CSSE2002/7023

Programming in the Large

Week 2.2: Variable Semantics

In this Session

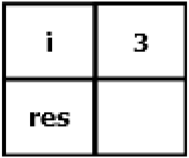
- Memory and Calls
- Parameter Passing and = Semantics
- Object Equality
- Mutable and Immutable Objects
- Inheritance

Memory and Calls

Consider `factorial()` from `Recursion.java`.

- code from week 1

factorial(3)



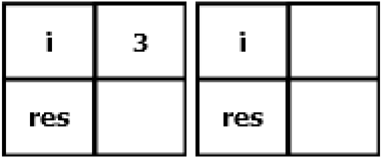
i	3
res	

Memory and Calls

Consider `factorial()` from `Recursion.java`.

- code from week 1

factorial(3)



i	3
res	

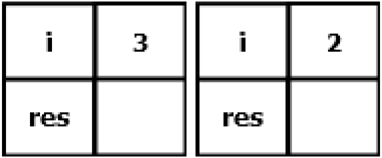
i	
res	

Memory and Calls

Consider `factorial()` from `Recursion.java`.

- code from week 1

factorial(3)



i	3
res	

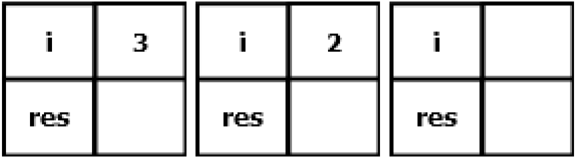
i	2
res	

Memory and Calls

Consider `factorial()` from `Recursion.java`.

- code from week 1

factorial(3)



i	3
res	

i	2
res	

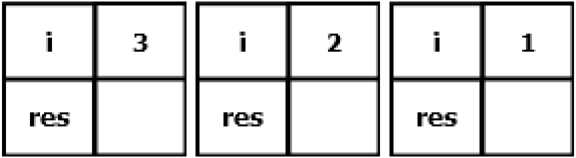
i	
res	

Memory and Calls

Consider `factorial()` from `Recursion.java`.

- code from week 1

factorial(3)



i	3
res	

i	2
res	

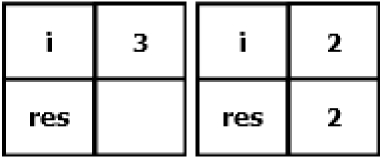
i	1
res	

Memory and Calls

Consider `factorial()` from `Recursion.java`.

- code from week 1

factorial(3)



i	3
res	

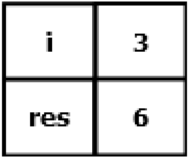
i	2
res	2

Memory and Calls

Consider `factorial()` from `Recursion.java`.

- code from week 1

factorial(3)



i	3
res	6

Memory and Calls

- When a call starts, memory is reserved to store local variables and parameters (treated as locals).
- Memory is reserved for as long as that *call* is active.
    - local variables exist as long as their call does
- When the call ends, the memory is released.
    - the variables no longer exist
- Calls won't end while they have a call active.
- A new call means a new block of memory is added to the end.

Called the call stack

- Provides an ordered lifetime

But what if you want something to live longer than the method that made it?

Heap

- Storage on the heap is not bound to calls.
- Things exist from when they are created until they are cleaned up.
  - automated garbage collection in Java
- In Java, all objects are stored on the heap.
- All local variables are stored on the stack.

Heap

- Storage on the heap is not bound to calls.
- Things exist from when they are created until they are cleaned up.
  - automated garbage collection in Java
- In Java, all objects are stored on the heap.
- All local variables are stored on the stack.

What about args in:

```
public static void main(String args[])
```

Isn't `args` a local variable *and* an object?

Parameter Passing and = Sematics

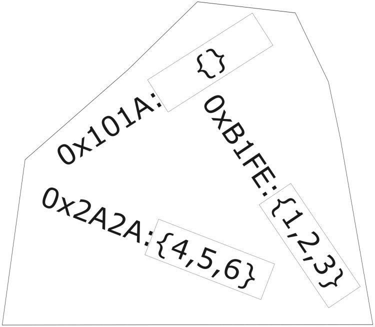
What value does a variable actually store? (What is transferred when you assign into a variable?)

- Variables of primitive types store the actual value.
- Variables of object types store a "reference[1]" to where the object is located on the heap.
- e.g. "Seat number" vs "Person"

`VariableSemantics.java`

[1] if you know C, you can think of them like pointers

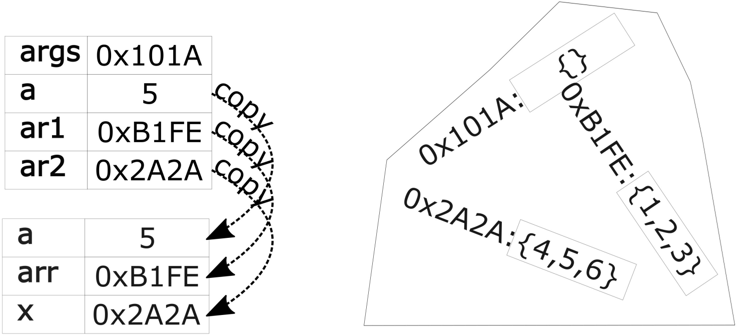
args	0x101A
a	5
ar1	0xB1FE
ar2	0x2A2A



0x101A: {}

0xB1FE: {1,2,3}

0x2A2A: {4,5,6}

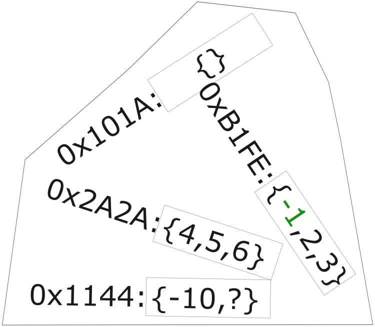


args	0x101A
a	5
ar1	0xB1FE
ar2	0x2A2A

a	5
arr	0xB1FE
x	0x2A2A

args	0x101A
a	5
ar1	0xB1FE
ar2	0x2A2A

a	100
arr	0xB1FE
x	0x1144



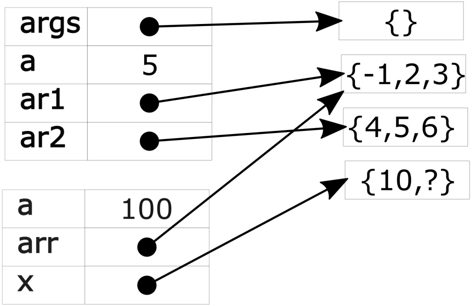
0x101A: {}

0xB1FE: {-1,2,3}

0x2A2A: {4,5,6}

0x1144: {-10,?}

Since we don't care what the actual addresses are, we are more likely to draw it like:



args	●
a	5
ar1	●
ar2	●

a	100
arr	●
x	●

= and ==

Primitive types:

```
x = y         // make x store a copy of y's value
x == y        // does x store the same value as y?
x != y        // or not?
```

Reference types: Exactly the same[3]

```
x = y         // make x refer to the same object as y refers to
x == y        // does x refer to the same object as y?
x != y        // or not?
```

Warning: This is different to Python.
In Python `x == y` does not check if x and y refer to the same object.

[3] provided you remember that the values are references to things

Aside: Comparing Floating Point Values

Testing floats for equality is not a good idea:

```
double f = 2;
double g = Math.sqrt(Math.sqrt(f));
double h = g * g * g * g;
System.out.println(h == f);
System.out.println(Math.abs(h - f));
```

```
 false
4.440892098500626E-16
```

It is better to check if the absolute value of the difference is less than some threshold.

```
Math.abs(h - f) < 0.0001
```

Object Equality?

If you want to see if two (possibly different) objects have “equivalent” values, then you need to call a method.

```
String s1 = "blue castello";
String s2 = "blue ";
String s3 = "castello";
String s4 = "blue castello";
String s5 = s4;
```

```
s1 == s2                false
s1.equals(s2)           false
s1 == (s2 + s3)         false
s1.equals(s2 + s3)      true
s1.equals(s4)           true
s4 == s5                true
s1 == s4                ?
```

Mutable/Immutable Objects

- If all access to an object's state is via methods, then you can control how state changes.

Question: Should state be able to change? Some languages can prevent change on a per object basis but Java and Python can't.

- Decisions as to whether state can change are made at the class level (does the class have mutators[2] or only accessors[3])[4]. Note: not all methods fall neatly into one of those categories.
- e.g. `Strings` are immutable, while arrays and `Lists` are mutable.
- If you are planning to use an object as a key or label for something else (e.g. in a `Map`/`dict`) it is better if it doesn't change.

[2] Methods which change state - "setters"

[3] Methods which return state information - "getters"

[4] Yes it is possible to have neither

Basic Inheritance

Inheritance — things you have because your parents have them. OO allows us to define classes as:

- *like that class but ...*

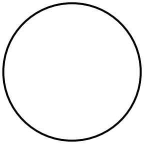
The new class is called the subclass (or possibly *child* class), the class being inherited from is called (the) superclass (or *parent* class).

Instances of a subclass are also considered to be instances of their superclass.

- a class is the set of all instances of that class

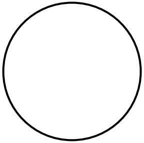
Befürchten der Pfefferkuchen Nicht

Consider a simple gingerbread cutter:

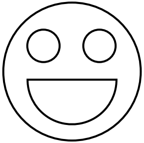


Befürchten der Pfefferkuchen Nicht

Consider a simple gingerbread cutter:

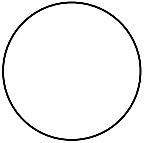


Now a second cutter which has more features but the same outside shape:

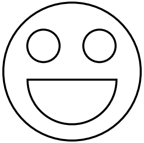


Befürchten der Pfefferkuchen Nicht

Consider a simple gingerbread cutter:

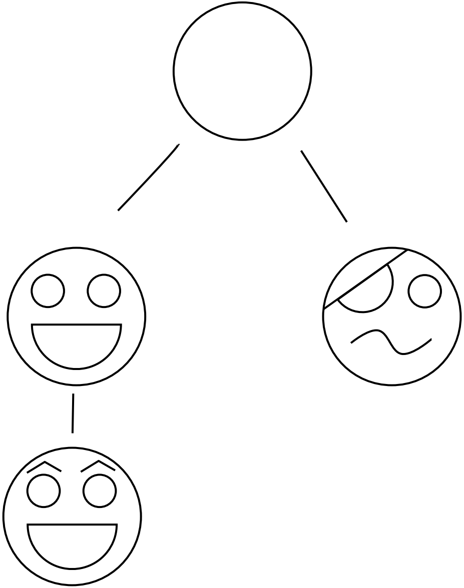


Now a second cutter which has more features but the same outside shape:



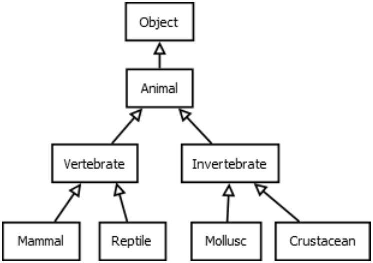
The first cutter will fit over shapes made by both cutters.
The second cutter will only fit (cleanly) over its own shapes.

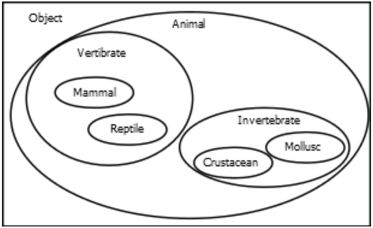
Gingerbread Inheritance



Shapes produced from more complex cutters also fit the cutters above them.

Basic Inheritance — “is-a”





```
public class Mammal extends Vertebrate
```

i.e. A `Mammal` is-a `Vertebrate`.

In Java if a class does not explicitly `extend` anything it automatically extends `java.lang.Object`. So (by transitivity) every object in Java is an instance of `Object`.

Basic Inheritance — like that class but ...

What changes can we make (in Java)?

- Add new methods (different name)
- Add new member variables
- Overload existing methods
- Override (redefine) existing methods

What can't we do?

- Change the type or parameters of existing methods
- Change the type of member variables
- Tighten access control of any members

That is, if it is part of a super class' interface, it must be part of the subclass' interface as well.

What's in an Object

Javadoc is a good place to start (online version at `https://docs.oracle.com/javase/8/docs/api/`).

`java.lang.Object` has 11 methods[5]. Of interest to us:

- `protected Object clone()`: involved in copying objects.
- `boolean equals(Object)`: is `this` object equal to another object.
- `int hashCode()`: get a number representing the object.
- `String toString()`: get a String to represent the object.

The `toString()` method is why you can `System.out.print` any object.
Note: Just because a method is defined, doesn't mean it is defined usefully.

[5] Constructors are not methods

@Override — Change toString() on CoffeeCup

We know there is a `toString` inherited from `Object` but we want to make a more useful one.

```
public class CoffeeCup {
  public double amountOfCoffee;
  public double strengthOfCoffee;

  @Override
  public String toString() {
    return "CoffeeCup (Amount: " + amountOfCoffee +
        " Strength: " + strengthOfCoffee + "%";
  }
}
```

Notes:

- `@Override` — not necessary, but may help identify errors.
- `"" + x` — string concatentation works for `String + ?`, but not for `?  + String` (not commutative).

Inheritance — What Goes Where?

```
public class X {
  public int a;
  private int b;
  public X() {...}
  public int f(){...}
  private int g(){...}
}
```

```
public class Y extends X{}
```

In Y, the following will be public: *default constructor*, `a`, `f()`.

The following will be inaccessible (not `private`): `b`, `g()` (they are still there but the only way to interact with them is via methods on X).

`private` keeps everyone else out, even subclasses.

`protected` is a compromise: Methods of that class *and* any subclasses can use it, but no one else. Members `protected` in the superclass are `protected` in the subclass.