COMP 250

Lecture 34

Polymorphism (continued.)

Garbage Collection
(mark and sweep)

Nov. 27, 2017
Recall last lecture

class Dog
String serialNumber
Person owner
void bark()
{print "woof"}
:
extends

class Beagle
void hunt ()
void bark()
{print "aowwwuuu"}
Recall last lecture

```java
Dog myDog = new Beagle();
myDog.bark();
```

```java
class Dog
  String serialNumber
  Person owner
  void bark()
  {print "woof"}
:
extends

class Beagle
  void hunt ()
  void bark()
  {print "aowwwuuu"}
```
This figure shows objects in a running Java program.

```java
Dog myDog = new Beagle();
myDog.bark();
```
Suppose we are running a class `TestDog`, which has a `main()` method.
Suppose we are running a class `TestDog`, which has a `main()` method.

There are no objects at the start of execution.
public static void main()
{
    Dog myDog = new Beagle();
    myDog.bark();
}

TestDog.main()

TestDog

null

Call Stack

Objects

Beagle

Dog

Object
public static void main()
{
    Dog myDog = new Beagle();
    myDog.bark();
}

Call Stack
(Beagle constructor called)
public static void main(){
    Dog myDog = new Beagle();
    myDog.bark();
}

Call Stack

Objects

Dogs

Beagle

TestDog

TestDog.main()

Dog myDog

TestDog

class descriptor

Dog

class descriptor

Beagle

class descriptor

Object
class descriptor

Beagle object
public static void main()
{
    Dog myDog = new Beagle();
    myDog.bark();
}

JVM looks for the bark() method in this.getClass() and finds it.
public static void main() {
    Dog myDog = new Beagle();
    myDog.bark();
    myDog.getOwner();
}

JVM looks for the getOwner() method in this.getClass() and doesn’t find it.
public static void main()
{
    Dog myDog = new Beagle();
    myDog.bark();
    myDog.getOwner();
}

JVM then looks for the getOwner() method in this.getClass().getSuperclass() and finds it.

JVM looks for the getOwner() method in this.getClass() and doesn’t find it.
COMP 250

Lecture 34

Polymorphism (continued.)

Garbage Collection
(mark and sweep)

Nov. 27, 2017
Garbage Collection

Dog  myDog = new Beagle(“Bob”);

myDog = new Terrier(“Tim”);

Nothing references the Bob the Beagle.

Bob is wasting memory.  Bob has become garbage.
Dog  myDog = new Beagle("Bob");

myDog = new Terrier("Tim");
Dog    myDog = new Beagle(“Bob”);

myDog = new Terrier(“Tim”);

Call Stack

Terrier       object   “Tim”
  class descriptor

Beagle        object   “Bob”
  class descriptor

Test          class descriptor

Test.main()

Dog    myDog

other class descriptors

Bob is garbage.
As the program continues, more “garbage” accumulates.
Let’s ignore the call descriptors for rest of today.

Objects

other objects

Terrier

Terrier object “Tim”

Beagle

Beagle object “Bob”

other objects

other objects

Call Stack

Test

class descriptor

Test

class descriptor

other class descriptors

other class descriptors

mA()

mB()

mC()

Test.main()

Dog myDog

other objects
Every object has a location in memory: `Object.hashCode()`.

Call Stack:

```
main()
mA()
mB()
mC()

```

Objects:

```
Terrier object "Tim"
Beagle object "Bob"
object
object
object
another garbage object
object

```
The Java Virtual Machine (JVM) maintains a linked list of all objects. i.e. The list stores the `Object.hashCode()` of each object.
Q: What to do when object space fills up?

A: Let the program crash.

A: Reuse the space we don’t need. (Garbage collection)
"Live objects" (not garbage) are those referenced either from a call stack variable or from an instance variable in a live object.

Call Stack

Objects

mC()
mB()
mA()
main()

Terrier object "Tim"

Beagle object "Bob"

object

object

object

object

another garbage object

object
Q: If these objects are only referenced by each other, then are they garbage?
A: Yes, because they will never be used by the program.
Garbage collection: “Mark and Sweep”

1) Build a graph, and identify live objects (“Mark”)

2) Remove garbage (“sweep”)
Garbage collector builds a graph that corresponds to the one here:

Vertices correspond to reference variables in call stack, and to objects. Edges correspond to references.

Call Stack

Objects

- mA()
- mB()
- mC()
- main()
For each vertex that corresponds to a reference variable on the call stack: traverse the graph.

Visiting a node means *mark* it as live.
Phase 1: “Mark” the garbage

Call Stack

Objects

- main()
- mA()
- mB()
- mC()

Terrier object “Tim”

Beagle object “Bob”

- another garbage object
- object
- object
- object
Phase 2: “Sweep” the garbage

Call Stack

Objects

mA()  

mB()  

mC()  

main()  

Terrier object “Tim”

- object
- object
- object
- object

remove node from list
remove node From list
Use another list to keep track of free space between objects.
Call Stack:

1. main()
2. mA()
3. mB()
4. mC()

Terrier object "Tim"

- new object

- new node in object list
Two lists: **free space, live objects**
After garbage collection, continue execution..

- New objects can be added, where there is a big enough gap in free space.

- Garbage collection is needed again when there is no gap big enough for the new object.

- Program needs to stop (temporarily) to do garbage collection. This is not good for real time applications.