Classes, objects, Java

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 3, 16 January 2024

Programming with objects

- Object are like abstract datatypes
  - Hidden data with set of public operations
  - All interaction through operations — messages, methods, member-functions, ...
- Class
  - Template for a data type
  - How data is stored
  - How public functions manipulate data
- Object
  - Concrete instance of template
  - Each object maintains a separate copy of local data
  - Invoke methods on objects — send a message to the object

Important features

- Abstraction
  - Public interface, private implementation
  - Control external access to internal details

Important features

- Abstraction
  - Public interface, private implementation
  - Control external access to internal details
- Subtyping
  - `A` subtype of `B` $\Rightarrow$ whenever object of type `B` is needed, object of type `A` can be used
  - Compatibility of interfaces

Important features

- Abstraction
  - Public interface, private implementation
  - Control external access to internal details
- Subtyping
  - `A` subtype of `B` $\Rightarrow$ whenever object of type `B` is needed, object of type `A` can be used
  - Compatibility of interfaces
- Inheritance
  - Extend functionality of a class, reuse of implementations

Important features

- Abstraction
  - Public interface, private implementation
  - Control external access to internal details
- Subtyping
  - `A` subtype of `B` $\Rightarrow$ whenever object of type `B` is needed, object of type `A` can be used
  - Compatibility of interfaces
- Inheritance
  - Extend functionality of a class, reuse of implementations
- Dynamic lookup
  - Whether a method can be invoked on an object is a static property — type-checking
  - How the method acts is a dynamic property of how the object is implemented

Objects in Python

- Need a mechanism to hide private implementation details
  - Declare component private or public

Objects in Python

- Need a mechanism to hide private implementation details
  - Declare component private or public
- Working within privacy constraints
  - Class `Square` extends `Rectangle`
  - Instance variables of `Rectangle` are private
  - How can the constructor for `Square` set these private variables?
  - `Square` doesn't (and shouldn't) know the names of the private instance variables

Objects in Python

- Need a mechanism to hide private implementation details
  - Declare component private or public
- Working within privacy constraints
  - Class `Square` extends `Rectangle`
  - Instance variables of `Rectangle` are private
  - How can the constructor for `Square` set these private variables?
  - `Square` doesn't (and shouldn't) know the names of the private instance variables
- Need to have elaborate declarations
  - Type and visibility of variables

Objects in Python

- Need a mechanism to hide private implementation details
  - Declare component private or public
- Working within privacy constraints
  - Class `Square` extends `Rectangle`
  - Instance variables of `Rectangle` are private
  - How can the constructor for `Square` set these private variables?
  - `Square` doesn't (and shouldn't) know the names of the private instance variables
- Need to have elaborate declarations
  - Type and visibility of variables
- Static type checking catches errors early

Getting started with Java

The C Programming Language, Brian W Kernighan, Dennis M Ritchie

The only way to learn a new programming language is by writing programs in it. The first program is the same for all languages.

Print the words

```
hello, world
```

This is a big hurdle; to leap over it you have to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy

Getting started with Java

The C Programming Language, Brian W Kernighan, Dennis M Ritchie

The only way to learn a new programming language is by writing programs in it. The first program is the same for all languages.

Print the words

```
hello, world
```

This is a big hurdle; to leap over it you have to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy

- In Python

```
print("hello, world")
```

Getting started with Java

The C Programming Language, Brian W Kernighan, Dennis M Ritchie

The only way to learn a new programming language is by writing programs in it. The first program is the same for all languages.

Print the words

```
hello, world
```

This is a big hurdle; to leap over it you have to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy

- In Python

```
print("hello, world")
```

- . . . C

```
#include <stdio.h>
main()
{
  printf("hello, world\n");
}
```

Getting started with Java

The C Programming Language, Brian W Kernighan, Dennis M Ritchie

The only way to learn a new programming language is by writing programs in it. The first program is the same for all languages.

Print the words

```
hello, world
```

This is a big hurdle; to leap over it you have to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy

- In Python

```
print("hello, world")
```

- . . . C

```
#include <stdio.h>
main()
{
  printf("hello, world\n");
}
```

- . . . and Java

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

Why so complicated?

- Let's unpack the syntax

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

Why so complicated?

- Let's unpack the syntax
- All code in Java lives within a class
  - No free floating functions, unlike Python and other languages
  - Modifier `public` specifies visibility

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

Why so complicated?

- Let's unpack the syntax
- All code in Java lives within a class
  - No free floating functions, unlike Python and other languages
  - Modifier `public` specifies visibility
- How does the program start?
  - Fix a function name that will be called by default
  - From C, the convention is to call this function `main()`

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

Why so complicated . . .

- Need to specify input and output types for `main()`
  - The signature of `main()`
  - Input parameter is an array of strings; command line arguments
  - No output, so return type is `void`

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

Why so complicated . . .

- Need to specify input and output types for `main()`
  - The signature of `main()`
  - Input parameter is an array of strings; command line arguments
  - No output, so return type is `void`
- Visibility
  - Function has be available to run from outside the class
  - Modifier `public`

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

Why so complicated . . .

- Availability
  - Functions defined inside classes are attached to objects
  - How can we create an object before starting?
  - Modifier `static` — function that exists independent of dynamic creation of objects

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

Why so complicated . . .

- The actual operation
  - `System` is a public class

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

Why so complicated . . .

- The actual operation
  - `System` is a public class
  - `out` is a stream object defined in `System`
    - Like a file handle
    - Note that `out` must also be `static`

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

Why so complicated . . .

- The actual operation
  - `System` is a public class
  - `out` is a stream object defined in `System`
    - Like a file handle
    - Note that `out` must also be `static`
  - `println()` is a method associated with streams
    - Prints argument with a newline, like Python `print()`

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

Why so complicated . . .

- The actual operation
  - `System` is a public class
  - `out` is a stream object defined in `System`
    - Like a file handle
    - Note that `out` must also be `static`
  - `println()` is a method associated with streams
    - Prints argument with a newline, like Python `print()`

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

- Punctuation {, }, ; to delimit blocks, statements
  - Unlike layout and indentation in Python

Compiling and running Java code

- A Java program is a collection of classes

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

Compiling and running Java code

- A Java program is a collection of classes
- Each class is defined in a separate file with the same name, with extension `java`
  - Class `helloworld` in `helloworld.java`

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

Compiling and running Java code

- A Java program is a collection of classes
- Each class is defined in a separate file with the same name, with extension `java`
  - Class `helloworld` in `helloworld.java`

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

- Java programs are usually interpreted on Java Virtual Machine (JVM)
  - JVM provides a uniform execution environment across operating systems
  - Semantics of Java is defined in terms of JVM, OS-independent
  - "Write once, run anywhere"

Compiling and running Java code

- `javac` compiles into JVM bytecode
  - `javac helloworld.java` creates bytecode file `helloworld.class`

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

Compiling and running Java code

- `javac` compiles into JVM bytecode
  - `javac helloworld.java` creates bytecode file `helloworld.class`
- `java helloworld` interprets and runs bytecode in `helloworld.class`

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

Compiling and running Java code

- `javac` compiles into JVM bytecode
  - `javac helloworld.java` creates bytecode file `helloworld.class`
- `java helloworld` interprets and runs bytecode in `helloworld.class`

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

- Note:
  - `javac` requires file extension `.java`
  - `java` should not be provided file extension `.class`
  - `javac` automatically follows dependencies and compiles all classes required
    - Sufficient to trigger compilation for class containing `main()`

Scalar types

- In an object-oriented language, all data should be encapsulated as objects

Scalar types

- In an object-oriented language, all data should be encapsulated as objects
- However, this is cumbersome
  - Useful to manipulate numeric values like conventional languages

Scalar types

- In an object-oriented language, all data should be encapsulated as objects
- However, this is cumbersome
  - Useful to manipulate numeric values like conventional languages
- Java has eight primitive scalar types
  - `int`, `long`, `short`, `byte`
  - `float`, `double`
  - `char`
  - `boolean`

Scalar types

- In an object-oriented language, all data should be encapsulated as objects
- However, this is cumbersome
  - Useful to manipulate numeric values like conventional languages
- Java has eight primitive scalar types
  - `int`, `long`, `short`, `byte`
  - `float`, `double`
  - `char`
  - `boolean`
- Size of each type is fixed by JVM
  - Does not depend on native architecture

Type	Size in bytes
`int`	4
`long`	8
`short`	2
`byte`	1
`float`	4
`double`	8
`char`	2
`boolean`	1

Scalar types

- In an object-oriented language, all data should be encapsulated as objects
- However, this is cumbersome
  - Useful to manipulate numeric values like conventional languages
- Java has eight primitive scalar types
  - `int`, `long`, `short`, `byte`
  - `float`, `double`
  - `char`
  - `boolean`
- Size of each type is fixed by JVM
  - Does not depend on native architecture

Type	Size in bytes
`int`	4
`long`	8
`short`	2
`byte`	1
`float`	4
`double`	8
`char`	2
`boolean`	1

- 2-byte `char` for Unicode

Declarations, assigning values

- We declare variables before we use them

```
int x, y;
double y;
char c;
boolean b1, b2;
```

  - Note the semicolons after each statement

Declarations, assigning values

- We declare variables before we use them

```
int x, y;
double y;
char c;
boolean b1, b2;
```

  - Note the semicolons after each statement

- The assignment statement works as usual

```
int x,y;
x = 5;
y = 7;
```

Declarations, assigning values

- We declare variables before we use them

```
int x, y;
double y;
char c;
boolean b1, b2;
```

  - Note the semicolons after each statement

- The assignment statement works as usual

```
int x,y;
x = 5;
y = 7;
```

- Characters are written with single-quotes (only)

```
char c,d;

c = 'x';
d = '\u03C0'; // Greek pi, unicode
```

  - Double quotes denote strings

Declarations, assigning values

- We declare variables before we use them

```
int x, y;
double y;
char c;
boolean b1, b2;
```

  - Note the semicolons after each statement

- The assignment statement works as usual

```
int x,y;
x = 5;
y = 7;
```

- Characters are written with single-quotes (only)

```
char c,d;

c = 'x';
d = '\u03C0'; // Greek pi, unicode
```

  - Double quotes denote strings

- Boolean constants are `true`, `false`

```
boolean b1, b2;

b1 = false;
b2 = true;
```

Initialization, constants

- Declarations can come anywhere

```
int x;
x = 10;
double y;
```

  - Use this judiciously to retain readability

Initialization, constants

- Declarations can come anywhere

```
int x;
x = 10;
double y;
```

  - Use this judiciously to retain readability

- Initialize at time of declaration

```
int x = 10;
double y = 5.7;
```

Initialization, constants

- Declarations can come anywhere

```
int x;
x = 10;
double y;
```

  - Use this judiciously to retain readability

- Initialize at time of declaration

```
int x = 10;
double y = 5.7;
```

- Can we declare a value to be a constant?

```
float pi = 3.1415927f;

pi = 22/7;  // Disallow?
```

  - Note: Append `f` after number for `float`, else interpreted as `double`

Initialization, constants

- Declarations can come anywhere

```
int x;
x = 10;
double y;
```

  - Use this judiciously to retain readability

- Initialize at time of declaration

```
int x = 10;
double y = 5.7;
```

- Can we declare a value to be a constant?

```
float pi = 3.1415927f;

pi = 22/7;  // Disallow?
```

  - Note: Append `f` after number for `float`, else interpreted as `double`

- Modifier `final` indicates a constant

```
final float pi = 3.1415927f;

pi = 22/7;  // Flagged as error;
```

Operators, shortcuts, type casting

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`

Operators, shortcuts, type casting

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`
- No separate integer division operator `//`
- When both arguments are integer, `/` is integer division

```
float f = 22/7;    // Value is 3.0
```

  - Note implicit conversion from `int` to `float`

Operators, shortcuts, type casting

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`
- No separate integer division operator `//`
- When both arguments are integer, `/` is integer division

```
float f = 22/7;    // Value is 3.0
```

  - Note implicit conversion from `int` to `float`
- No exponentiation operater, use `Math.pow()`
- `Math.pow(a,n)` returns a^n

Operators, shortcuts, type casting

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`
- No separate integer division operator `//`
- When both arguments are integer, `/` is integer division

```
float f = 22/7;   // Value is 3.0
```

  - Note implicit conversion from `int` to `float`
- No exponentiation operater, use `Math.pow()`
- `Math.pow(a,n)` returns a^n
- Special operators for incrementing and decrementing integers

```
int a = 0, b = 10;
a++;   // Same as a = a+1
b--;   // Same as b = b-1
```

Operators, shortcuts, type casting

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`
- No separate integer division operator `//`
- When both arguments are integer, `/` is integer division

```
float f = 22/7;    // Value is 3.0
```

  - Note implicit conversion from `int` to `float`
- No exponentiation operater, use `Math.pow()`
- `Math.pow(a,n)` returns a^n
- Special operators for incrementing and decrementing integers

```
int a = 0, b = 10;
a++;    // Same as a = a+1
b--;    // Same as b = b-1
```

- Shortcut for updating a variable

```
int a = 0, b = 10;
a += 7;     // Same as a = a+7
b *= 12;    // Same as b = b*12
```

Strings

- `String` is a built in class

```
String s,t;
```

Strings

- `String` is a built in class

```
String s,t;
```

- String constants enclosed in double quotes

```
String s = "Hello", t = "world";
```

Strings

- `String` is a built in class

```
String s,t;
```

- String constants enclosed in double quotes

```
String s = "Hello", t = "world";
```

- `+` is overloaded for string concatenation

```
String s = "Hello";
String t = "world";
String u = s + " " + t;
  // "Hello world"
```

Strings

- `String` is a built in class

```
String s,t;
```

- String constants enclosed in double quotes

```
String s = "Hello", t = "world";
```

- `+` is overloaded for string concatenation

```
String s = "Hello";
String t = "world";
String u = s + " " + t;
  // "Hello world"
```

- Strings are not arrays of characters
  - Cannot write

```
s[3] = 'p';
s[4] = '!';
```

Strings

- `String` is a built in class

```
String s,t;
```

- String constants enclosed in double quotes

```
String s = "Hello", t = "world";
```

- `+` is overloaded for string concatenation

```
String s = "Hello";
String t = "world";
String u = s + " " + t;
  // "Hello world"
```

- Strings are not arrays of characters
  - Cannot write

```
s[3] = 'p';
s[4] = '!';
```

- Instead, invoke method `substring` in class `String`
  - `s = s.substring(0,3) + "p!";`

Strings

- `String` is a built in class

```
String s,t;
```

- String constants enclosed in double quotes

```
String s = "Hello", t = "world";
```

- `+` is overloaded for string concatenation

```
String s = "Hello";
String t = "world";
String u = s + " " + t;
  // "Hello world"
```

- Strings are not arrays of characters
  - Cannot write

```
s[3] = 'p';
s[4] = '!';
```

s.substring(3,1) = '6';

- Instead, invoke method `substring` in class `String`
  - `s = s.substring(0,3) + "p!";`
- If we change a `String`, we get a new object
  - After the update, `s` points to a new `String`
  - Java does automatic garbage collection

Arrays

- Arrays are also objects

Arrays

- Arrays are also objects
- Typical declaration

```
int[] a;
a = new int[100];
```

  - Or `int a[]` instead of `int[] a`
  - Combine as `int[] a = new int[100];`

int[] a;
int b;

int a[], b;

Arrays

- Arrays are also objects
- Typical declaration

```
int[] a;
a = new int[100];
```

  - Or `int a[]` instead of `int[] a`
  - Combine as `int[] a = new int[100];`
- `a.length` gives size of `a`
  - Note, for `String`, it is a method `s.length()`!

Arrays

- Arrays are also objects
- Typical declaration

```
int[] a;
a = new int[100];
```

  - Or `int a[]` instead of `int[] a`
  - Combine as `int[] a = new int[100];`
- `a.length` gives size of `a`
  - Note, for `String`, it is a method `s.length()`!
- Array indices run from `0` to `a.length-1`

Arrays

- Arrays are also objects
- Typical declaration

```
int[] a;
a = new int[100];
```

  - Or `int a[]` instead of `int[] a`
  - Combine as `int[] a = new int[100];`
- `a.length` gives size of `a`
  - Note, for `String`, it is a method `s.length()`!
- Array indices run from `0` to `a.length-1`
- Size of the array can vary

Arrays

- Arrays are also objects
- Typical declaration

```
int[] a;
a = new int[100];
```

  - Or `int a[]` instead of `int[] a`
  - Combine as `int[] a = new int[100];`
- `a.length` gives size of `a`
  - Note, for `String`, it is a method `s.length()`!
- Array indices run from `0` to `a.length-1`
- Size of the array can vary
- Array constants: `{v1, v2, v3}`

Arrays

- Arrays are also objects
- Typical declaration

```
int[] a;
a = new int[100];
```

  - Or `int a[]` instead of `int[] a`
  - Combine as `int[] a = new int[100];`
- `a.length` gives size of `a`
  - Note, for `String`, it is a method `s.length()`!
- Array indices run from `0` to `a.length-1`
- Size of the array can vary
- Array constants: `{v1, v2, v3}`
- For example

```
int[] a;
int n;

n = 10;
a = new int[n];

n = 20;
a = new int[n];

a = {2, 3, 5, 7, 11};
```

Control flow

- Program layout
  - Statements end with semi-colon
  - Blocks of statements delimited by braces

Control flow

- Program layout
  - Statements end with semi-colon
  - Blocks of statements delimited by braces
- Conditional execution
  - `if (condition) { ... } else { ... }`

Control flow

- Program layout
  - Statements end with semi-colon
  - Blocks of statements delimited by braces
- Conditional execution
  - `if (condition) { ... } else { ... }`
- Conditional loops
  - `while (condition) { ... }`
  - `do { ... } while (condition)`

Control flow

- Program layout
  - Statements end with semi-colon
  - Blocks of statements delimited by braces
- Conditional execution
  - `if (condition) { ... } else { ... }`
- Conditional loops
  - `while (condition) { ... }`
  - `do { ... } while (condition)`
- Iteration
  - Two kinds of `for`

Control flow

- Program layout
  - Statements end with semi-colon
  - Blocks of statements delimited by braces
- Conditional execution
  - `if (condition) { ... } else { ... }`
- Conditional loops
  - `while (condition) { ... }`
  - `do { ... } while (condition)`
- Iteration
  - Two kinds of `for`
- Multiway branching – `switch`

Conditional execution

- `if (c) {...} else {...}`
  - `else` is optional
  - Condition must be in parentheses
  - If body is a single statement, braces are not needed
- No `elif`, à la Python
  - Indentation is not forced
  - Just align `else if`
  - Nested `if` is a single statement, no separate braces required
- No surprises
- Aside: no `def` for function definition

```
public class MyClass {

  ...

  public static int sign(int v) {
    if (v < 0) {
      return(-1);
    } else if (v > 0) {
      return(1);
    } else {
      return(0);
    }
  }

}
```

Conditional loops

- `while (c) {...}`
  - Condition must be in parentheses
  - If body is a single statement, braces are not needed

```
public class MyClass {

  ...

  public static int sumupto(int n) {
    int sum = 0;

    while (n > 0){
      sum += n;
      n--;
    }

    return(sum);
  }

}
```

Conditional loops

- `while (c) {...}`
  - Condition must be in parentheses
  - If body is a single statement, braces are not needed
- `do {...} while (c)`
  - Condition is checked at the end of the loop
  - At least one iteration

```
public class MyClass {

  ...

  public static int sumupto(int n) {
    int sum = 0;
    int i = 0;

    do {
      sum += i;
      i++;
    } while (i <= n);

    return(sum);
  }

}
```

Conditional loops

- `while (c) {...}`
  - Condition must be in parentheses
  - If body is a single statement, braces are not needed
- `do {...} while (c)`
  - Condition is checked at the end of the loop
  - At least one iteration
  - Useful for interactive user input

```
do {
  read input;
} while (input-condition);
```

```
public class MyClass {

  ...

  public static int sumupto(int n) {
    int sum = 0;
    int i = 0;

    do {
      sum += i;
      i++;
    } while (i <= n);

    return(sum);
  }

}
```

Iteration

- `for` loop is inherited from C
- `for (init; cond; upd) {...}`
  - `init` is initialization
  - `cond` is terminating condition
  - `upd` is update

Iteration

- `for` loop is inherited from C
- `for (init; cond; upd) {...}`
  - `init` is initialization
  - `cond` is terminating condition
  - `upd` is update
- Intended use is
  `for(i = 0; i < n; i++){...}`

```
public class MyClass {

  ...

  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;
    int i;

    for (i = 0; i < n; i++){
      sum += a[i];
    }

    return(sum);
  }

}
```

Iteration

- `for` loop is inherited from C
- `for (init; cond; upd) {...}`
  - `init` is initialization
  - `cond` is terminating condition
  - `upd` is update
- Intended use is
  `for(i = 0; i < n; i++){...}`
- Completely equivalent to

```
i = 0;
while (i < n) {
  i++;
}
```

```
public class MyClass {

  ...

  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;
    int i;

    for (i = 0; i < n; i++){
      sum += a[i];
    }

    return(sum);
  }

}
```

Iteration

- Intended use is
  `for(i = 0; i < n; i++){...}`
- Completely equivalent to

```
i = 0;
while (i < n) {
  i++;
}
```

```
public class MyClass {

  ...

  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;
    int i;

    for (i = 0; i < n; i++){
      sum += a[i];
    }

    return(sum);
  }

}
```

Iteration

- Intended use is
  `for(i = 0; i < n; i++){...}`
- Completely equivalent to

```
i = 0;
while (i < n) {
  i++;
}
```

- However, not good style to write `for` instead of `while`

```
public class MyClass {

  ...

  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;
    int i;

    for (i = 0; i < n; i++){
      sum += a[i];
    }

    return(sum);
  }

}
```

Iteration

- Intended use is
  `for(i = 0; i < n; i++){...}`
- Completely equivalent to

```
i = 0;
while (i < n) {
  i++;
}
```

- However, not good style to write `for` instead of `while`
- Can define loop variable within loop
  - The scope of `i` is local to the loop
  - An instance of more general local scoping allowed in Java

```
public class MyClass {

  ...

  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;
    for (int i = 0; i < n; i++){
      sum += a[i];
    }

    return(sum);
  }

}
```