

Programming Language Concepts: Lecture 1

Madhavan Mukund

Chennai Mathematical Institute

`madhavan@cmi.ac.in`

`http://www.cmi.ac.in/~madhavan/courses/pl2009`

PLC 2009, Lecture 1, 12 January 2009

Data and datatypes

- ▶ Programs manipulate data
- ▶ Basic built in data types
 - ▶ `Int`, `Float`, `Char`, ...
- ▶ Built in collective datatypes
 - ▶ Arrays, lists, ...
 - ▶ Choice depends on underlying architecture
 - ▶ Random access arrays for traditional von Neumann machines
 - ▶ Lists for functional programming
- ▶ Many useful data structures
 - ▶ Stacks, queues, trees, ...
- ▶ Programming language cannot anticipate all requirements

User defined datatypes

- ▶ Stack in C

```
int s[100];  
int tos = 0; /* points to top of stack */
```

- ▶ Should not be able to access `s[5]` if `tos == 7`

- ▶ Abstract datatype

- ▶ Data organization in terms of how the data in the data structure can be manipulated
- ▶ Implementation should not allow user to circumvent this

User defined datatypes

- ▶ Stack in C

```
int s[100];  
int tos = 0; /* points to top of stack */
```

- ▶ Should not be able to access `s[5]` if `tos == 7`
- ▶ Abstract datatype
 - ▶ Data organization in terms of how the data in the data structure can be manipulated
 - ▶ Implementation should not allow user to circumvent this
- ▶ Can we enforce this rather than depend on programmer discipline?

Class

Classes, [Simula, 1967]

- ▶ The word “class” is not very significant

Class

Classes, [Simula, 1967]

- ▶ The word “class” is not very significant

Class definition has two parts

- ▶ How the data is stored in this type.
- ▶ What functions are available to manipulate this data.

Stack as a class

```
class stack {  
    int values[100];    /* values stored in an array */  
    int tos = 0;        /* top of stack, initially 0 */  
  
    push (int i, ...){  /* push i onto stack */  
        values[tos] = i;  
        tos = tos+1;    /* Should check tos < 100!! */  
    }  
  
    int pop (...){      /* pop and return top of stack */  
        tos = tos - 1;  /* Should check tos > 0!! */  
        return values[tos];  
    }  
  
    bool is_empty (...){ /* is the stack empty? */  
        return (tos == 0); /* yes iff tos is 0 */  
    }  
}
```

Classes

- ▶ Traditionally, we pass data to functions
 - ▶ `push(s,i) /* stack s, data i */`

Classes

- ▶ Traditionally, we pass data to functions
 - ▶ `push(s,i) /* stack s, data i */`
- ▶ Instead, instantiate **classes** as **objects**, each with a private copy of functions

```
stack s,t;      /* References to stack */

s = new stack;  /* Create one stack ... */
t = new stack;  /* ... and another      */
s.push(7);
```

Classes

- ▶ Traditionally, we pass data to functions

- ▶ `push(s,i) /* stack s, data i */`

- ▶ Instead, instantiate **classes** as **objects**, each with a private copy of functions

```
stack s,t;      /* References to stack */
```

```
s = new stack;  /* Create one stack ... */
```

```
t = new stack;  /* ... and another      */
```

```
s.push(7);
```

- ▶ This creates only one object with two “names”

```
s = new stack;  /* Create one stack ... */
```

```
t = s;          /* ... assign another name */
```

Classes ...

- ▶ In our class definition, the data to be passed to a function is implicit
- ▶ Each function is implicitly attached to an object, and works on that object

```
i = s.pop();  
if (t.is_empty()) {...}
```

No ... in arguments to functions

```
class stack {  
    int values[100];        /* values stored in an array */  
    int tos = 0;            /* top of stack, initially 0 */  
  
    push(int i){            /* push i onto stack */  
        values[tos] = i;  
        tos = tos+1;        /* Should check tos < 100!! */  
    }  
  
    int pop(){              /* pop and return top of stack */  
        tos = tos - 1;      /* Should check tos > 0!! */  
        return values[tos];  
    }  
  
    bool is_empty(){        /* is the stack empty? */  
        return (tos == 0);  /* yes iff tos is 0      */  
    }  
}
```

Classes and objects

- ▶ An object is an instance of a class
- ▶ Traditionally, functions are more “fundamental” than data
- ▶ Here, functionality is implicitly tied to data representation

Classes and objects

- ▶ An object is an instance of a class
- ▶ Traditionally, functions are more “fundamental” than data
- ▶ Here, functionality is implicitly tied to data representation
- ▶ OO terminology
 - ▶ Internal variables — instance variables, fields
 - ▶ Functions — methods

Public vs private

- ▶ Implementation details should be private

Public vs private

- ▶ Implementation details should be private

```
class date {  
    int day, month, year;  
}
```

- ▶ How do we read and set values for `date` objects?
- ▶ Functions `getdate` and `setdate`
 - ▶ Accessor and mutator methods

Public vs private

- ▶ Implementation details should be private

```
class date {  
    int day, month, year;  
}
```

- ▶ How do we read and set values for `date` objects?
- ▶ Functions `getdate` and `setdate`
 - ▶ Accessor and mutator methods
- ▶ Programmers are lazy!
- ▶ Allow access to internal variables of an object

```
if (s.tos == 0){ ... }
```

Public vs private

- ▶ To restore data integrity, classify internals as `public` or `private`

```
class stack{  
    private int values[100];  
    private int tos = 0;  
    ...  
}
```

Public vs private

- ▶ To restore data integrity, classify internals as `public` or `private`

```
class stack{  
    private int values[100];  
    private int tos = 0;  
    ...  
}
```

- ▶ Should private variables be visible to other objects of the same class?

Public vs private

- ▶ To restore data integrity, classify internals as `public` or `private`

```
class stack{  
    private int values[100];  
    private int tos = 0;  
    ...  
}
```

- ▶ Should private variables be visible to other objects of the same class?
- ▶ Does it make sense to have private methods?

Private methods?

```
class stack {  
    ...  
    push (int i){    /* push i onto stack */  
        if (stack_full){  
            extend_stack();  
        }  
        ...          /* Code to add i to stack * /  
    }  
  
    extend_stack(){  
        ... /* Code to get additional space for stack data */  
    }  
    ...  
}
```

Static components

- ▶ All functions defined in classes
- ▶ Classes have to be instantiated
- ▶ Where does computation begin?

Static components

- ▶ All functions defined in classes
- ▶ Classes have to be instantiated
- ▶ Where does computation begin?
- ▶ Need functions that exist without instantiating a class
 - ▶ `static` functions
- ▶ Also useful for library functions
 - ▶ `IO.read(), IO.write(...)`

Static components

- ▶ All functions defined in classes
- ▶ Classes have to be instantiated
- ▶ Where does computation begin?
- ▶ Need functions that exist without instantiating a class
 - ▶ `static` functions
- ▶ Also useful for library functions
 - ▶ `IO.read(), IO.write(...)`
- ▶ Also static fields

```
class Math {  
    public static double PI = 3.1415927;  
    public static double E  = 2.7182818;  
    public static double sin(double x) { ... }  
    ...  
}
```


Private static?

Does a combination of private and static make sense?

```
class interest-rate {  
    private static double base_rate = 7.32;  
  
    private double deposit-amount;  
  
    public double sixmonth-yield(){ ... }  
        /* uses base-rate and deposit-amount */  
  
    public double oneyear-yield(){ ... }  
        /* uses base-rate and deposit-amount */  
    ...  
}
```

Static fields and methods

- ▶ Static entities exist before any objects are created
- ▶ Static fields are shared across objects

Static fields and methods

- ▶ Static entities exist before any objects are created
- ▶ Static fields are shared across objects

```
class stack {  
    ...  
    private static int num_push = 0;  
        /* number of pushes across all  stacks */  
  
    push (int i, ...){  
        ...  
        num_push++;      /* update static variable */  
        ...  
    }  
    ...  
}
```

Static fields and methods

- ▶ Static entities exist before any objects are created
- ▶ Static fields are shared across objects

```
class stack {  
    ...  
    private static int num_push = 0;  
        /* number of pushes across all stacks */  
  
    push (int i, ...){  
        ...  
        num_push++;      /* update static variable */  
        ...  
    }  
    ...  
}
```

- ▶ Static methods should not refer to non-static fields

Constants

```
class Math {  
    public static double PI = 3.1415927;  
    ...  
}
```

User can modify `PI`!

Constants

```
class Math {  
    public static double PI = 3.1415927;  
    ...  
}
```

User can modify `PI`!

Declare `PI` to be `final`

```
class Math {  
    public static final double PI = 3.1415927;  
    ...  
}
```

What could it mean for a function to be `final`?

Java basics

- ▶ Java program : collection of classes
- ▶ Each class `xyz` in a separate file `xyz.java`

Java basics

- ▶ Java program : collection of classes
- ▶ Each class `xyz` in a separate file `xyz.java`
- ▶ To start the computation: one class must have a static method

```
public static void main(String[] args)
```

- ▶ `void` is the return type
- ▶ `String[] args` refers to command line arguments

Java basics

- ▶ Java program : collection of classes
- ▶ Each class `xyz` in a separate file `xyz.java`
- ▶ To start the computation: one class must have a static method

```
public static void main(String[] args)
```

- ▶ `void` is the return type
 - ▶ `String[] args` refers to command line arguments
- ▶ Java programs are usually interpreted on **Java Virtual Machine**

Java basics

- ▶ Java program : collection of classes
- ▶ Each class `xyz` in a separate file `xyz.java`
- ▶ To start the computation: one class must have a static method

```
public static void main(String[] args)
```

- ▶ `void` is the return type
 - ▶ `String[] args` refers to command line arguments
- ▶ Java programs are usually interpreted on **Java Virtual Machine**
- ▶ `javac` compiles Java into **bytecode** for JVM
 - ▶ `javac xyz.java` creates “class” file `xyz.class`

Java basics

- ▶ Java program : collection of classes
- ▶ Each class `xyz` in a separate file `xyz.java`
- ▶ To start the computation: one class must have a static method

```
public static void main(String[] args)
```

- ▶ `void` is the return type
 - ▶ `String[] args` refers to command line arguments
- ▶ Java programs are usually interpreted on **Java Virtual Machine**
- ▶ `javac` compiles Java into **bytecode** for JVM
 - ▶ `javac xyz.java` creates “class” file `xyz.class`
- ▶ `java xyz` interprets and runs bytecode in class file

Hello world

```
class helloworld{  
    public static void main(String[] args){  
        System.out.println("Hello world!");  
    }  
}
```

Hello world

```
class helloworld{  
    public static void main(String[] args){  
        System.out.println("Hello world!");  
    }  
}
```

- Store in `helloworld.java`

Hello world

```
class helloworld{  
    public static void main(String[] args){  
        System.out.println("Hello world!");  
    }  
}
```

- ▶ Store in `helloworld.java`
- ▶ `javac helloworld.java` to compile to bytecode
 - ▶ Creates `helloworld.class`

Hello world

```
class helloworld{  
    public static void main(String[] args){  
        System.out.println("Hello world!");  
    }  
}
```

- ▶ Store in `helloworld.java`
- ▶ `javac helloworld.java` to compile to bytecode
 - ▶ Creates `helloworld.class`
- ▶ `java helloworld` to execute

Hello world

```
class helloworld{  
    public static void main(String[] args){  
        System.out.println("Hello world!");  
    }  
}
```

- ▶ Store in `helloworld.java`
- ▶ `javac helloworld.java` to compile to bytecode
 - ▶ Creates `helloworld.class`
- ▶ `java helloworld` to execute
- ▶ **Note:**
 - ▶ `javac` requires extension `.java`
 - ▶ `java` should not be provided `.class`
 - ▶ `javac` automatically follows dependencies and compiles all classes required