Chapter 13.

Life and Death of an Object

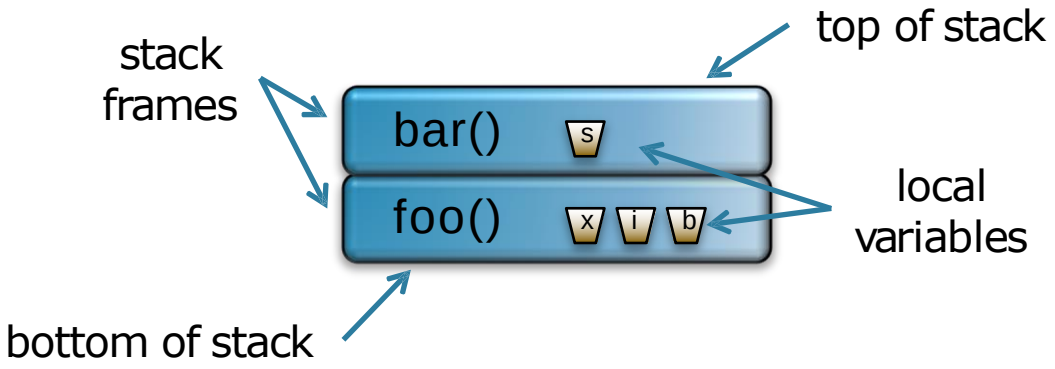
2020-2021
COMP2396 Object-Oriented Programming and Java
Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)
Department of Computer Science, The University of Hong Kong

The Stack and the Heap

- In Java, we care about 2 areas of memory, namely the stack and the heap
- The stack
  - Memory set aside as a scratch space for a thread of execution
  - Used to store method invocations and local variables
  - Last-In-First-Out (LIFO)
- The Heap
  - Memory set aside for dynamic allocation
  - Also known as the garbage-collectible heap
  - Where all objects live

Methods are Stacked

- When a method is called, a new stack frame is created and pushed onto the top of the stack
- A stack frame holds the state of the method, including which line of code is executing and the values of all local variables
- The method at the top of the stack is always the currently running method for that stack
- A method stays on the stack until it hits its closing curly brace
- Example
  - If foo() calls bar(), bar() is stacked on the top of foo()



A Stack Scenario

Example

```
public void doStuff() {
   boolean b = true;
   go(4);
}

public void go(int x) {
   int z = x + 24;
   crazy();
   // imagine more code here
}

public void crazy() {
   char c = 'a';
}
```

1. Code from another class calls doStuff(), and doStuff() goes into a stack frame at the top of the stack. The boolean variable named 'b' goes on the doStuff() stack frame

A Stack Scenario

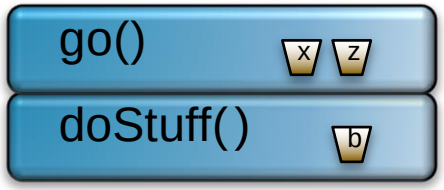
Example

```
public void doStuff() {
   boolean b = true;
   go(4);
}

public void go(int x) {
   int z = x + 24;
   crazy();
   // imagine more code here
}

public void crazy() {
   char c = 'a';
}
```

2. doStuff() calls go(), go() is pushed on top of the stack. Variables 'x' and 'z' are on the go() stack frame



A Stack Scenario

Example

```
public void doStuff() {
   boolean b = true;
   go(4);
}

public void go(int x) {
   int z = x + 24;
   crazy();
   // imagine more code here
}

public void crazy() {
   char c = 'a';
}
```

3. go() calls crazy(), crazy() is now on the top of the stack, with variable 'c' on the frame

A Stack Scenario

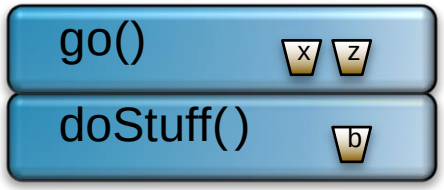
Example

```
public void doStuff() {
   boolean b = true;
   go(4);
}

public void go(int x) {
   int z = x + 24;
   crazy();
   // imagine more code here
}

public void crazy() {
   char c = 'a';
}
```

4 crazy() completes, and its stack frame is popped off the stack. Execution goes back to the go() method, and picks up at the line following the call to crazy()



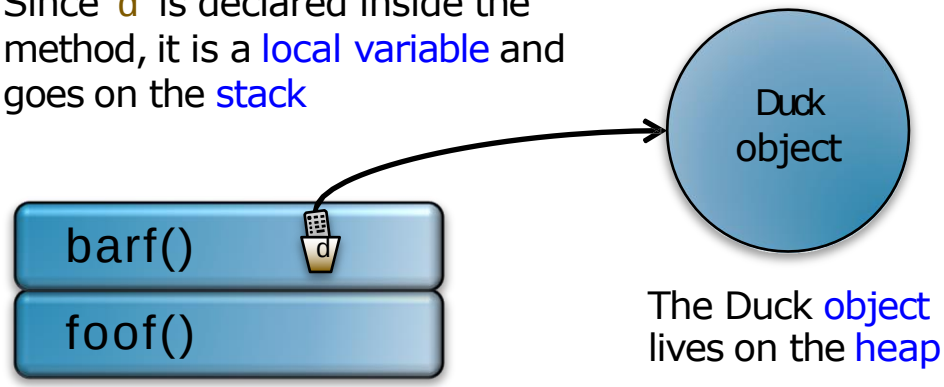
Local Variables for Objects

- Remember, a non-primitive variable holds a reference to an object, but not the object itself
- For a local variable holding a reference to an object, only the variable goes on the stack, the object still lives on the heap
- Example

```
public class StackRef {
   public void foof() {
      barf();
   }

   public void barf() {
      Duck d = new Duck(24);
   }
}
```

barf() declares a reference variable 'd' and creates a new Duck object. Since 'd' is declared inside the method, it is a local variable and goes on the stack



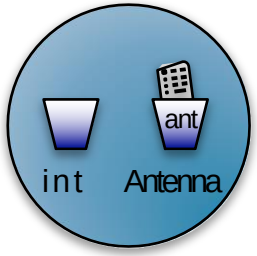
The Duck object lives on the heap

Instance Variables

If local variables live on the stack, where do instance variable live?

- Instance variables live inside the object they belong to, and therefore they also live on the heap
- Example

```
public class CellPhone {
   private int x;
   private Antenna ant;
}
```



CellPhone object

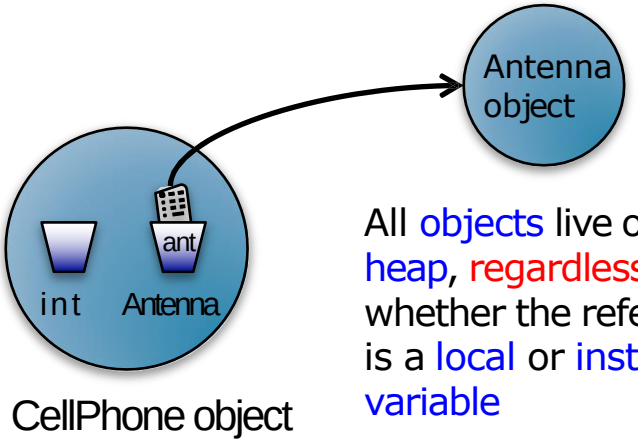
When a CellPhone object is created, its instance variables 'x' and 'ant' live inside the object on the heap

Instance Variables

If local variables live on the stack, where do instance variable live?

- Instance variables live inside the object they belong to, and therefore they also live on the heap
- Example

```
public class CellPhone {
   private int x;
   private Antenna ant = new Antenna();
}
```

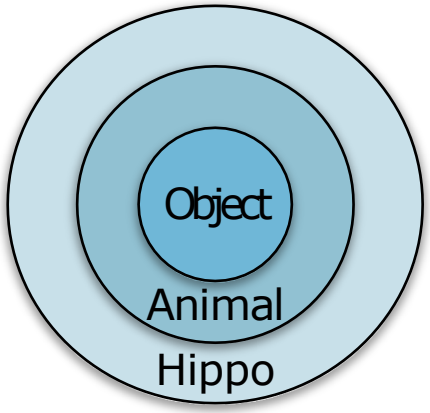


All objects live on the heap, regardless of whether the reference is a local or instance variable

When a CellPhone object is created, its instance variables 'x' and 'ant' live inside the object on the heap

The Role of Superclass Constructors

- Recall that every object holds not just its own declared instance variables, but also everything from its superclasses
- Conceptually, an object can be thought of as having layers of itself representing each superclass and its own class
- Note that private instance variables in a superclass, though not inherited by a subclass object, also forms part of the superclass part of the object
- A subclass object might inherit methods that depend on private instance variables of the superclass!



The Role of Superclass Constructors

- When an object is being created, all the constructors (including those of the abstract classes) up the inheritance tree must run to build out both the superclass parts and its own class part of the object
- When a constructor runs, it immediately calls its superclass constructor, which in turn immediately calls its superclass constructor…
- This calling of superclass constructor will go all the way up the hierarchy until the Object class constructor is reached
- This process is known as constructor chaining

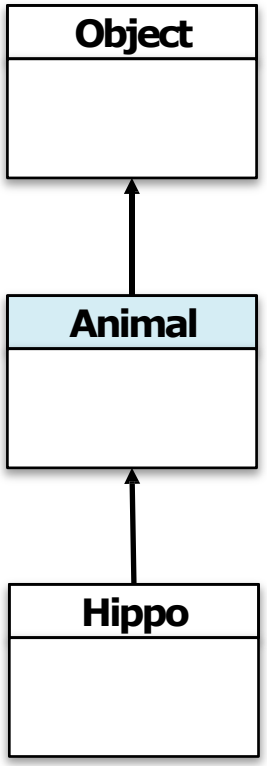
Constructor Chaining

Example

```
public abstract class Animal {
   public Animal() {
      System.out.println("Making an Animal");
   }
}
```

```
public class Hippo extends Animal {
   public Hippo() {
      System.out.println("Making a Hippo");
   }
}
```

```
public class TestHippo {
   public static void main(String[] args) {
      System.out.println("Starting...");
      Hippo h = new Hippo();
   }
}
```



Constructor Chaining

Example

```
public abstract class Animal {
   public Animal() {
      System.out.println("Making an Animal");
   }
}
```

```
public class Hippo extends Animal {
   public Hippo() {
      System.out.println("Making a Hippo");
   }
}
```

```
public class TestHippo {
   public static void main(String[] args) {
      System.out.println("Starting...");
      Hippo h = new Hippo();
   }
}
```

1. The main() in the TestHippo class says newHippo() and the Hippo() constructor goes into a stack frame at the top of the stack

Constructor Chaining

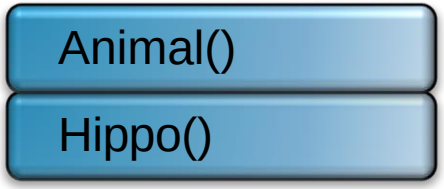
Example

```
public abstract class Animal {
   public Animal() {
      System.out.println("Making an Animal");
   }
}
```

```
public class Hippo extends Animal {
   public Hippo() {
      System.out.println("Making a Hippo");
   }
}
```

```
public class TestHippo {
   public static void main(String[] args) {
      System.out.println("Starting...");
      Hippo h = new Hippo();
   }
}
```

2. Hippo() invokes the superclass constructor which pushes the Animal() constructor onto the top of the stack



Constructor Chaining

Example

```
public abstract class Animal {
   public Animal() {
      System.out.println("Making an Animal");
   }
}
```

```
public class Hippo extends Animal {
   public Hippo() {
      System.out.println("Making a Hippo");
   }
}
```

```
public class TestHippo {
   public static void main(String[] args) {
      System.out.println("Starting...");
      Hippo h = new Hippo();
   }
}
```

3. Animal() invokes the superclass constructor which pushes the Object() constructor onto the top of the stack (Object is the superclass of Animal!)

Constructor Chaining

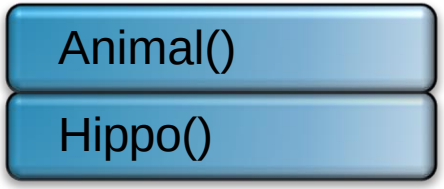
Example

```
public abstract class Animal {
   public Animal() {
      System.out.println("Making an Animal");
   }
}
```

```
public class Hippo extends Animal {
   public Hippo() {
      System.out.println("Making a Hippo");
   }
}
```

```
public class TestHippo {
   public static void main(String[] args) {
      System.out.println("Starting...");
      Hippo h = new Hippo();
   }
}
```

4. Object() completes, and its stack frame is popped off the stack. Execution goes back to the Animal() constructor, and picks up at the line following Animal's call to its superclass constructor



Constructor Chaining

Example

```
public abstract class Animal {
   public Animal() {
      System.out.println("Making an Animal");
   }
}
```

```
public class Hippo extends Animal {
   public Hippo() {
      System.out.println("Making a Hippo");
   }
}
```

```
public class TestHippo {
   public static void main(String[] args) {
      System.out.println("Starting...");
      Hippo h = new Hippo();
   }
}
```

Sample output

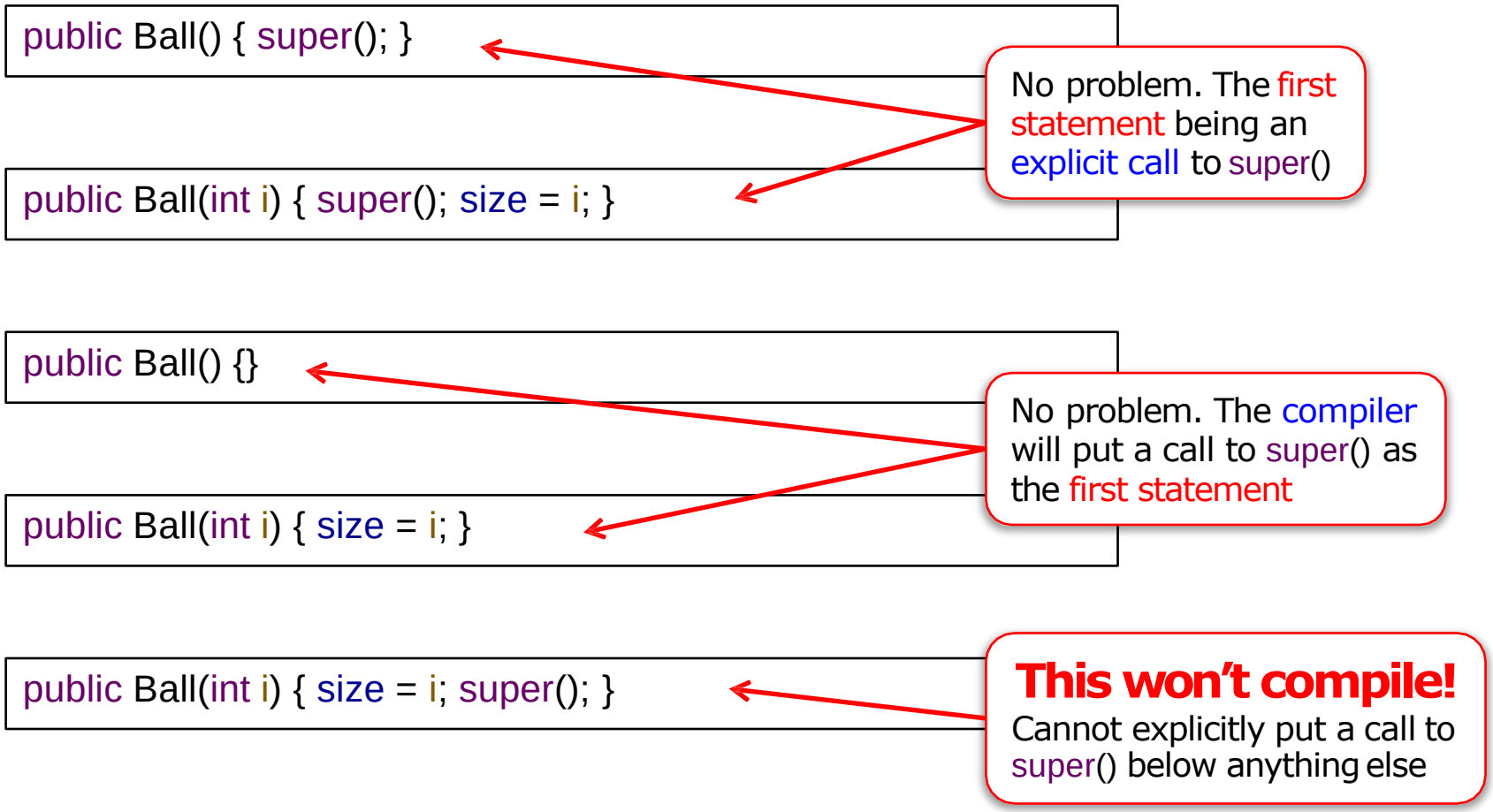
```
Starting...
Making an Animal
Making a Hippo
```

Child Cannot Exist Before Parents

- Remember, a subclass object might depend on things it inherits from its superclass
- Hence, the superclass parts of an object have to be fully formed before the subclass part can be constructed
- This implies the superclass constructor must finish before its subclass constructor
- The call to super() must therefore be the first statement in each constructor!
- The complier will put a call to super() as the first statement in each of your overloaded constructors if you have not done so!

Child Cannot Exist Before Parents

- Example: Identify problems in the constructors



Invoking an Overloaded Constructor

- It is possible to call a constructor from another overloaded constructor in the same class using `this()` with appropriate arguments
- The call to `this()` can only be used in a constructor, and must be the first statement in a constructor
- A constructor can have a call to either `super()` or `this()`, but never both!
- Example

```
import java.awt.Color;
class MiniCooper extends Car {
   public MiniCooper() { this(Color.RED); }
   public MiniCooper(Color color) {
      super("Mini Cooper");
      this.color = color;
        // more initialization
   }
}
```

How Long Does a Variable Live?

- How long does a variable live depends on whether the variable is a local variable or an instance variable
- A local variable is alive as long as its stack frame is on the stack. In other words, until the method completes
- A local variable is in scope only within the method in which it is declared
- When its own method calls another, the variable is alive, but not in scope until its method resumes
- The variable can only be used when it is in scope

The Difference between Life & Scope

Example

```
public void doStuff() {
   boolean b = true;
   go(4);
}

public void go(int x) {
   int z = x + 24;
   crazy();
   // imagine more code here
}

public void crazy() {
   char c = 'a';
}
```

1. doStuff() goes into a stack frame at the top of the stack. Variable 'b' is alive and in scope

The Difference between Life & Scope

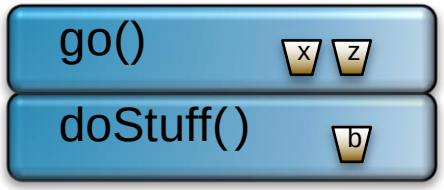
Example

```
public void doStuff() {
   boolean b = true;
   go(4);
}

public void go(int x) {
   int z = x + 24;
   crazy();
   // imagine more code here
}

public void crazy() {
   char c = 'a';
}
```

2. go() is pushed on top of the stack. Variables 'x' and 'z' are alive and in scope, and 'b' is alive but not in scope.



The Difference between Life & Scope

Example

```
public void doStuff() {
   boolean b = true;
   go(4);
}

public void go(int x) {
   int z = x + 24;
   crazy();
   // imagine more code here
}

public void crazy() {
   char c = 'a';
}
```

3. crazy() is pushed on top of the stack, with 'c' now alive and in scope. The other 3 variables are alive but out of scope

The Difference between Life & Scope

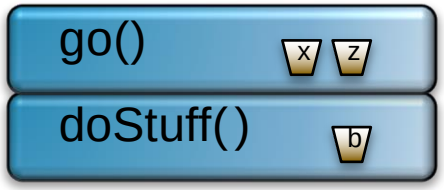
Example

```
public void doStuff() {
   boolean b = true;
   go(4);
}

public void go(int x) {
   int z = x + 24;
   crazy();
   // imagine more code here
}

public void crazy() {
   char c = 'a';
}
```

4. crazy() completes and is popped off the stack, so ‘c’ is out of scope and dead. When go() resumes where it left off, ‘x’ and ‘z’ are both alive and back in scope. Variable ‘b’ is still alive but out of scope



How Long Does a Variable Live?

What about instance variables?

- An instance variable lives as long as the object does, and is scoped to the life of the object
- In other words, instance variables live and die with the object they belong

What about reference variable?

- The rules are the same for primitives and references
- A reference variable can be used only when it is in scope

How Long Does an Object Live?

- An object is alive as long as there are live references to it
- If a reference variable goes out of scope but is still alive, the object it refers to is still alive on the heap
- If the stack frame holding a reference gets popped off the stack and that is the only live reference to the object, the object is now abandoned on the heap and becomes eligible for garbage collection
- More precisely, an object becomes eligible for garbage collection when its last live reference disappears

How Long Does an Object Live?

- 3 ways to get rid of an object's reference
  - The reference variable goes out of scope permanently

```
void go () {
   Life z = new Life();
}
```

The reference 'z' dies at the end of the method

  - The reference variable is assigned another object

```
Life z = new Life();
z = new Life();
```

The first object is abandoned when 'z' is 're-programmed' to a new object

  - The reference variable is explicitly set to null

```
Life z = new Life();
z = null;
```

The first object is abandoned when 'z' is 'de-programmed'

Example: Object Killer #1

- Example: Reference goes out of scope permanently

```
public class StackRef {
   public void foof() {
      barf();
   }

   public void barf() {
      Duck d = new Duck();
   }
}
```

1. Code from another class calls foof(), and foof() is pushed on top of the stack, with no variables declared

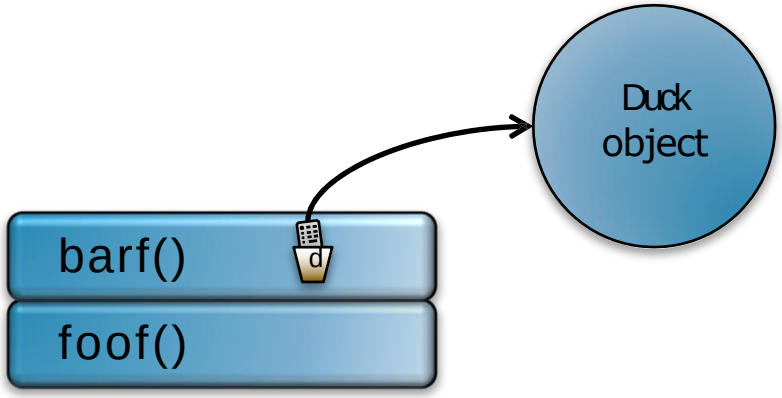
Example: Object Killer #1

- Example: Reference goes out of scope permanently

```
public class StackRef {
   public void foof() {
      barf();
   }

   public void barf() {
      Duck d = new Duck();
   }
}
```

2. foof() calls barf(), and barf() is pushed on top of the stack. barf() declares a reference variable 'd', creates a new Duck object on the heap, and assigns its reference to 'd'. 'd' is alive and in scope



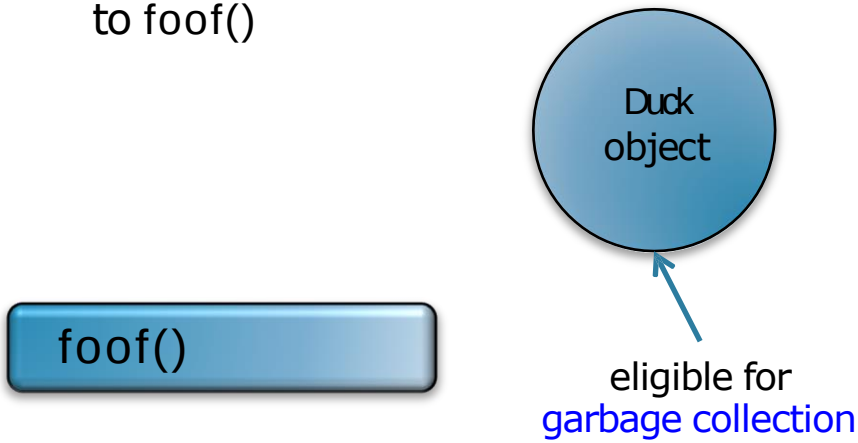
Example: Object Killer #1

— Example: Reference goes out of scope permanently

```
public class StackRef {
   public void foof() {
      barf();
   }

   public void barf() {
      Duck d = new Duck();
   }
}
```

3. barf() completes and is popped off the stack. Its stack frame disintegrates, so 'd' is now dead and gone. The Duck object is now abandoned and becomes eligible for garbage collection. Execution returns to foof()



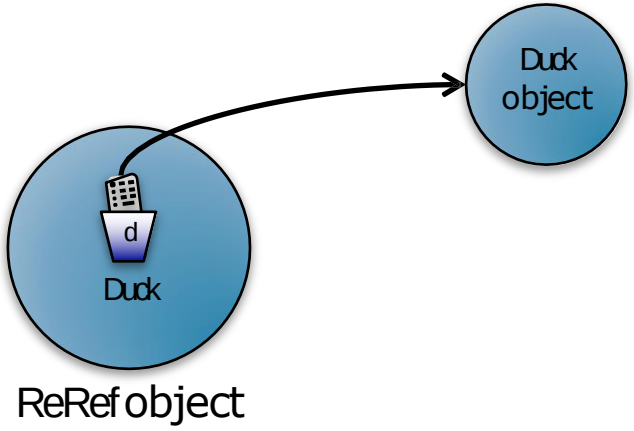
Example: Object Killer #2

- Example: Assign the reference to another object

```
public class ReRef {
   Duck d = new Duck();

   public void go() {
      d = new Duck();
   }
}
```

1. The new Duck object goes on the heap and is referenced by the instance variable 'd'. The Duck object will live as long as the ReRef object that instantiated it is alive, unless…



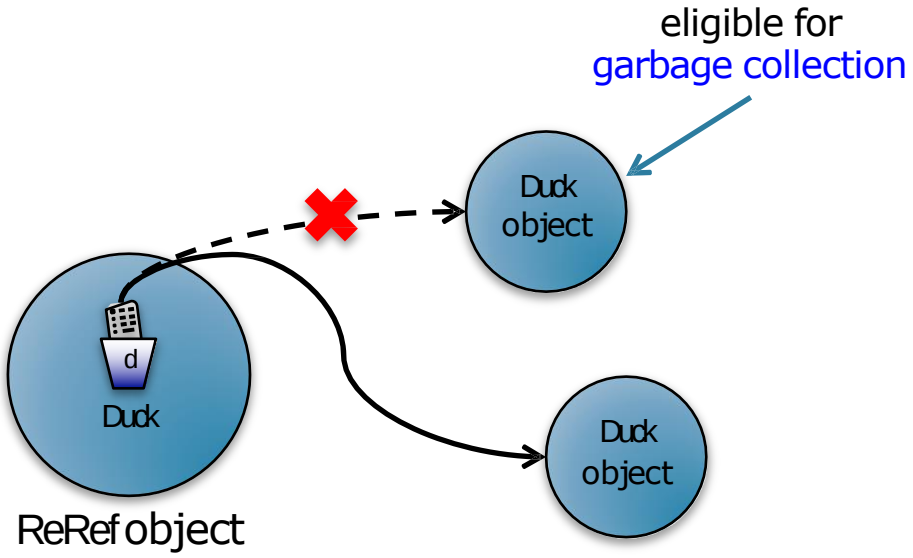
Example: Object Killer #2

- Example: Assign the reference to another object

```
public class ReRef {
   Duck d = new Duck();

   public void go() {
      d = new Duck();
   }
}
```

2. Someone calls go() where 'd' is assigned a new Duck object, leaving the first Duck object abandoned which becomes eligible for garbage collection



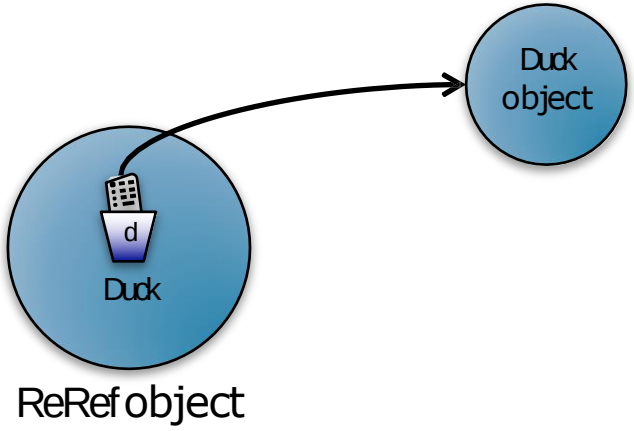
Example: Object Killer #3

- Example: Explicitly set the reference to null

```
public class ReRef {
   Duck d = new Duck();

   public void go() {
      d = null;
   }
}
```

1. The new Duck object goes on the heap and is referenced by the instance variable 'd'. The Duck object will live as long as the ReRef object that instantiated it is alive, unless…



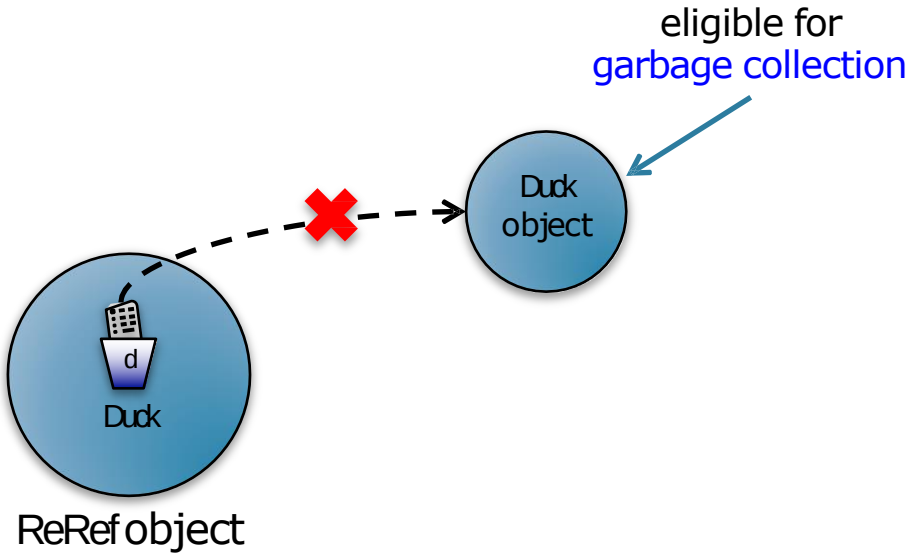
Example: Object Killer #3

- Example: Explicitly set the reference to null

```
public class ReRef {
   Duck d = new Duck();

   public void go() {
      d = null;
   }
}
```

2. Someone calls go() where 'd' is set to null, leaving the Duck object abandoned which becomes eligible for garbage collection



Any problems?

If you encounter any problems in understanding this set of materials, **please feel free to contact me or my TAs**.
We are always with you!

Chapter 13.

End

2020-2021
COMP2396 Object-Oriented Programming and Java
Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)
Department of Computer Science, The University of Hong Kong