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?
- Introduce the idea of encapsulation
- I also want you to have a mental model of how Java programs are executed
  - Ur-Java has a simple execution model

Two aspects of Ur-Java

- **Statics**: what does the program look like?
  - What are the constructs in the language?
- **Dynamics**: what happens when you run the program?
  - What is the sequence in which program operations are executed?
  - What is the correspondence between names and storage locations?

Statics of Ur-Java
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;
    }
}

Names of members

- **Top**
  - Main
- **Work**
  - PowCalls
  - Squares
  - Main
  - Pow

Binding

- **Binding**: association between name and class member
- (eg) System.out.println(Work.powCalls);
- **pow** is name for some class member. Which one is it?

  - **Ur-Java**: static binding
  - Association between name and member can be determined from
    text of program without running the program
  - (eg) pow means the method defined in Work.pow
  - “static” means compiler can determine binding (using types of
    names if necessary)

- **Contrast: dynamic binding** – association between name and member can only be determined by running program
  - See later when we look at object-oriented Java

Ur-Java program

- Collection of classes
  - Example: Top and Work are two classes
- **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, …
Binding in example program

```java
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;
    }
}
```

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 \rightarrow int
Type of desired method for floats:
  – float x int \rightarrow float
We need another method – what should we name it?

Methods have same name but types of parameters are different.
Finds powers of integers
Finds powers of floats

```java
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;
}
```
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.
  – Note: this is an example where the compiler needs to use type information to determine binding; path name of the method is not enough

Visibility and Encapsulation

• Class member M can be declared to be
  – public: visible to methods in other classes
  – private: visible only to methods in same class as M
• Encapsulation: hiding members from methods in other classes
  – Variables like powCalls should usually be declared private so methods in other classes cannot write to them directly
  – Instead, introduce methods to implement functionality you want to expose
  – You might want to make methods private as well if they are not needed by methods in other classes
  – Idea: control the amount of interaction between code in different classes

Example of Encapsulation

```java
public static void main(String[] args) {
    Work.squares(1,10);
    System.out.println(Work.numCalls());
}

class Work{
    private static int powCalls = 0; //variables declared private
    public int numCalls() {   //read-only access enforced by method
        return powCalls;
    }
    public static void squares(int lo, int hi){
        for (int i = lo; i < hi; i++)
            System.out.printf(pow(i,2));
    }
}
```

Variables in methods

• Two kinds of variables:
  – Parameters: b,p
  – Local variables: value,i
• Variables not visible outside method
• Method parameters and local variables should not be declared to be public/private
  – by definition, they are visible only in that method
Dynamics of Ur-Java

Memory map

- **Class variables**
  - Created in static area when program execution begins
  - Stay in existence till program terminates

- **Method parameters/local variables**
  - Frame containing parameters/local variables created in frame area when method is invoked
  - Frame contains other information: ignore for now
  - 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.

Example: class Work

```java
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;
    }
}
```

Memory map for modern languages

- **Program area**: code (like our SaM code)
  - Each method is compiled to SaM-like code by compiler
  - When program runs, this code is loaded into program area
- **Static area**: class variables
- **Frame area**: frames containing method parameters/local variables
- **Heap**: objects created by constructor invocation
- **Ur-Java**: no objects, no heap

Frame for method invocation: from bottom to top
- One slot per parameter, left to right order
- One slot per local variable

Static area

- Work
- powCalls
- lo
- hi
- value
- pow
- powCalls
- templates

Program area

- squares
- pow
- squares
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;
    }
}

Let us look at frame area after 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;
    }
}

Static area

Frame area

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;
    }
}

Frame for invocation pow(1,2)

Frame for invocation squares(1,10)

Static area

Frame area

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;
    }
}

Frame for invocation squares(1,10)

Frame for invocation of main

Static area

Frame area

Frame for invocation of main
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
  \[ x_0 = 19 \\ x_k = (106 \cdot x_{k-1} + 1283) \mod 6075 \]

```java
class Random {//returns sequence starting at \( x_1 \)
    private static int current = 19;
    public static float rand() {
        current = (106*current + 1283) % 6075;
        //return float in range \([0,1)\]
        float scaled = current/6074;
        return scaled;
    }
}
```

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.

Java note

• Java Math class has a random number generator
  – `Math.random()` : returns a random double value in range \([0.0, 1.0)\)
  – Example: simulating a die \([1..6]\)
    ```java
    public static int die() {
        return (int)(Math.floor(Math.random() * 6) + 1);
    }
    ```
Editorial note

- Difficult of writing and maintaining large programs
  - Related to complexity of interaction between different portions of code
  - More disciplined interactions ➔ less complex code

- Encapsulation:
  - Visibility of class members is controlled
  - Permits control over complexity of interactions between classes
  - Public/private are linguistic mechanisms for this
  - In a language like C, this can be accomplished by disciplined programming

- Binding:
  - 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 in Ur-Java is a simple example of this.
    - Method overriding is a more complex and powerful example (see later in inheritance).

Additional material

Program Development

- Edit/compile/run
  - When do you catch mistakes?
  - Prefer to do it as early as possible in development cycle
  - To understand this, let us look at categories of mistakes
# Categories of mistakes

- Similar to categories in English
- **Syntactic mistakes**: “Spot give lecture.”
  - Grammatical: “Spot gives a lecture.”
- **Semantic mistakes**:
  - **Type error**: if Spot is a name only for dogs, sentence is syntactically correct, but meaningless
    - Do not need to know which dog Spot is
  - **Runtime error**: “John gives a lecture.”
    - May or may not make sense depending on who John is
      - If John is 3 years old, does not make sense

## PL examples

- **Syntactic errors**:
  - `(eg) 3var = 5;`  
    //Java identifiers cannot start with digit
- **Semantic errors**:
  - Type errors:
    - `(eg) a/b`  
      //if type of “a” is boolean
  - Runtime errors:
    - `(eg) a/b`  
      //if value of b is 0

---

# Program Debugging

- When do you catch mistakes?
  - Edit time: some syntactic errors
  - Compile time: type errors, missing method definitions,...
  - Run time: divide by zero errors,…
- Prefer to catch mistakes as early as possible in development cycle