = 0;  
    public Address address = null;  
    public Course course = null;  
}  
public class Course {  
    public int id = 0;  
    public String name = null;  
}  
public class Address {  
    public int no = 0;  
    public String street = null;  
    public String town = null;  
}
```

Student Object Graph



Cloning Course

```
public class Course {  
    public int id = 0;  
    public String name = null;  
    public Course clone() {  
        Course clone = new Course();  
        clone.name = this.name;  
        clone.id = this.id;  
        return clone;  
    }  
}
```

create new instance

copy state

- easy to implement
- important: `int` is a value type, `String` is immutable
- similar for `Address`

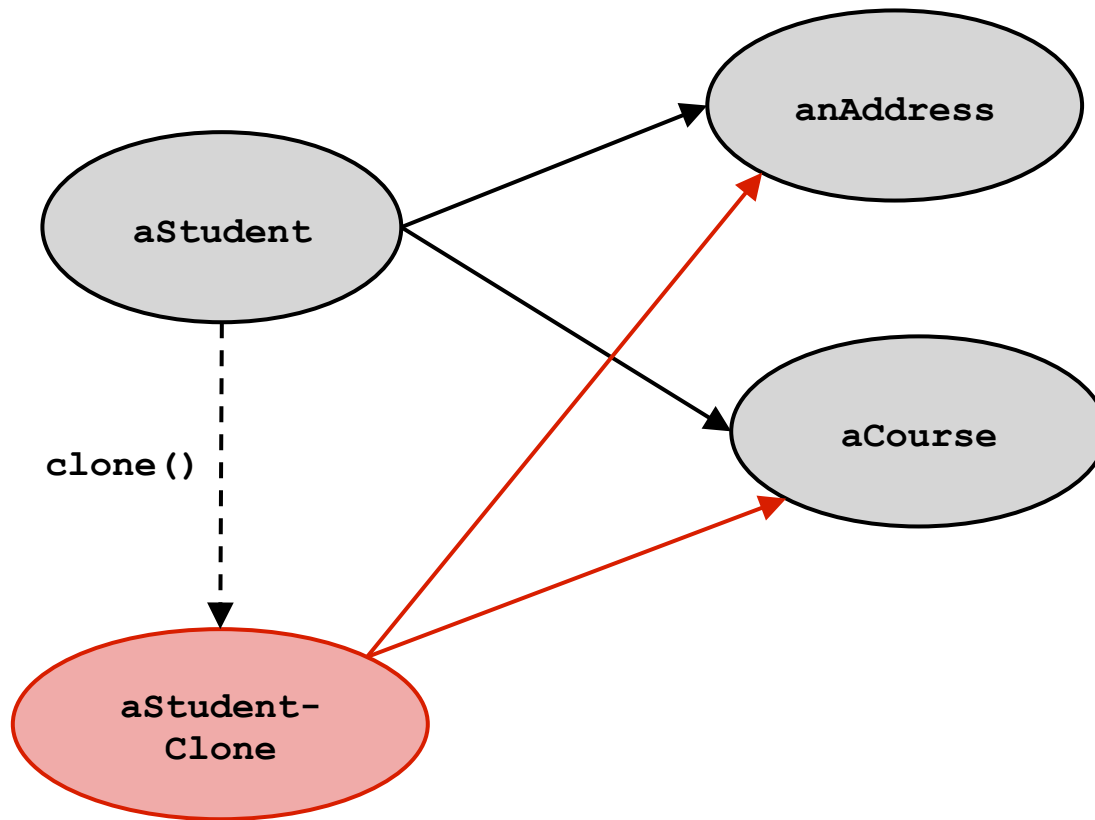
Cloning Student v1 - Shallow Copy

```
public class Student {  
    public int id = 0;  
    public Address address = null;  
    public Course course = null;  
    public Student clone() {  
        Student clone = new Student();  
        clone.id = this.id;  
        clone.address = this.address;  
        clone.course = this.course;  
        return clone;  
    }  
}
```

Cloning Student v1 - Shallow Copy

- the clone references the same course and address !
- consequence: if we change the name of the course of a student, the course of all objects cloned from this student changes as well!
- this is called a **shallow copy**

Shallow Copy Object Graph



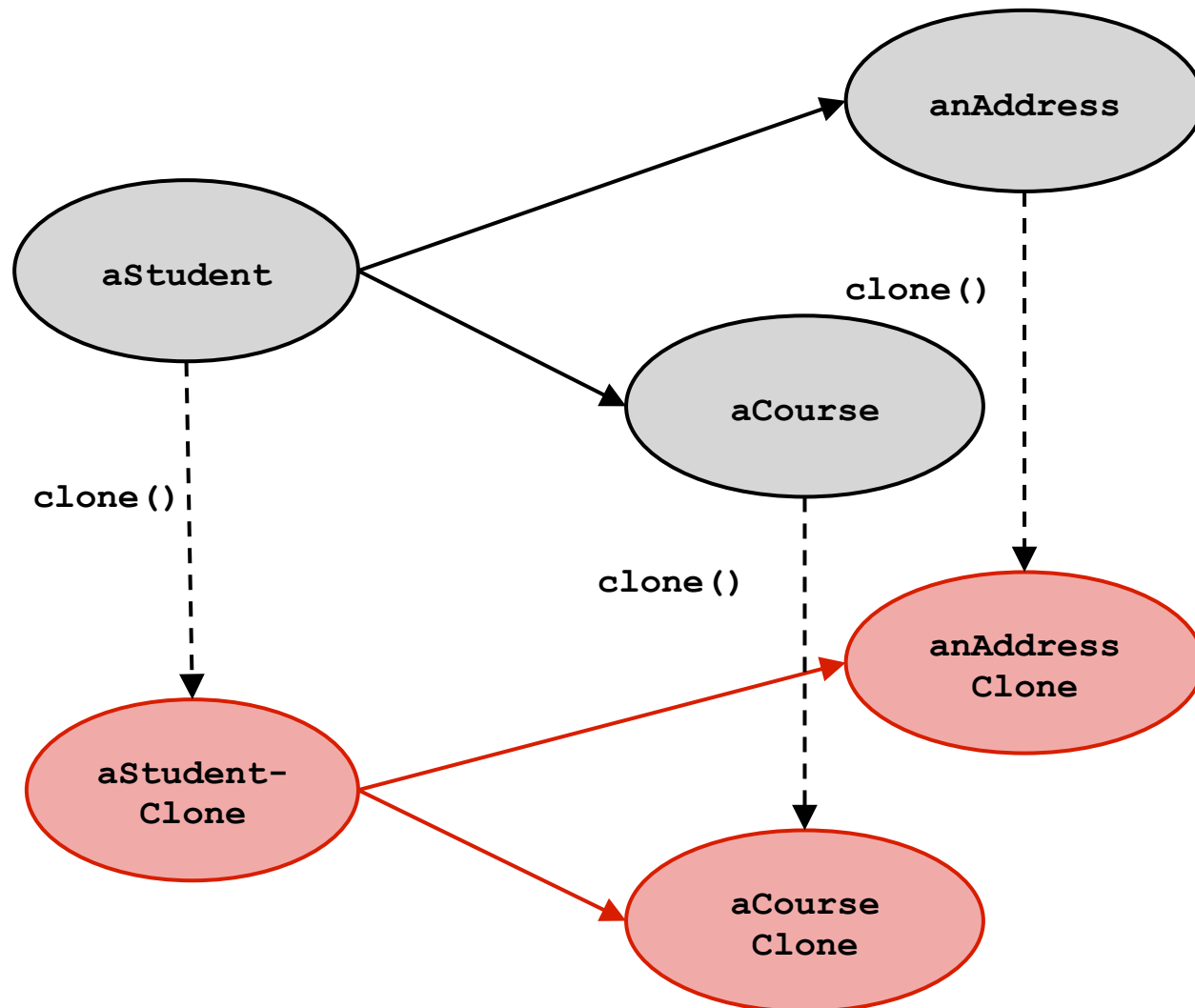
Cloning Student v2 - Deep Copy

```
public class Student {  
    public int id = 0;  
    public Address address = null;  
    public Course course = null;  
    public Student clone() {  
        Student clone = new Student();  
        clone.id = this.id;  
        clone.address = this.address.clone();  
        clone.course = this.course.clone();  
        return clone;  
    }  
}
```

Cloning Student v2 - Deep Copy

- clone references copies of the course and address !
i.e., cloning is done **recursively**
- consequence: if we change the name of the course of a student, the course of all objects cloned from this student does not change!
- this is called a **deep copy**

Deep Copy Object Graph



Cloning Student v3 - Sheep Copy

```
public class Student {  
    public int id = 0;  
    public Address address = null;  
    public Course course = null;  
    public Student clone() {  
        Student clone = new Student();  
        clone.id = this.id;  
        clone.address = this.address.clone();  
        clone.course = this.course;  
        return clone;  
    }  
}
```

Cloning Student v3 - Sheep Copy

- often a mix of shallow and deep copy is needed: some fields are cloned recursively, others not
- this depends on whether the object just knows another object (association), or owns it (aggregation). E.g. a spouse field requires a deep copy - this is ownership
- **shallow + deep = sheep**

Sheep Copy Object Graph

