Java: basic datatypes, control flow, classes

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 4, 18 January 2024

Scalar types

- 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

Strings

- String is a built in class

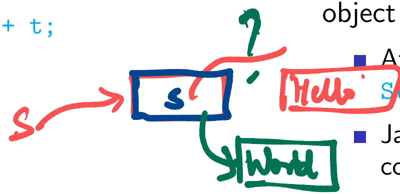
```
String s,t;
```

- 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

  "Memory leak"

  - Cannot write

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

- 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
- Typical declaration

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

— Declare

— Create

  - 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};
```

— Useful, e.g. to create aux array to merge two array!

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 and conditional loops

- `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

do
s
while(c)
⇓
s;
while(c)
s

- `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);
```

Iteration

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

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

- Can define loop variable within loop
  - The scope of `i` is local to the loop

```
public class MyClass {
```

for(i=0, j=0; i+j<k; i++, j++)

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

```
public class MyClass {

  ...

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

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

    return(sum);
  }


}
```

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

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;
}
```

- *Note:* loop variable must be declared in local scope for this version of `for`

```
public class MyClass {

  ...

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

    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;
    }
    }
}
```

no return

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

Date.java

```
~~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;

  ...

}
```

Creating objects

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

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

Python

p = Point(3,5)

Creating objects

- Declare type using class name
- `new` creates a new object
  - How do we set the 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;
  }
}
```

+ Sanity check

Creating objects

- Declare type using class name
- `new` creates a new object
  - How do we set the 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();
  ...
}
```

No "self"

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

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

d.setDate(1, 18, 2024)

Creating objects

- Declare type using class name
- `new` creates a new object
  - How do we set the 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 the 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

- Constructors — special functions called when an object is created
  - Set up an object when we create it
  - Function with the same name as the class
  - `d = new Date(13,8,2024);`

```
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

__init__ in Python

int[] a = new int[100]

Initializing objects

- Constructors — special functions called when an object is created
  - Set up an object when we create it
  - Function with the same name as the class
  - `d = new Date(13,8,2024);`
- Constructors with different signatures
  - `d = new Date(13,8);` sets `year` to `2024`
  - 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 = 2024;
  }
}
```

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,2024);
  }
}
```

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,2024);
  }
}
```

d = new Date();

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

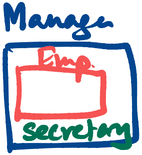
- Manager objects do not automatically have access to private data of parent class.
  - Common to extend a parent class written by someone else
- How can a constructor for Manager set instance variables that are private to Employee?
- Some constructors for Employee
- Use parent class's constructor using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

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!

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(...)
```

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(...)
  ```

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

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(...)
  ```

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()? → checked at compile time
  - 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!
- Recall the event simulation loop that motivated Simula to introduce objects

```
Q := make-queue(first event)
repeat
  remove next event e from Q
  simulate e
  place all events generated
     by e on Q
until Q is empty
```

Polymorphism

- Every Employee in emparray "knows" how to calculate its bonus correctly!
- Recall the event simulation loop that motivated Simula to introduce objects
- 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!
- Recall the event simulation loop that motivated Simula to introduce objects
- 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))
}
```

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)`

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;
  ```