Object-oriented Programming

Goal of lecture

- Understand inadequacies of class languages like Ur-Java
- Extend Ur-Java so it becomes an
  object-oriented language
- Implementation in SaM
  - heap allocation of objects
  - references to objects
- Important built-in classes in Java
  - String
  - Array
- Understand concepts of
  - modularity of code
  - data abstraction: important design technique for making code more modular

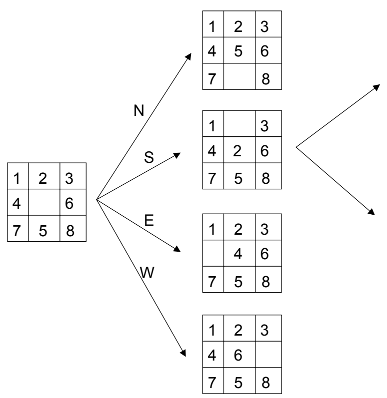
Context of discussion

- Programming in the large
  - Big applications require many programmers
- General approach to coding:
  - Break problem into smaller sub-problems
  - Assign responsibility for each sub-problem to somebody
  - Assemble everyone's contributions
- For this to work, each sub-problem must have a carefully defined specification
  - Functionality: what services must code provide?
  - Interface: how do clients obtain that functionality?
- Job of programmer: provide an implementation (code) that meets the specification

Message of lecture

- Separating interface from implementation is useful for writing more modular code that is easy to maintain
  - Implementer of class can change his code without affecting interface, so client of class does not need to change code
  - Called data abstraction in literature
- Class language:
  - may be difficult to separate interface from implementation
- Object-oriented language:
  - separation of interface from implementation is possible
  - more modular code

Running application: 8-puzzle



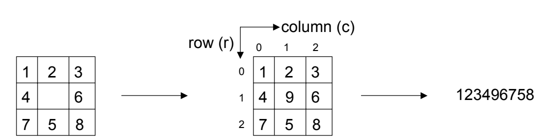
Program organization

- Two classes
  – Puzzle: written by you
    - Functionality: methods to
      – scramble
        » put puzzle in Sam Loyd configuration
      – move
        » move a tile N/S/E/W to get new configuration
      – tile
        » what tile is there in some position?
    - Interface: ?? ← this is what we will study
  – TestPuzzle: client class, written by someone else, which will use interface to play game

Implemention of class Puzzle

- Two sub-tasks:
  – What is the representation of state (puzzle configuration) in your class?
    - Ur-Java complication: only base types like int, long, float, etc.
  – Given this representation, how do we implement operations like scramble and move?

Representation of state



- Model puzzle as 3x3 grid of integers
  – Use 9 to represent empty square
  – Three rows (0..2) and three columns (0..2)
- Convert grid into single integer
  – Concatenate tiles in left to right, top to bottom order
- Converting integer s into grid
  – Remainder when s is divided by 10: tile in position (2,2)
    - Java expression: s %10
  – Quotient after dividing by 10 gives encoding of remaining tiles
    - Java expression: s / 10
  – Repeat remainder/quotient operations to extract remaining tiles
- This kind of encoding may seem strange but the need for it arises in many places in computer science
  – Storing multi-dimensional arrays in memory
  – Godelization of logical proofs

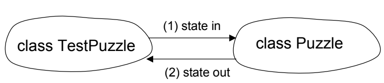
Implementing puzzle operations

- scramble: put puzzle into Sam Loyd configuration
  – s = 123456879;
- tile (r,c): what is tile in position (r,c)?
  – return $(s/10^{8-(3r+c)})$ % 10
- move: left to reader

Key question in Ur-Java

- What kind of variable should we use to store the integer that represents puzzle state?
  1. method parameter/local variable
     - stack allocation
  2. class variable
     - allocated in static area
- As we will see, these implementation choices affect the interface of the Puzzle class.

Interface L(ocal)



- State is implemented as local variable in class TestPuzzle
  – passed to/returned from methods in Puzzle class
- Interface of Puzzle class: //informal

```
{ int scramble(); //returns encoding of initial state
  int tile(int state, int r, int c); //return tile in
                   position (r,c) of grid
  int move(int state, char d); //returns encoding of new
           state
}
```

Ur-Java code: Interface L

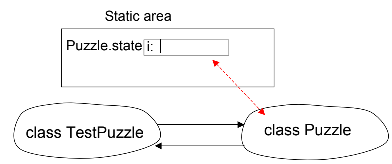
```
class testPuzzle {

 public static void main(String[] args) {
    int  state = Puzzle.scramble();
    display(state);
    state = Puzzle.move(state, 'N');
    …..
 }

 public static void display(int state) {
   for (int r = 0;r < 3; r++) {
     for (int c = 0; c < 3; c++)
       System.out.print(Puzzle.tile(state,r,c) + " ");
     System.out.println(" "); //new line
   }
 }

}
```

```
class  Puzzle {

  public static int scramble() {
    return 123456879;
  }

  public static int tile(int state, int r, int c) {
    return state/((int)Math.pow(10,8-(3*r+c))) %10 ;
  }

  public static int move(int state, char d) {
       …..
  }

}
```

Critique of Interface L

- No data abstraction
  - Puzzle class implementer chose to implement state as an int.
  - This is visible to the interface to the class, so client code is aware of it.
  - If Puzzle class implementer decided to implement state as a *long*, client code must change.

Interface S(tatic)



- State is implemented as class variable in class Puzzle
  - state does not have to be passed back and forth between two classes
- Interface of Puzzle class: //informal

```
{ void scramble(); //updates class variable state
  int tile(int r, int c); //return tile in position (r,c) of grid
  void move(char d); //updates class variable state
}
```

Ur-Java code: Interface S

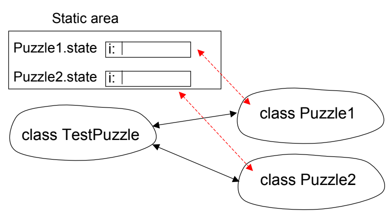
```
class testPuzzle {

 public static void main(String[] args) {
    Puzzle.scramble();
    display();
    Puzzle.move('N');
    …..
 }

 public static void display() {
   for (int r = 0;r < 3; r++) {
     for (int c = 0; c < 3; c++)
       System.out.print(Puzzle.tile(r,c) + " ");
     System.out.println(" "); //new line
   }
 }

}
```

```
class  Puzzle {
  public static int state;

  public static void scramble {
    state = 123456879;
  }

  public static int tile(int r, int c) {
    return state/((int)Math.pow(10,8-(3*r+c))) %10 ;
  }

  public static void move(char d) {
       …..
  }

}
```

Critique of Interface S

- Data abstraction: yes
  - Puzzle class implementer chose to implement state as an int.
  - This is NOT visible to the interface of class, so client code is NOT aware of it.
  - If Puzzle class implementer decides to change implementation of state to long, client code does not have to change.
  - Modularity: change to representation of data in programmer's code does not affect client code
- Problem: client can have only one puzzle to play with
  - state is a private class variable in class Puzzle
  - Mechanism we have used (class variable) gives right of puzzle creation to implementer of class rather than the client of the class
  - ☹

Sneaky solution to problem



- Make copies of code of Puzzle class and rename classes!
- If client wants *n* puzzles, it makes *n* copies of code of Puzzle class and renames them.

Sneaky Ur-Java code: Interface S

```
class testPuzzle {

 public static void main(String[] args) {
     Puzzle1.scramble();
     display1();
     Puzzle1.move('N');
     Puzzle2.scramble();
     .....
 }

 public static void display1() {
   for (int r = 0;r < 3; r++) {
     for (int c = 0; c < 3; c++)
       System.out.print(Puzzle1.tile(r,c) + " ");
     System.out.println(" "); //new line
   }
 }
 //similar display method for Puzzle2
}
```

```
class  Puzzle1{
  public static int state;

  public static void scramble {
    state = 123456879;
  }
  .....................
}
```

```
class  Puzzle2{
  public static int state;

  public static void scramble {
    state = 123456879;
  }
  .....................
}
```

Critique

- Data abstraction: yes
- Creation on demand: yes, but at cost of duplicating code for Puzzle class and renaming classes
  - Size of program increases
  - Must know at compile-time how many puzzles are needed
  - Class code duplication and renaming must be repeated whenever new release of Puzzle code becomes available
- Some client code may need to be duplicated
  - Look at display method(s) in previous slide

High-level picture

- Copying code for Puzzle class and renaming gives us
  - name to refer to a particular puzzle
  - variable (*state*) to store configuration
  - methods to manipulate *state*: variable *state* in method of class *Puzzlen* refers to class variable of *Puzzlen*
- Question: can we design mechanisms in the language for doing all this?

Solution: ask Gutenberg!

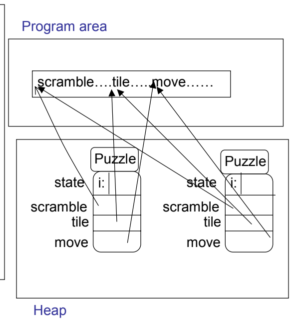
- Algorithm for making a copy of a book in ancient times:
  - Hire a monk
  - Give monk paper and quill
  - Ask monk to copy text of book
- Algorithm for making *n* copies of a book
  - Hire a monk
  - Give monk lots of paper and quills
  - Ask monk to copy text of book *n* times
- Gutenberg (Strasbourg ca.1450 AD) : new algorithm for making n copies of a book
  - First make a template of book using movable type
  - Stamp out as many copies of book as needed
- Copying class code is like medieval approach to copying books!
- How do we exploit Gutenberg's insight in our context?
  - What is template for puzzles?
  - What do we stamp puzzle instances out of?
  - How do we stamp out puzzle instances as needed from template?
  - How do we name different puzzle instances?

Object-oriented languages

- Class definition is template
  - Instance variables: when puzzle is stamped out, it will have storage for these variables
  - Instance methods: methods for manipulating instance variables
  - Constructor: special method in class definition that is invoked to stamp out puzzles
    - Provided automatically by system when class is defined
- Instances of classes are called objects
  - Type of object: class name
- Objects are stamped out in area of memory called heap
  - objects have one slot for each instance member

Heap allocation

```
class  Puzzle {
  private int state;

  public void scramble {
    state = 123456879;
  }

  public int tile(int r, int c) {
    return state/((int)Math.pow(10,8-(3*r+c))) %10 ;
  }

  public void move(char d) {
        …..
  }
}
```



Heap shows two instances of class Puzzle.
Class name is used as type of object.
Each object has its own instance variables.
Instance variables are declared private, so not accessible to client.
Instance methods are compiled into SaM code and stored in Program area.
All objects of type Puzzle share code for instance methods as shown in picture.

Naming instances

- Reference: variable that is a name for instances of some class
  - Implementation note: reference contains either NULL or the address of some object
- Type of reference: specifies the type of objects that reference can name
  - class name used as type of reference

    Puzzle p1; // declare a reference
- Assignment to reference

  p1 = new Puzzle(); //p1 is name for the new object

  Puzzle p2 = new Puzzle();//combination
- Accessing instance member

  p1.scramble()
  - Implementation note: this member access is implemented by SaM that does the following:

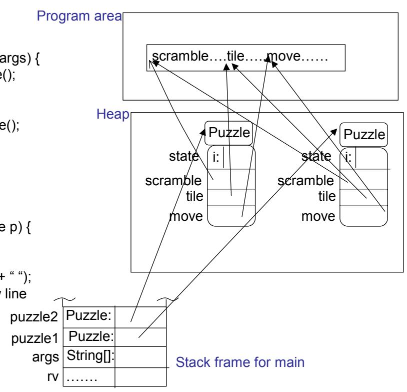
    . Examine object pointed to by p1

    . Look inside object for starting address of method named scramble

    . Invoke that method

Client code

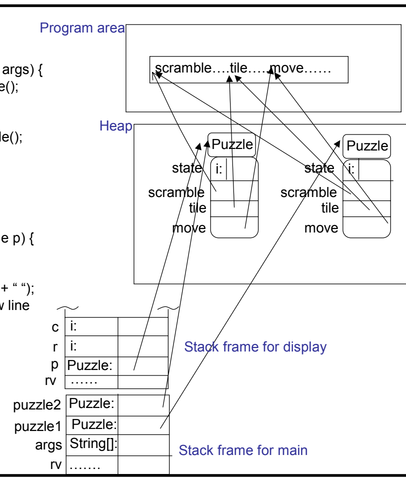
```
class testPuzzle {

 public static void main(String[] args) {
     Puzzle puzzle1= new Puzzle();
     puzzle1.scramble();
     display(puzzle1);
     Puzzle puzzle2 = new Puzzle();
     puzzle2.scramble();
     display(puzzle2);
     …..
 }

 public static void display(Puzzle p) {
   for (int r = 0;r < 3; r++) {
     for (int c = 0; c < 3; c++)
       System.out.print(p.tile(r,c) + " ");
     System.out.println(" "); //new line
   }
 }
}
```



Method invocation

- References can be passed as parameters
  - formal parameter becomes name for object in caller
  - object can be manipulated in callee using that name
  - on method return, caller sees any changes made to object by callee
- Ex: display method (see overleaf)
  - no need to have different code for each puzzle instance

```
class testPuzzle {

 public static void main(String[] args) {
     Puzzle puzzle1= new Puzzle();
     puzzle1.scramble();
     display(puzzle1);
     Puzzle puzzle2 = new Puzzle();
     puzzle2.scramble();
 →   display(puzzle2);
     …..
 }

 public static void display(Puzzle p) {
 → for (int r = 0;r < 3; r++) {
     for (int c = 0; c < 3; c++)
       System.out.print(p.tile(r,c) + " ");
     System.out.println(" "); //new line
   }
 }
}
```



Critique

- Data abstraction: yes
- Creation on demand: yes
  - No need to duplicate class code
- No need to duplicate code in client
  - Same display method handles all puzzles

Storage reclamation

- *Live object*: object is live at a particular time *t* during program execution if that object can be accessed by program after time *t.*
  – Finding live objects (recursive definition)
    - If stack contains reference to an object O, O is live.
    - If live object O1 contains reference to object O2, O2 is itself live.
- *Dead object*: object not live
- Periodically, system detects dead objects and reclaims their storage.
  – *Garbage collection*
- C,C++: difficult to determine what is a reference
  – Storage reclamation must be done explicitly by programmer (free)
  – Error-prone

Editorial remarks

- Tension in class language like Ur-Java:
  – Data abstraction: client class should not know how “state” is implemented
    → state cannot be implemented as local variable of client code
    → state must be represented as class variable.
  – Unfortunately this makes it difficult to create state dynamically.
- OO-languages resolve this tension.
  – State is encapsulated in class definition, but class definition is only a template.
  – Instances of class can be created on demand by client without breaking abstraction.
  – Client hold holds references to objects, but implementation of state is not visible to it.
  – User-defined types: class names are used as types of object and as types of references

Extra material

Aliasing

- Aliasing in real life: two names for same person
  – Example: Samuel Clemens aka Mark Twain
- Java aliasing: two references for same object

```
Puzzle p1 = new Puzzle();
Puzzle p2 = new Puzzle();
        p2 = p1;  //p2 and p1 point to same object
```

- What happens to Puzzle object pointed to by p2 before the assignment p2 = p1; ?
  – Dead object: storage gets garbage collected

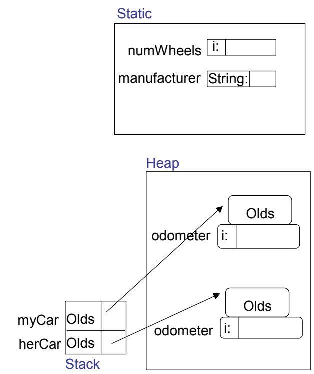
Java references vs C pointers

- C
  – Reference to object is a pointer (address) that must be dereferenced explicitly to access object
  – Pointer arithmetic is allowed
- Java
  – Implicit dereferencing: programmer only sees references, and cannot manipulate objects directly
  – No arithmetic is allowed on references
- Java approach is safer
  – Many program bugs arise from incorrect pointer manipulation
  – Viruses often get foothold by exploiting buffer overflows that arise in C programs that manipulate addresses in insecure ways

Class variables revisited

- Class variables/methods can coexist with instance variables/methods
- All instances share class variables

```
class Olds {
  private static int numWheels = 4;
  private static String manufacturer =
     "GM";
  private int odometer = 0;
  ….
}
….
Olds myCar = new Olds();
Olds herCar = new Olds();
//myCar.numWheels is same location
   as herCar.numWheels
//Each car has its own location called
   odometer
```



Keyword: *this*

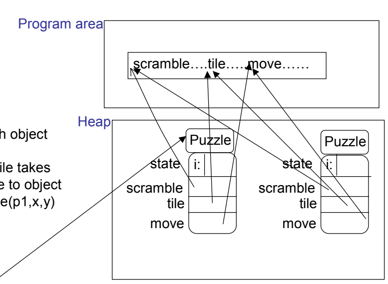
- In instance method, keyword *this* is a reference to object in which the method exists (conceptually).

```
class Puzzle {
  ……….
  public void move(char d) {
   ………
   //invoke display method in testPuzzle
   testPuzzle.display(this);
  }
 ……..
}
```

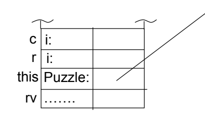
```
class testPuzzle {
 public static void main(String[] args) {
     Puzzle puzzle1= new Puzzle();
     puzzle1.scramble();

      …..
 }
 public static void display(Puzzle p) {
   for (int r = 0;r < 3; r++) {
   ………}
 }
```

Compiling instance methods

….p1.tile(2,3)…
….p2.tile(0,1)…



- How does tile method know which object to manipulate?
- Answer: low-level SaM code for tile takes an additional parameter: reference to object
  - p1.tile(x,y) is converted to p1.tile(p1,x,y) in essence



Stack frame for invocation of tile

Strings

- String: sequence of characters
- String s = new String(“green”);
- Length of string: s.length()
- Characters in string: s.charAt(i) returns character in position i. Positions start at 0.
- String comparison: s1.equals(s2) returns true is s1 and s2 are same sequence of characters
  – s1 == s2 returns true if s1 and s2 are references to same object

Arrays

- Arrays are objects in Java.
- Declaring array references
  Puzzle[] ap; //ap is reference to array of puzzles
- Allocating array
  ap = new Puzzle[20]; //create an array that can hold references to 20 Puzzle objects
- Array assignment
  ap[0] = new Puzzle();
- Array access
  ap[0]

Array example

…….
Puzzle[] ap;
ap = new Puzzle[20];
ap[0] = new Puzzle();
……..