Lecture 4, 20 January 2026
Java: classes, inheritance, polymorphism, [redacted]

Madhavan Mukund, S P Suresh

Programming Language Concepts
January–April 2026

Classes and objects

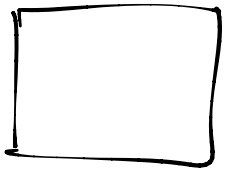
- A class is a template for an encapsulated type
- An object is an instance of a class
- How do we create objects?
- How are objects initialized?

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;
  ...
}
```

"global" declarations



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;

  ...

}
```

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(3,5)

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

vs
set Day
set Month
set Year

```
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

Unlike Python !

```
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){
    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

```
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(12,2,2019);`

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

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

no return type

__init__ in Python

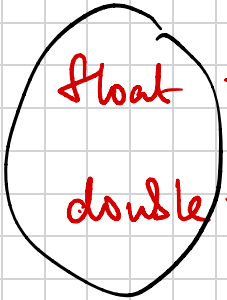
Initializing objects

- 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(12,2,2019);`
- Constructors with different signatures
  - `d = new Date(12,2);` sets `year` to 2026
  - 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 = 2026;
  }
}
```



?

not part of "signature"

double f(long a, int b)

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,2026);
  }
}
```

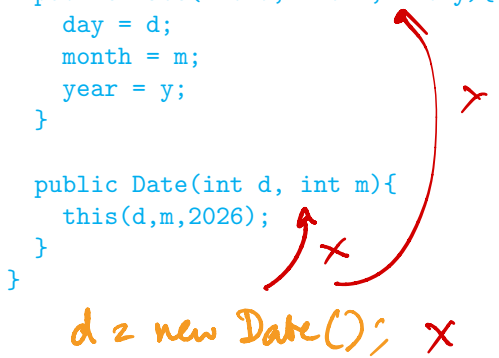
Constructors . . .

- 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,2026);
  }
}
```



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;
  }
}
```

name = s

Float

double

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(){ ... }
}
```

class Square (Rectangle)

Inheritance

- `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;
  }
}
```

Inheritance

- In general, subclass has more features than parent class
  - Subclass inherits instance variables, methods from parent class

Inheritance

- In general, subclass has more features than parent class
  - Subclass inherits instance variables, methods from parent class
- Every `Manager` is an `Employee`, but not vice versa!
- Can use a subclass in place of a superclass

  ```
  Employee e = new Manager(...)
  ```
- But the following will not work

  ```
  Manager m = new Employee(...)
  ```

```
public Date(int d, int m, int y){
}

public Date(int d, int m){
    this(d, m, 2026);
}
public Date(){
    this(1, 1);
}
```

Inheritance

- In general, subclass has more features than parent class
  - Subclass inherits instance variables, methods from parent class
- Every `Manager` is an `Employee`, but not vice versa!
- Can use a subclass in place of a superclass

  ```
  Employee e = new Manager(...)
  ```
- But the following will not work

  ```
  Manager m = new Employee(...)
  ```

- Recall
  - `int[] a = new int[100];`
  - Why the seemingly redundant reference to `int` in `new`?

Inheritance

- In general, subclass has more features than parent class
  - Subclass inherits instance variables, methods from parent class
- Every `Manager` is an `Employee`, but not vice versa!
- Can use a subclass in place of a superclass

  ```
  Employee e = new Manager(...)
  ```
- But the following will not work

  ```
  Manager m = new Employee(...)
  ```
- Recall
  - `int[] a = new int[100];`
  - Why the seemingly redundant reference to `int` in `new`?
- One can now presumably write

  ```
  Employee[] e = new Manager[100];
  ```

Dynamic dispatch

- `Manager` can redefine `bonus()`

```
double bonus(float percent){
   return 1.5*super.bonus(percent);
}
```

  - Uses parent class `bonus()` via `super`
  - Overrides definition in parent class

Overloading

Overriding

Dynamic dispatch

- `Manager` can redefine `bonus()`

```
double bonus(float percent){
   return 1.5*super.bonus(percent);
}
```

  - Uses parent class `bonus()` via `super`
  - Overrides definition in parent class

- Consider the following assignment

```
Employee e = new Manager(...)
```

- Can we invoke `e.setSecretary()`?

At compile time

Dynamic dispatch

- `Manager` can redefine `bonus()`

```
double bonus(float percent){
   return 1.5*super.bonus(percent);
}
```

  - Uses parent class `bonus()` via `super`
  - Overrides definition in parent class

- Consider the following assignment

```
Employee e = new Manager(...)
```

- Can we invoke `e.setSecretary()`?
  - `e` is declared to be an `Employee`
  - Static typechecking — `e` can only refer to methods in `Employee`

Dynamic dispatch

- `Manager` can redefine `bonus()`

```
double bonus(float percent){
   return 1.5*super.bonus(percent);
}
```

  - Uses parent class `bonus()` via `super`
  - Overrides definition in parent class

- Consider the following assignment

```
Employee e = new Manager(...)
```

- Can we invoke `e.setSecretary()`?
  - `e` is declared to be an `Employee`
  - Static typechecking — `e` can only refer to methods in `Employee`

- What about `e.bonus(p)`? Which `bonus()` do we use?
  - Static: Use `Employee.bonus()`
  - Dynamic: Use `Manager.bonus()`

Dynamic dispatch

- `Manager` can redefine `bonus()`

```
double bonus(float percent){
   return 1.5*super.bonus(percent);
}
```

  - Uses parent class `bonus()` via `super`
  - Overrides definition in parent class

- Consider the following assignment

```
Employee e = new Manager(...)
```

- Can we invoke `e.setSecretary()`?
  - `e` is declared to be an `Employee`
  - Static typechecking — `e` can only refer to methods in `Employee`

- What about `e.bonus(p)`? Which `bonus()` do we use?
  - Static: Use `Employee.bonus()`
  - Dynamic: Use `Manager.bonus()`
- Dynamic dispatch (dynamic binding, late method binding, ...) turns out to be more useful
  - Default in Java, optional in languages like C++ (`virtual` function)

Polymorphism

- Every `Employee` in `emparray` "knows" how to calculate its `bonus` correctly!

```
Employee[] emparray = new Employee[2];
Employee e = new Employee(...);
Manager m = new Manager(...);

emparray[0] = e;
emparray[1] = m;

for (i = 0; i < emparray.length; i++){
  System.out.println(emparray[i].bonus(5.0))
}
```

Polymorphism

- Every `Employee` in `emparray` "knows" how to calculate its `bonus` correctly!
- Object oriented programming originated in Simula — event simulation loop

```
Q := make-queue(start event)
repeat
  remove first event e from Q
  e.simulate()
  add all events generated
     by e to Q
until Q is empty
```

Polymorphism

- Every `Employee` in `emparray` "knows" how to calculate its `bonus` correctly!
- Object oriented programming originated in Simula — event simulation loop
- Also referred to as runtime polymorphism or inheritance polymorphism

```
Employee[] emparray = new Employee[2];
Employee e = new Employee(...);
Manager m = new Manager(...);

emparray[0] = e;
emparray[1] = m;

for (i = 0; i < emparray.length; i++){
  System.out.println(emparray[i].bonus(5.0))
}
```

Polymorphism

- Every `Employee` in `emparray` "knows" how to calculate its `bonus` correctly!
- Object oriented programming originated in Simula — event simulation loop
- Also referred to as runtime polymorphism or inheritance polymorphism
- Different from structural polymorphism of Haskell etc — called generics in Java

```
Employee[] emparray = new Employee[2];
Employee e = new Employee(...);
Manager m = new Manager(...);

emparray[0] = e;
emparray[1] = m;

for (i = 0; i < emparray.length; i++){
  System.out.println(emparray[i].bonus(5.0))
}
```

Functions, signatures and overloading

- Signature of a function is its name and the list of argument types

But not return type

Functions, signatures and overloading

- Signature of a function is its name and the list of argument types
- Can have different functions with the same name and different signatures
  - For example, multiple constructors

Functions, signatures and overloading

- Signature of a function is its name and the list of argument types
- Can have different functions with the same name and different signatures
  - For example, multiple constructors
- Java class `Arrays` has a method `sort` to sort arbitrary scalar arrays

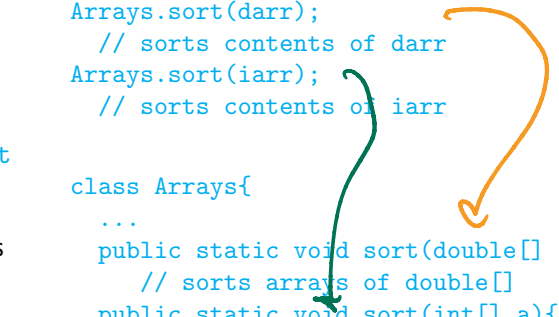
```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr
```

Functions, signatures and overloading

- Signature of a function is its name and the list of argument types
- Can have different functions with the same name and different signatures
  - For example, multiple constructors
- Java class `Arrays` has a method `sort` to sort arbitrary scalar arrays
- Made possible by overloaded methods defined in class `Arrays`

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr


class Arrays{
  ...
  public static void sort(double[] a){..}
     // sorts arrays of double[]
  public static void sort(int[] a){..}
     // sorts arrays of int[]
  ...
}
```



Functions, signatures and overloading

- Overloading: multiple methods, different signatures, choice is static

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr


class Arrays{
  ...
  public static void sort(double[] a){..}
     // sorts arrays of double[]
  public static void sort(int[] a){..}
     // sorts arrays of int[]
  ...
}
```

Functions, signatures and overloading

- Overloading: multiple methods, different signatures, choice is static
- Overriding: multiple methods, same signature, choice is static
  - `Employee.bonus()`
  - `Manager.bonus()`

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr


class Arrays{
  ...
  public static void sort(double[] a){..}
     // sorts arrays of double[]
  public static void sort(int[] a){..}
     // sorts arrays of int[]
  ...
}
```

Functions, signatures and overloading

- Overloading: multiple methods, different signatures, choice is static
- Overriding: multiple methods, same signature, choice is static
  - `Employee.bonus()`
  - `Manager.bonus()`
- Dynamic dispatch: multiple methods, same signature, choice made at run-time

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr


class Arrays{
  ...
  public static void sort(double[] a){..}
     // sorts arrays of double[]
  public static void sort(int[] a){..}
     // sorts arrays of int[]
  ...
}
```

Type casting

- Consider the following assignment

```
Employee e = new Manager(...)
```

Type casting

- Consider the following assignment

  `Employee e = new Manager(...)`

- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this

Type casting

- Consider the following assignment
  `Employee e = new Manager(...)`
- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this
- Type casting — convert `e` to `Manager`
  `((Manager) e).setSecretary(s)`

"promise" to compiler

Type casting

- Consider the following assignment
  `Employee e = new Manager(...)`
- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this
- Type casting — convert `e` to `Manager`
  `((Manager) e).setSecretary(s)`
- Cast fails (error at run time) if `e` is not a `Manager`

Type casting

- Consider the following assignment

  `Employee e = new Manager(...)`
- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this
- Type casting — convert `e` to `Manager`

  `((Manager) e).setSecretary(s)`
- Cast fails (error at run time) if `e` is not a `Manager`
- Can test if `e` is a `Manager`

```
if (e instanceof Manager){
  ((Manager) e).setSecretary(s);
}
```

Type casting

- Consider the following assignment
  `Employee e = new Manager(...)`
- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this
- Type casting — convert `e` to `Manager`
  `((Manager) e).setSecretary(s)`
- Cast fails (error at run time) if `e` is not a `Manager`
- Can test if `e` is a `Manager`

```
if (e instanceof Manager){
  ((Manager) e).setSecretary(s);
}
```

- A simple example of reflection in Java
  - "Think about oneself"

Type casting

- Consider the following assignment

  `Employee e = new Manager(...)`

- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this
- Type casting — convert `e` to `Manager`

  `((Manager) e).setSecretary(s)`
- Cast fails (error at run time) if `e` is not a `Manager`
- Can test if `e` is a `Manager`

```
if (e instanceof Manager){
  ((Manager) e).setSecretary(s);
}
```

- A simple example of reflection in Java
  - "Think about oneself"
- Can also use type casting for basic types

```
double d = 29.98;
long nd = (long) d;
```

From C

Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes

Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes
- Create a class `Shape` so that `Circle`, `Square` and `Rectangle` extend `Shape`
- We want to force every `Shape` to define a function
  `public double perimeter()`

Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes
- Create a class `Shape` so that `Circle`, `Square` and `Rectangle` extend `Shape`
- We want to force every `Shape` to define a function
  `public double perimeter()`
- Could define a function in `Shape` that returns an absurd value
  `public double perimeter() { return(-1.0); }`
- Rely on the subclass to redefine this function

Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes
- Create a class `Shape` so that `Circle`, `Square` and `Rectangle` extend `Shape`
- We want to force every `Shape` to define a function
  `public double perimeter()`
- Could define a function in `Shape` that returns an absurd value
  `public double perimeter() { return(-1.0); }`
- Rely on the subclass to redefine this function
- What if this doesn't happen?
  - Should not depend on programmer discipline