Java: Scalars, control flow, classes, inheritance

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 3, 21 January 2025

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");
  }
}
```

Scalar types

- Java is an object-oriented language
  - All data encapsulated as objects?

```
n = Integer(7)
n.add(m)
```

Scalar types

- Java is an object-oriented language
  - All data encapsulated as objects?
- Not quite

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

ASCII → Unicode ←

Variables: declarations, assignment, initialization, constants

- Declare variables before use

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

Variables: declarations, assignment, initialization, constants

- Declare variables before use
- Assign values to variables as usual

```
int x, y;
double z;
x = 5;
z = 7.0;
```

Variables: declarations, assignment, initialization, constants

- Declare variables before use
- Assign values to variables as usual
- Characters — single quotes

```
char c,d;

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

Variables: declarations, assignment, initialization, constants

- Declare variables before use
- Assign values to variables as usual
- Characters — single quotes
- Boolean constants

```
boolean b1, b2;

b1 = false;
b2 = true;
```

Variables: declarations, assignment, initialization, constants

- Declare variables before use
- Assign values to variables as usual
- Characters — single quotes
- Boolean constants
- Declarations can come anywhere

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

Variables: declarations, assignment, initialization, constants

- Declare variables before use
- Assign values to variables as usual
- Characters — single quotes
- Boolean constants
- Declarations can come anywhere
- Initialize with declaration

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

Variables: declarations, assignment, initialization, constants

- Declare variables before use
- Assign values to variables as usual
- Characters — single quotes
- Boolean constants
- Declarations can come anywhere
- Initialize with declaration
- Constants

f = float

```
float pi = 3.1415927f;
pi = 22/7;  // Disallow
```

Variables: declarations, assignment, initialization, constants

- Declare variables before use
- Assign values to variables as usual
- Characters — single quotes
- Boolean constants
- Declarations can come anywhere
- Initialize with declaration
- Constants

```
final float pi = 3.1415927f;

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

Operators, shortcuts

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

Operators, shortcuts

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

```
float f = 22/7;    // Value is 3.0
                   // Implicit conversion,
                   // int to float
```

Operators, shortcuts

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`
  - No separate integer division `//`
  - When both arguments are integer, `/` is integer division
- Exponentiation:
  `Math.pow(a,n)` returns a^n

Operators, shortcuts

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`
  - No separate integer division `//`
  - When both arguments are integer, `/` is integer division
- Exponentiation:
  `Math.pow(a,n)` returns a^n
- Special operators for incrementing and decrementing integers

$x = a++$

$++a$

$a++$

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

Operators, shortcuts

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`
  - No separate integer division `//`
  - When both arguments are integer, `/` is integer division
- Exponentiation:
  `Math.pow(a,n)` returns a^n
- Special operators for incrementing and decrementing integers
- Shortcut for updating a variable

$v = v \; op \; w$

```
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 constants within double quotes `String s = "Hello", t = "world";`

Strings

- `String` is a built in class
- String constants within double quotes
- `+` overloaded for string concatenation

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

Strings

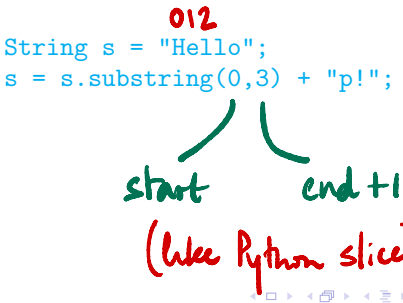
- `String` is a built in class
- String constants within double quotes
- `+` overloaded for string concatenation
- Strings are not arrays of characters

Cannot write

```
String s = "Hello";
s[3] = 'p';
s[4] = '!';
```

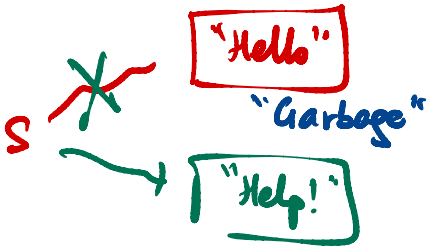
Strings

- `String` is a built in class
- String constants within double quotes
- `+` overloaded for string concatenation
- Strings are not arrays of characters
- Instead, use method `substring` in class `String`



Strings

- `String` is a built in class
- String constants within double quotes
- `+` overloaded for string concatenation
- Strings are not arrays of characters
- Instead, use method `substring` in class `String`
- If we update a `String`, we get a new object
  - Java does automatic garbage collection




```
String s = "Hello";
s = s.substring(0,3) + "p!";
```

Manual space allocation = "Memory leak"

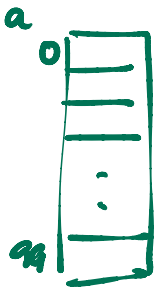
Arrays

- Arrays are also objects

Arrays

- Arrays are also objects
- Typical declarations

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



Arrays

- Arrays are also objects
- Typical declarations

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

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

Arrays

- Arrays are also objects
- Typical declarations

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

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

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

Arrays

- Arrays are also objects
- Typical declarations
- Array indices run from `0` to `a.length-1`
  - `a.length` gives size of `a`
  - For `String`, it is a method `s.length()`!

Arrays

- Arrays are also objects
- Typical declarations
- Array indices run from `0` to `a.length-1`
  - `a.length` gives size of `a`
  - For `String`, it is a method `s.length()`!
- Array constants: `{v1, v2, v3}`

Arrays

- Arrays are also objects
- Typical declarations
- Array indices run from `0` to `a.length-1`
  - `a.length` gives size of `a`
  - For `String`, it is a method `s.length()`!
- Array constants: `{v1, v2, v3}`
- Size of an array can vary dynamically

```
int[] a;
int n;

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

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

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

– a has 10 elems

– new array, 20 elems

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

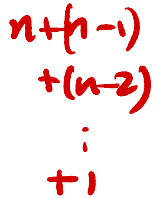
- `if (c) {...} else {...}`
  - Condition must be in parentheses
  - `else` is optional
  - No braces needed if body is single statement
- No `elif`, à la Python
  - Indentation is not forced - just align `else if`
  - Nested `if` is a single statement, no separate braces required
- No surprises
- Aside: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

```
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
  - No braces needed if body is single statement
- `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);
  }

}
```

0+1+2+ .. +n

Conditional loops

- `while (c) {...}`
  - Condition must be in parentheses
  - No braces needed if body is single statement
- `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);
  }

}
```

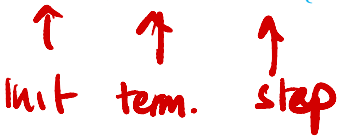
Iteration

- `for (init; cond; upd) {...}`
  - `init` — initialization
  - `cond` — terminating condition
  - `upd` — update
  - Inherited from C

Iteration

- `for (init; cond; upd) {...}`
  - `init` — initialization
  - `cond` — terminating condition
  - `upd` — update
  - Inherited from C
- 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 (init; cond; upd) {...}`
  - `init` — initialization
  - `cond` — terminating condition
  - `upd` — update
  - Inherited from C
- 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

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

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

- 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

- 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;
    int i;

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

    return(sum);
  }

}
```

logically, i is irrelevant

Iteration

- 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

i does not exist

```
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);
  }

}
```

Iterating over elements directly

- Java later introduced a `for` in the style of Python

```
for x in l:
  do something with x
```

Iterating over elements directly

- Java later introduced a `for` in the style of Python

```
for x in l:
  do something with x
```

- Again `for`, different syntax

```
for (type x : a)
  do something with x;
}
```

array of type x

x[] a

```
public class MyClass {

  ...

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

    for (int v : a){
      sum += v;
    }

    return(sum);
  }


}
```

Iterating over elements directly

- Java later introduced a `for` in the style of Python

```
for x in l:
  do something with x
```

- Again `for`, different syntax

```
for (type x : a)
  do something with x;
}
```

- In this version of `for`, the loop variable must be declared in local scope

```
public class MyClass {

  ...

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

    return(sum);
  }


}
```

Multiway branching

- `switch` selects between different options

```
public static void printsign(int v) {
  switch (v) {
    case -1: {
      System.out.println("Negative");
      break;
    }
    case 1: {
      System.out.println("Positive");
      break;
    }
    case 0: {
      System.out.println("Zero");
      break;
    }
  }
}
```

Multiway branching

- `switch` selects between different options
- Be careful, default is to "fall through" from one case to the next
  - Need to explicitly `break` out of switch
  - `break` available for loops as well
  - Check the Java documentation

```
public static void printsign(int v) {
  switch (v) {
    case -1: {
      System.out.println("Negative");
      break;
    }
    case 1: {
      System.out.println("Positive");
      break;
    }
    case 0: {
      System.out.println("Zero");
      break;
    }
  }
}
```

Multiway branching

- `switch` selects between different options
- Be careful, default is to "fall through" from one case to the next
  - Need to explicitly `break` out of switch
  - `break` available for loops as well
  - Check the Java documentation
- Options have to be constants
  - Cannot use conditional expressions

```
public static void printsign(int v) {
  switch (v) {
    case -1: {
      System.out.println("Negative");
      break;
    }
    case 1: {
      System.out.println("Positive");
      break;
    }
    case 0: {
      System.out.println("Zero");
      break;
    }
  }
}
```

Multiway branching

- `switch` selects between different options
- Be careful, default is to "fall through" from one case to the next
  - Need to explicitly `break` out of switch
  - `break` available for loops as well
  - Check the Java documentation
- Options have to be constants
  - Cannot use conditional expressions
- Aside: here return type is `void`
  - Non-`void` return type requires an appropriate `return` value

```
public static void printsign(int v) {
  switch (v) {
    case -1: {
      System.out.println("Negative");
      break;
    }
    case 1: {
      System.out.println("Positive");
      break;
    }
    case 0: {
      System.out.println("Zero");
      break;
    }
  }
}
```

Classes and objects

- A class is a template for an encapsulated type
- An object is an instance of a class
- How do we create objects?
- How are objects initialized?

Point

p.x =

p.y =

def addz(self, v):

p.z = v

Defining a class

- Definition block using `class`, with class name
  - Modifier `public` to indicate visibility
  - Java allows `public` to be omitted
  - Default visibility is public to `package`
  - Packages are administrative units of code
  - All classes defined in same directory form part of same package

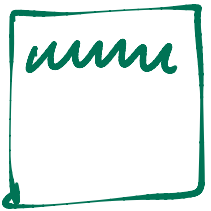
```
public class Date {
  private int day, month, year;
  ...
}
```

Defining a class

- Definition block using `class`, with class name
  - Modifier `public` to indicate visibility
  - Java allows `public` to be omitted
  - Default visibility is public to `package`
  - Packages are administrative units of code
  - All classes defined in same directory form part of same package
- Instance variables
  - Each concrete object of type `Date` will have local copies of `date`, `month`, `year`
  - These are marked `private`
  - Can also have `public` instance variables, but breaks encapsulation

```
public class Date {
  private int day, month, year;
  ...
}
```

should be default!



Creating objects

- Declare type using class name
- `new` creates a new object
  - How do we set private instance variables?

```
public void UseDate() {
  Date d;
  d = new Date();
  ...
}
```

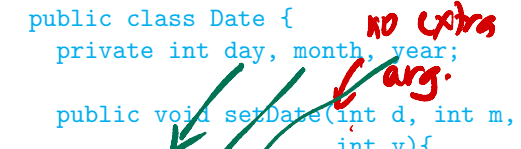
p = Point(...)

Creating objects

- Declare type using class name
- `new` creates a new object
  - How do we set private instance variables?
- Can add methods to update values
  - `this` is a reference to current object

```
public void UseDate() {
  Date d;
  d = new Date();
  ...
}
```

```
public class Date {
  private int day, month, year;

  public void setDate(int d, int m,
                      int y){
    this.day = d;
    this.month = m;
    this.year = y;
  }
}
```



Creating objects

- Declare type using class name
- `new` creates a new object
  - How do we set private instance variables?
- Can add methods to update values
  - `this` is a reference to current object
  - Can omit `this` if reference is unambiguous

```
public void UseDate() {
  Date d;
  d = new Date();
  ...
}
```

```
public class Date {
  private int day, month, year;

  public void setDate(int d, int m,
                      int y){
    this.day = d;
    month = m;
    year = y;
  }
}
```

Creating objects

- Declare type using class name
- `new` creates a new object
  - How do we set private instance variables?
- Can add methods to update values
  - `this` is a reference to current object
  - Can omit `this` if reference is unambiguous
- What if we want to check the values?
  - Methods to read and report values

```
public class Date {
  ...

  public int getDay(){
    return(day);
  }

  public int getMonth(){
    return(month);
  }

  public int getYear(){
    return(year);
  }

}
```

Creating objects

- Declare type using class name
- `new` creates a new object
  - How do we set private instance variables?
- Can add methods to update values
  - `this` is a reference to current object
  - Can omit `this` if reference is unambiguous
- What if we want to check the values?
  - Methods to read and report values
- Accessor and Mutator methods

get set

```
public class Date {
  ...

  public int getDay(){
    return(day);
  }

  public int getMonth(){
    return(month);
  }

  public int getYear(){
    return(year);
  }

}
```

Initializing objects

- Would be good to set up an object when we create it
  - Combine `new Date()` and `setDate()`

Initializing objects

- Would be good to set up an object when we create it
  - Combine `new Date()` and `setDate()`
- Constructors — special functions called when an object is created
  - Function with the same name as the class
  - `d = new Date(13,8,2015);`

```
public class Date {
  private int day, month, year;
  public Date(int d, int m, int y){
    day = d;
    month = m;
    year = y;
  }
}
```

no return value

```
def __init__(self, a=0, b=0)
:
:
```

Initializing objects

- Would be good to set up an object when we create it
  - Combine `new Date()` and `setDate()`
- Constructors — special functions called when an object is created
  - Function with the same name as the class
  - `d = new Date(13,8,2015);`
- Constructors with different signatures
  - `d = new Date(13,8);` sets `year` to 2025
  - Java allows function overloading — same name, different signatures
    - Python: default (optional) arguments, no overloading

```
public class Date {
  private int day, month, year;

  public Date(int d, int m, int y){
    day = d;
    month = m;
    year = y;
  }

  public Date(int d, int m){
    day = d;
    month = m;
    year = 2025;
  }
}
```

only arguments, not return type

Constructors . . .

- A later constructor can call an earlier one using `this`

```
public class Date {
  private int day, month, year;

  public Date(int d, int m, int y){
    day = d;
    month = m;
    year = y;
  }

  public Date(int d, int m){
    this(d,m,2025);
  }
}
```

Constructors . . .

- A later constructor can call an earlier one using `this`
- If no constructor is defined, Java provides a default constructor with empty arguments
  - `new Date()` would implicitly invoke this
  - Sets instance variables to sensible defaults
  - For instance, `int` variables set to `0`
  - Only valid if *no* constructor is defined
  - Otherwise need an explicit constructor without arguments

```
public class Date {
  private int day, month, year;

  public Date(int d, int m, int y){
    day = d;
    month = m;
    year = y;
  }

  public Date(int d, int m){
    this(d,m,2025);
  }
}
```

Subclasses

- An `Employee` class

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
```

Subclasses

- An `Employee` class
- Two private instance variables

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
```

Subclasses

- An `Employee` class
- Two private instance variables
- Some constructors to set up the object

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
```

Subclasses

- An `Employee` class
- Two private instance variables
- Some constructors to set up the object
- Accessor and mutator methods to set instance variables

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
```

Subclasses

- An `Employee` class
- Two private instance variables
- Some constructors to set up the object
- Accessor and mutator methods to set instance variables
- A public method to compute bonus

```
public class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  public double bonus(float percent){
     return (percent/100.0)*salary;
  }
}
```

Subclasses

- Managers are special types of employees with extra features

```
public class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

Subclasses

- Managers are special types of employees with extra features

```
public class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

- `Manager` objects inherit other fields and methods from `Employee`
  - Every `Manager` has a `name`, `salary` and methods to access and manipulate these.

Subclasses

- Managers are special types of employees with extra features

```
public class Manager extends Employee{
     private String secretary;
     public boolean setSecretary(name s){ ... }
     public String getSecretary(){ ... }
}
```

- `Manager` objects inherit other fields and methods from `Employee`
  - Every `Manager` has a `name`, `salary` and methods to access and manipulate these.
- `Manager` is a subclass of `Employee`
  - Think of subset

Subclasses

- `Manager` objects do not automatically have access to private data of parent class.
  - Common to extend a parent class written by someone else

Subclasses

- `Manager` objects do not automatically have access to private data of parent class.
  - Common to extend a parent class written by someone else
- How can a constructor for `Manager` set instance variables that are private to `Employee`?

Subclasses

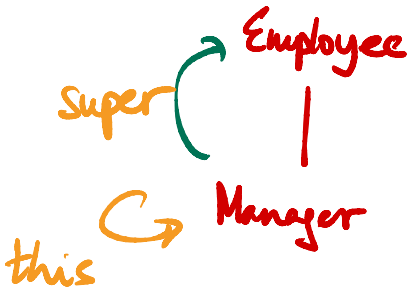
- `Manager` objects do not automatically have access to private data of parent class.
  - Common to extend a parent class written by someone else
- How can a constructor for `Manager` set instance variables that are private to `Employee`?
- Some constructors for `Employee`

```
public class Employee{
  ...
  public Employee(String n, double s){
     name = n; salary = s;
  }
  public Employee(String n){
     this(n,500.00);
  }
}
```

Subclasses

- `Manager` objects do not automatically have access to private data of parent class.
  - Common to extend a parent class written by someone else
- How can a constructor for `Manager` set instance variables that are private to `Employee`?
- Some constructors for `Employee`
- Use parent class's constructor using `super`

```
public class Employee{
  ...
  public Employee(String n, double s){
     name = n; salary = s;
  }
  public Employee(String n){
     this(n,500.00);
  }
}
```



Subclasses

- `Manager` objects do not automatically have access to private data of parent class.
  - Common to extend a parent class written by someone else
- How can a constructor for `Manager` set instance variables that are private to `Employee`?
- Some constructors for `Employee`
- Use parent class's constructor using `super`
- A constructor for `Manager`

```
public class Employee{
  ...
  public Employee(String n, double s){
     name = n; salary = s;
  }
  public Employee(String n){
     this(n,500.00);
  }
}


public class Manager extends Employee{
  ..
  public Manager(String n, double s, String sn){
     super(n,s);    /* super calls
                       Employee constructor */
     secretary = sn;
  }
}
```