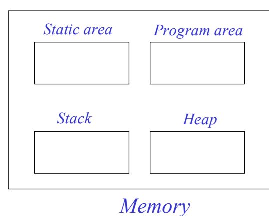
Ur-Java

Ur-Java

- Let us introduce Java in two stages:
  - Ur-Java: a class language, no objects
  - Java: a language with objects
- Ur-Java is a subset of Java
  - every Ur-Java program is a Java program
- Why study Ur-Java?
  - I want you to have a mental model of how Java programs are executed
  - Ur-Java has a simple execution model

Memory map for modern languages



- Static area: class variables
- Program area: code (like our SaM code)
- Stack: frames containing method parameters/variables
- Heap: objects created by constructor invocation
- Ur-Java: no objects, so no heap

Example of Ur-Java program

```
Class Top{
   public static void main(String[] args) {
            Work.squares(1,10);
             System.out.println(Work.powCalls);
              }
 }
Class Work{
    public static int powCalls = 0;

    public static void squares(int lo, int hi){
        for (int i = lo; i < hi; i++)
            System.out.println(pow(i,2));
     }
     public static int pow(int b, int p){//p>0
        powCalls = powCalls + 1;
        int value = 1;
        for (int i = 0; i < p; i++)
            value = value*b;
        return value;
     }
}
```

class variable (points to `public static int powCalls = 0;`)

class method (points to `squares` and `pow`)

Ur-Java program

- Class: like a folder that contains
  - some class variables (maybe none)
  - some class methods (maybe none)
- These are called class members.
- Just as in folder, class should contain logically related members.
- Example: members in Java class Math
  - Class variables named PI, E etc.
  - Class methods named sin,cos,pow,….

Names of members

How does a method in one class refer to a member of another class?

- Complete path name: className.memberName
  - (eg) Top.main, Work.powCalls,Work.squares
- Relative path name: memberName only
  - Used when referring to member in same class as method
  - (eg) method Work.squares can refer to member Work.powCalls simply as powCalls
- Analogy: long-distance call vs local call in phone system

Method overloading

- Can two methods in a class have the same name?
- Two methods in a class can have the same name provided
  - they take different numbers of arguments, or
  - the type of at least one argument is different
- This is called method overloading.
- Why is this useful?

- Suppose we want to define a power method for floats.
- Type of method for integers:
  - int x int → int
- Type of desired method for floats:
  - float x int → float
- We need another method – what should we name it?

Method overloading

```
public static int pow(int b, int p){//p>0
        powCalls = powCalls + 1;
        int value = 1;
        for (int i = 0; i < p; i++)
            value = value*b;
        return value;
    }

public static float pow(float b, int p){
        powCalls = powCalls + 1;
        float value = 1.0;
        for (int i = 0;i <p; i++)
            value=value*b;
        return value;
    }
```

Finds powers of integers

Methods have same name but types of parameters are different.

Finds powers of floats

Why overloading

- We could of course have called the two methods iPow (powers of integers) and fPow (powers of floats).
- This obscures the similarity in their functionality: overloading method name is cleaner.
- How does compiler figure out which method to call when it sees invocation pow(….,…)?
    - In this example, type of first parameter tells it which method was intended to be invoked.

Editorial note

- Much of the power (and conceptual complexity) in OO-languages comes from the subtleties of determining the association between names and “things”.
- In older languages like FORTRAN, a name stood for exactly one thing.
- On OO-languages, a name may mean different things at different places in program or at different times in program execution.
    - Method overloading is a simple example of this.
    - Method overriding is a more complex and powerful example (see later in inheritance).

Let us look at Ur-Java execution.

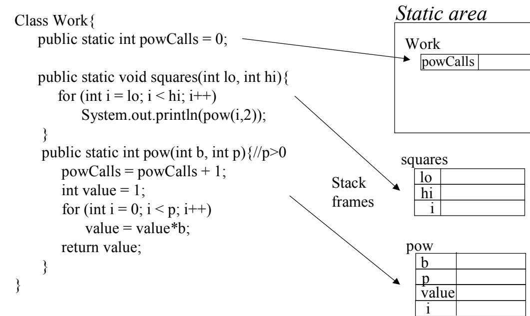
Memory map

- Class variables
  - Created in static area when program execution begins
  - Stay in existence till program terminates
- Method parameters/variables
  - Stack frame containing parameters/variables created in stack area when method is invoked
  - Stack frame destroyed when method returns
- Note difference between these two
  - Each class variable corresponds to exactly one memory location for entire duration of program.
  - Method parameters/variables can correspond to different locations at different points in program execution.

Memory map of class Work

```
Class Work{
    public static int powCalls = 0;

    public static void squares(int lo, int hi){
        for (int i = lo; i < hi; i++)
            System.out.println(pow(i,2));
    }
    public static int pow(int b, int p){//p>0
        powCalls = powCalls + 1;
        int value = 1;
        for (int i = 0; i < p; i++)
            value = value*b;
        return value;
    }
}
```

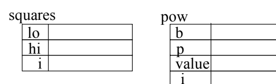


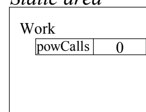
Let us look at invocation squares(1,10).

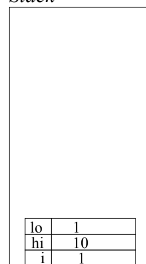
Just after invocation Work.squares(1,10).

```
Class Work{
    public static int powCalls = 0;

    public static void squares(int lo, int hi){
        for (int i = lo; i < hi; i++)
            System.out.println(pow(i,2));
    }
    public static int pow(int b, int p){//p>0
        powCalls = powCalls + 1;
        int value = 1;
        for (int i = 0; i < p; i++)
            value = value*b;
        return value;
    }
}
```







Just after invocation pow(1,2)

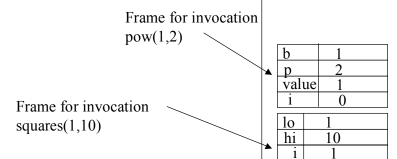
```
Class Work{
    public static int powCalls = 0;

    public static void squares(int lo, int hi){
        for (int i = lo; i < hi; i++)
            System.out.println(pow(i,2));
    }
    public static int pow(int b, int p){//p>0
        powCalls = powCalls + 1;
        int value = 1;
        for (int i = 0; i < p; i++)
            value = value*b;
        return value;
    }
}
```

Static area

Work	
powCalls	0

Stack



Just after invocation pow(1,2) returns

```
Class Work{
    public static int powCalls = 0;

    public static void squares(int lo, int hi){
        for (int i = lo; i < hi; i++)
            System.out.println(pow(i,2));
    }
    public static int pow(int b, int p){//p>0
        powCalls = powCalls + 1;
        int value = 1;
        for (int i = 0; i < p; i++)
            value = value*b;
        return value;
    }
}
```

Static area

Work

powCalls	1

Stack

Frame for invocation squares(1,10)

lo	1
hi	10
i	1

Why class variables?

- Constants needed by many methods/classes
  - PI,E in class Math
- Data that must survive method invocations
  - powCalls is one example
  - Another example: random number generation

Random number generation

- The following formula can be used to generate a sequence of random numbers

  $x_0 = 19$

  $x_k = (106*x_{k-1} + 1283) \mod 6075$

```
Class random {//returns sequence starting at x1
    private static int current = 19;
    public static float Rand() {
        current = (106*current + 1283) % 6075;
        return current;
}
```

Note

- Use of class variable *current* is essential because value returned by an invocation of method Rand depends on values computed by previous invocation of Rand.
- Method parameters/variables are not adequate for this purpose.

Final comments

- Ur-Java has classes, but no objects.
- Visibility of class members can be controlled with access specifiers such as *public* and *private*.
- Ur-Java is a conventional non-OO language like C except that visibility of class members can be controlled.