Java: control flow, classes

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 3, 12 January 2023

Built-in datatypes

- Eight primitive scalar types
  - `int`, `long`, `short`, `byte`
  - `float`, `double`
  - `char`
  - `boolean`
- `String` is a built-in class
  - Constants enclosed in double quotes
  - `+` is overloaded for concatenation
  - Strings are immutable

```
String s = "Hello", t = "world";
String u = s + " " + t;
  // "Hello world"
s = s.substring(0,3) + "p!";
  // s is now "Help!"
```

- Arrays are also objects
  - Size of the array can vary
  - Array constants: `{v1, v2, v3}`

```
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
- 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
- `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;

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

    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++;
}
```

- 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;
    int i;

    for (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
```

- Again `for`, different syntax

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

- It appears that 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
- 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?

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?
- 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 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;
  }
}
```

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 2022
  - 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 class Date {
  private int day, month, year;

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

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,2022);
  }
}
```

Copy constructors

- Create a new object from an existing one
- Copy constructor takes an object of the same type as argument
  - Copies the instance variables
  - Use object name to disambiguate which instance variables we are talking about
  - Note that private instance variables of argument are visible
- Shallow copy vs deep copy
  - Want new object to be disjoint from old one
  - If instance variable are objects, we may end up aliasing rather than copying
  - Discuss later — cloning objects

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

  public Date(Date d){
    this.day = d.day;
    this.month = d.month;
    this.year = d.year;
  }
}
```

```
public void UseDate() {
  Date d1,d2;
  d1 = new Date(12,4,1954);
  d2 = new.Date(d1);
}
```