Object-Oriented Programming

Object-Oriented Programming

- What do we mean by *object-oriented*?
- Why use it?
  - modularity (implementation hiding)
  - code reuse
  - type safety
  - inheritance (next time)
- Implementation
  - heap allocation of objects
  - references to objects

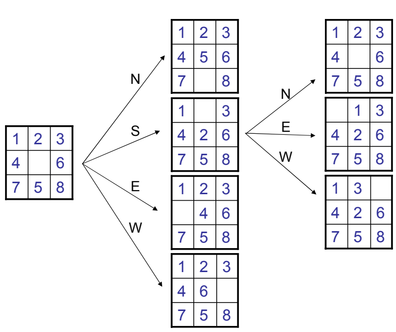
Some Context

- Programming “in the large”
  - big applications require many programmers
- General approach
  - break problem into smaller subproblems
  - assign responsibility for each subproblem to somebody
  - keep the interfaces small!
- Each subproblem must have a *specification*
  - Functionality: What services must code provide?
  - Interface: What input conditions does the code expect? What output conditions does it guarantee?
- Job of the programmer: provide an *implementation* (code) that meets the specification

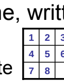
The Message

- Separate the *specification* from the *implementation*
  - called data abstraction in the literature
  - more modular, easier to maintain
  - implementation is hidden from the client, can be changed without changing the interface
  - the client’s code does not break
- Object-oriented languages
  - encourage data abstraction
  - more modular code

The 8-Puzzle



Program Organization

- `class Puzzle`
  - an implementation of the game, written by you
  - functionality:
    
    - init – put puzzle in the initial state
    - move – move a tile N, S, E, or W to get a new state
    - tile – report which tile is in a given position
- `class TestPuzzle`
  - a client class, written by someone else
  - will communicate with `Puzzle` (your code) to play the game

Implementation

- Two subtasks
  - How do we represent a state (puzzle configuration)?
  - Given the representation, how do we implement **init**, **move**, and **tile**?
- Suppose no objects...

Representation of State

1	2	3
4	9	6
7	5	8

→ 123496758

- Model puzzle state as an integer between 123456789 and 987654321
  - 9 represents the empty square
- To convert integer s into a grid representation:
  - Remainder when s is divided by 10: tile in bottom right position
    - 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 encoding may seem strange, but it arises many places in CS
  - Storing multidimensional arrays in memory

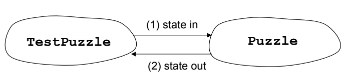
Implementing Operations

- `init`: put into initial configuration
  `s = 123456879;`
- `tile`: what tile is in position `(row,col)`?
  ```
  return s/((int)Math.pow(10,8-
    (3*row+col)))%10;
  ```
- `move`: left to the reader

A Key Question

- Where do we keep the state?
  1. method parameter/local variable
     - client keeps track of it
     - passed to Puzzle methods on each call
     - allocated on stack
  2. class variable of **Puzzle** class
     - client does not see it
     - allocated in static area
- 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:

```
//return encoding of initial state
int init();
//return number of tile at grid (r,c)
int tile(int s, int r, int c);
//move to a new state, return new encoding
int move(int s, char d);
```

Implementation using L

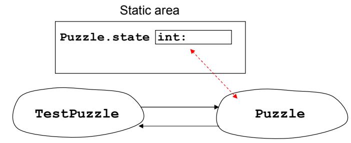
```
class TestPuzzle {

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

 public static void display(int s) {
   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(); //newline after row
   }
 }
}
```

```
class  Puzzle {

  public static int init() {
    return 123456879;
  }

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

  public static int move(int s, char d) {
    ...
  }
}
```

Critique of Interface L

- No data abstraction!
  - **`Puzzle`** class implementer chose to implement state as an **`int`**
  - This representation is exposed in the interface, so the client code is aware of it
  - Client's code may depend on this encoding
  - If **`Puzzle`** class implementer decides to change the implementatation (say, to represent state as a **`long`**), client code breaks

Interface S(tatic)



- State is implemented as class variable in class **`Puzzle`**
  - state does not have to be passed back and forth
  - representation is hidden from client
- Interface of **`Puzzle`** class:

```
void init(); //initialize the state
int tile(int r, int c); //return tile in position (r,c)
void move(char d); //move in direction d
```

Implementation using S

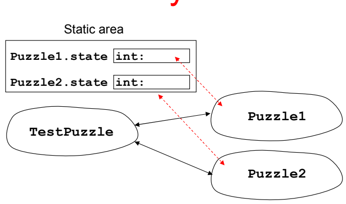
```
class TestPuzzle {

 public static void main(String[] args) {
     Puzzle.init();
     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(); //newline
   }
 }
}
```

```
class  Puzzle {
  private static int state;

  public static void init {
    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 **`int`**
  - State representation is not visible outside of **`Puzzle`** class
  - If **`Puzzle`** class implementer decides to change implementation of state to **`long`**, client code does not have to change
- Problem: only one client and one puzzle at a time
  - 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

A Sneaky Solution



- Make copies of **`Puzzle`** class and rename them
- If client wants *n* puzzles, make *n* copies

Sneaky Implementation of S

```
class TestPuzzle {

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

     Puzzle2.init();
     display2();
     Puzzle2.move('N');
     ...
 }

 public static void display1() {
   ...
   }
 }

 public static void display2() {
   ...
 }
}
```

```
class  Puzzle1 {
  private static int state;

  public static void init {
    state = 123456879;
  }

  ...
}
```

```
class  Puzzle2 {
  private static int state;

  public static void init {
    state = 123456879;
  }

  ...
}
```

Critique

- Data abstraction: yes
- Creation on demand: yes, but at cost of duplication of code
- Must know number of instances at compile time
- Naming issues

The Case for Objects

- Copying and renaming gives us
  - a unique name for each instance of the puzzle
  - a separate variable (`state`) to store the state of each instance
  - allows multiple simultaneous instances of the puzzle
- But all the instances are identical!
- Can we design language mechanisms to support the creation of separate instances?

Solution: Ask Gutenberg!

- Algorithm for making a copy of a book in the middle ages:
  - 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
- Modern algorithm (Gutenberg, Strasbourg ca.1450 AD):
  - First make a template 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 the template for puzzles?
  - How do we stamp out new puzzle instances from the template?
  - How do we name different puzzle instances?

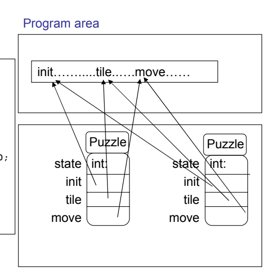
Gutenberg Bible
– The Huntington Collection

Object-Oriented Languages

- The *class definition* is the template
- Instances of the class are called *objects*
- Objects are stamped out (created) in an area of memory called the *heap*
- instance variables: when different instances are stamped out, they will each have their own copies of all instance variables (e.g. `state`)
- instance methods: code is shared among all instances of the same class, but references to instance variables in the code access those belonging to the correct object!
- constructor: a special method associated with a class invoked to create new instances of that class

Heap Allocation

```
class  Puzzle {
  private int state;

  public void init() {
    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) {
     ...
  }
}
```



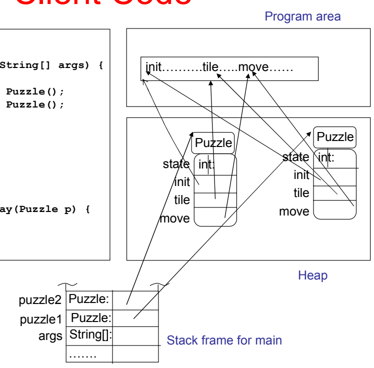
- Heap shows two instances of class **Puzzle**
- Class name is used as type of object
- Each object has its own instance variables
- Instance variables are declared private, so not accessible to client
- Compiled instance methods are stored in Program area
- All objects of type **Puzzle** share code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

Naming Instances

- Reference: a variable that is a name for objects of some class
  - contains either a pointer to some object or `null`
- Type of reference = class name
  `Puzzle p1; //declare a reference variable`
- Creation of an object using a constructor and assignment to a reference:
  `p1 = new Puzzle(); //create a new object, call it p1`
  `Puzzle p2 = new Puzzle(); //can do both at once`
- Invoking instance method
  `p1.init();`
- Implementation:
  - examine object pointed to by **p1**
  - look inside object for starting address of method named **init**
  - invoke that method

Client Code

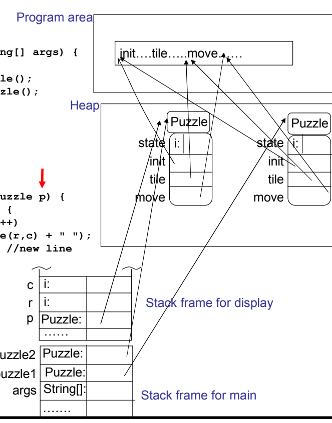
```
class TestPuzzle {

 public static void main(String[] args) {

     Puzzle puzzle1 = new Puzzle();
     Puzzle puzzle2 = new Puzzle();
     puzzle1.init();
     display(puzzle1);
     puzzle2.init();
     display(puzzle2);
     ...
 }

 public static void display(Puzzle p) {
   ...
 }
}
```



Method Invocation

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

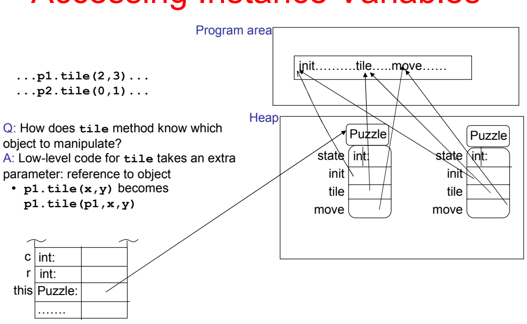
```
class TestPuzzle {

 public static void main(String[] args) {

     Puzzle puzzle1= new Puzzle();
     Puzzle puzzle2 = new Puzzle();
     puzzle1.init();
 →   display(puzzle1);
     puzzle2.init();
     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
   }
 }
}
```



Accessing Instance Variables

```
...p1.tile(2,3)...
...p2.tile(0,1)...
```

- Q: How does **tile** method know which object to manipulate?
- A: Low-level code for **tile** takes an extra parameter: reference to object
  - `p1.tile(x,y)` becomes `p1.tile(p1,x,y)`



Stack frame for invocation of tile

Keyword `this`

- In instance method, **this** is a reference to object in which the method exists

```
class TestPuzzle {

  public static void main(String[] args) {
    Puzzle puzzle1 = new Puzzle();
    puzzle1.init();
    ...
  }
}

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

```
class Puzzle {

  ...
  public void move(char d) {
    ...
    TestPuzzle.display(this);
  }
  ...
}
```

Critique

- Data abstraction: yes
- Creation on demand: yes
- Duplicate class code: no
- Duplicate client code: no

Garbage Collection

- Intuitively, an object is *live* at time *t* if that object is still in use and can be accessed by the program after time *t*
- Formally (recursive definition), an object O is *live* if:
  - The runtime stack contains a reference to O
  - There is a live object O' that contains a reference to O
- Everything else is *garbage*
- Periodically, system detects garbage and reclaims it
- Start with the stack, trace all references, mark all objects seen – anything not marked is garbage
- C, C++:
  - Pointer arithmetic makes it hard to determine what is a reference
  - Storage reclamation must be done explicitly by programmer (**`malloc`**, **`mfree`**)
  - Highly error-prone

Conclusion

- Object-oriented languages support data abstraction and code reuse
- Objects (instances of a class) can be created on demand by client without breaking abstraction
- Client can hold a reference to an object, but implementation is hidden from it
- User-defined types: class names are used as types of objects and references