Polymorphism

- Invoking a method on an object in Java
  - variable that stores the object has a static type
  - object reference is dynamic and so is its type
  - object's type must match the type of the reference variable
  - object's type may override methods of the base type

- Polymorphic Dispatch
  - target method address depends on the type of the referenced object
  - one call site can invoke different methods at different times

Polymorphic Dispatch

- Method address is determined dynamically
  - compiler can not hardcode target address in procedure call
  - instead, compiler generates code to lookup procedure address at runtime
  - address is stored in memory in the object's class jump table

Class Jump Table

- every class is represented by class object
- the class object stores the class's jump table
- the jump table stores the address of every method implemented by the class
- objects store a pointer to their class object

Static and dynamic of method invocation

- address of jump table is determined dynamically
- method's offset into jump table is determined statically

Example of Java Dispatch

- invoking ping and pong on an A and a B ...
- jump to address stored in memory using base-offset addressing

Dispatch Diagram for C (the dispatch)

- ISA for Polymorphic Dispatch
  - function signature (b+o)
  - jump to address stored in memory using base-offset addressing

Switch Statements
Switch Statement

```c
void bar () {
  if (i==0)
    j=10;
  else if (i==1) 
    j = 11;
  else if (i==2) 
    j = 12;
  ...
}
```

Human vs Compiler

- Benefits for humans
  - The syntax models a common idiom: choosing one computation from a set

- But, switch statements have interesting restrictions
  - Case labels must be static, cardinal values
  - A cardinal value is a number that specifies a position relative to the beginning of an ordered set
  - For example, integers are cardinal values, but strings are not
  - Case labels must be comparable for equality to a single dynamic expression
  - Some languages permit the expression to benow a single inequality

Do these restrictions benefit humans?

- You’ve wanted to do something like this?
  ```c
  switch (i) {
    case 0: j = 10; break;
    case 1: j = 11; break;
    case 2: j = 12; break;
    ...
  }
  ```

- Implementing switch statements
  - We already know how to implement if statements; is there anything more to consider?

The basic implementation strategy

- General form of a switch statement
  ```c
  switch (exp) {
    case label0:
      statement;
      break;
    case label1:
      statement;
      break;
    ...
    default:
      statement;
  }
  ```

- Naive implementation strategy
  ```c
  if (exp0 == exp) {
    statement;
  } else if (exp1 == exp) {
    statement;
  } ...
  ```

- Dynamic jumps
  - Jump target address is dynamic
  - Jump-table implementation running time is independent of the number of case labels

- Non-contiguous case labels
  - What is the solution?
    ```c
    static jumptable[static label]=address of code_i for all label_i
    ```

- Case labels not starting at 0
  - What is the solution?
    ```c
    static jumptable[static label]=address of code_i for all label_i
    ```

Static and Dynamic Control Flow

- Jump instructions
  - Specify a target address and a jump-taken condition
  - Target address can be static or dynamic
  - Jump-target address can be static (unconditional) or dynamic (conditional)

- Static jumps
  - Jump target address is static
  ```c
  if (exp0 == exp) {
    goto L0;
  } else if (exp1 == exp) {
    goto L1;
  } ...
  ```

- Dynamic jumps
  - Jump target address is dynamic
  ```c
  goto L{exp0} = jumptable[exp0];
  ```

Why Compilers like Switch Statements

- Notice what we have
  ```c
  switch (i) {
    case 0: j = 10; break;
    case 1: j = 11; break;
    case 2: j = 12; break;
    ...
  }
  ```

- And so, the implementation has a simplified form
  - Build a table with the address of each case arm, indexed by case value
  - Switch by indexing into this table and jumping to matching case arm

- For example
  ```c
  switch (i) {
    case 0: j = 10; break;
    case 1: j = 11; break;
    case 2: j = 12; break;
    ...
  }
  ```

Implementing Switch Statements

- Choose strategy
  - Use jump-table unless case labels are sparse or there are very few of them
  ```c
  static jumpTable = { L0, L1, L2, L3 }
  ```

- Use nested-if statements otherwise

- Jump-table strategy
  - Static:
    - Build jump table for all label values between lowest and highest
  - Generate code to:
    - Give default if label is less than minimum case label or greater than maximum case label
    - Normalize condition to lowest case label
    - Use jump to go directly to code selected case arm

- Case labels starting at 0
  ```c
  goto L{exp0} = jumptable[exp0];
  ```

Static vs Dynamic flow control

- Static if jump target is known by compiler
- Dynamic for polymorphic dispatch, function pointers, and switch statements

Polymorphic Dispatch in Java

- Invoking a method on an object in Java
- Method address depends on object’s types, which is not known statically
- Object has pointer to class object; class object contains method jump table
- Procedure call is a double-indirect jump – i.e., target address in memory

Function Pointers in C

- A variable that stores an address of a procedure
- Used to implement dynamic procedure calls, similar to polymorphic dispatch
- Switch statements
  - Syntax restricted so that they can be implemented with jump table
  - Jump-table implementation running time is independent of the number of case labels
  - But, only works if case label values are reasonably dense