

Lecture 4 — Scopes and Converting Types

CITS2005 Object Oriented Programming

Department of Computer Science and Software Engineering
University of Western Australia

Contents

- See Chapter 2 of the textbook
- More variables
- Scopes
- Type conversion and casting

Variables again!

- We've seen variables: *type* variable-name;
- The *type* describes what sort of thing the variable holds
- We use the variable-name to refer to the variable
- Variables only get one type; cannot be changed at runtime
- All variables must have a type
- Java checks the correctness of all operations based on types (an `int` cannot store a `String`)
- Java has *static* typing (instead of *dynamic* like Python, JavaScript, etc)

Variable Initialisation

```
public class VariableInit {  
    public static void main(String[] args) {  
        double a = 4.5, b, c = 1.0;  
        b = a+c;  
        double d = a*b*c*2.0;  
        System.out.println("d=" + d);  
    }  
}
```

- Variables can be initialised in many ways
- Comma separated, dynamic initialisation, declared and initialised separately

Variable Initialisation

```
public class NoInit {  
    public static void main(String[] args) {  
        int i;  
        System.out.println("i=" + i);  
    }  
}
```

- What happens if we don't initialise a variable?

Scopes

- So far, we have only looked at variables declared inside the `main` method
- They always exist after they are declared
- There are other places (*scopes*) where they can be declared
- Variables defined in a different scope exist for a different span of time

Scopes

- Every block of code inside curly braces `{...}` is a new scope
- The `main` method has a scope
- Each of the following examples have a scope

Scopes

```
public class Scopes {  
    public static void main(String[] args) {  
        if (true) {  
            // This is a scope  
        }  
        while (true) {  
            // This is a scope  
        }  
        for (int i = 0; i < 10; i++) {  
            // This is a scope  
        }  
        {  
            // This is a scope (by itself!)  
        }  
    }  
}
```


Scopes

```
public class Scopes2 {  
    public static void main(String[] args) {  
        int x = 5;  
        if (true) {  
            // Duplicate names are not allowed!  
            // int x = 2;  
            int y = 10;  
            System.out.println ("x=" + x);  
            System.out.println ("y=" + y);  
        }  
        for (int i = 0; i < 1; i++) {  
            int y = 20;  
            System.out.println ("x=" + x);  
            System.out.println ("y=" + y);  
        }  
        System.out.println ("x=" + x);  
        // y cannot be accessed here!  
        // System.out.println ("y=" + y);  
    }  
}
```

Scopes

```
public class NestedScope {  
    public static void main(String[] args) {  
        int x = 5;  
        if (true) {  
            int y = 10;  
            if (true) {  
                int z = 20;  
                // x, y, and z are all in scope here  
            }  
            // x and y are in scope here, but not z  
        }  
        // x is in scope here, but not y or z  
    }  
}
```

-
- Scopes can be nested and capture variables in the enclosing scope(s)

Scopes

```
public class ScopeLoop {  
    public static void main(String[] args) {  
        for (int i = 0; i < 3; i++) {  
            int j = 10;  
            System.out.println("j=" + j);  
            j = 100;  
        }  
    }  
}
```

- Variables get reinitialised every time a scope is seen, even if it's the same scope

Assignment Operator

- We saw lots of operators in the last lecture
- e.g., +, -, &&, /, ||, and more
- Assignment = is also a kind of operator!
- `x = 1+y+5`
- It is a little different to usual
- The left side must be a variable
- The right side must evaluate to a value that can be stored in the variable
- All the other operators we saw produce a result, what is the result of assignment?

Assignment Operator

```
public class Assignment {  
    public static void main(String[] args) {  
        int x;  
        int y = (x = 10);  
        y++;  
        System.out.println("x=" + x);  
        System.out.println("y=" + y);  
    }  
}
```

- Assignment (=) evaluates to the value that is assigned

Compound Assignment

```
public class CompoundAssignment {  
    public static void main(String[] args) {  
        int x = 1;  
        x += 4;  
        x -= 3;  
        x *= 2;  
        x /= 3;  
        System.out.println("x=" + x);  
    }  
}
```

- Java allows a shorthand called *compound assignment*
- Works whenever the following would work `x = x + 1;`
- `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`

Type Conversion

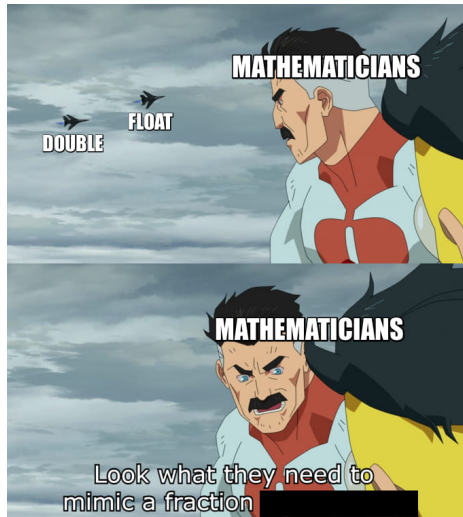
```
public class AutomaticTypeConversion {  
    public static void main(String[] args) {  
        int num = 10;  
        double d = num;  
        System.out.println("num=" + num);  
    }  
}
```

- Sometimes we want to convert an integer value to a double
- Java allows *some* automatic type conversions like this
- When an assignment is made, an *automatic type conversion* happens when a *widening conversion* is allowed

Type Conversion

- `int` can be convert to `float` or `double`
- This is *widening* since `float` and `double` hold integer values and more
- The opposite direction would not be allowed since an `int` cannot always store a `float` value

Mid-lecture break



Type Conversion

```
public class AutomaticTypeConversion {  
    public static void main(String[] args) {  
        double d = 2.3;  
        // Not allowed!  
        // int num = d;  
        int x = 10;  
        long y = x;  
        // Not allowed!  
        // short z = x;  
    }  
}
```

- Recall that short, int, long are 16, 32, and 64 bits respectively
- We cannot convert from double to int
- We cannot convert from int to short
- But we can from int to long because any int fits in a long

Casting

```
public class Casting {  
    public static void main(String[] args) {  
        double d = 66.3;  
        int num = (int) d;  
        char c = (char) num;  
        long l = 11123456789L;  
        int x = (int) l;  
        System.out.println("num=" + num);  
        System.out.println("c=" + c);  
        System.out.println("x=" + x);  
    }  
}
```

- We can force Java to convert between types via a *cast*
- This happens when changing types might mean losing information (e.g., double to int)
- Floating point numbers are truncated

Casting

```
public class Casting {  
    public static void main(String[] args) {  
        double d = 66.3;  
        int num = (int) d;  
        char c = (char) num;  
        long l = 11123456789L;  
        int x = (int) l;  
        System.out.println("num=" + num);  
        System.out.println("c=" + c);  
        System.out.println("x=" + x);  
    }  
}
```

- Integer types chop off higher order bits
- 10100010 00001000 10100011 11110010 → 10100011 11110010
- In this course, we only do this when we know the number will fit

Expressions

- An *expression* in Java has a similar meaning to mathematics
- It is something that must evaluate to some value
- `if (expression)...`
- `while (expression)...`
- `System.out.println(expression)`
- `int x = expression;`
- Sometimes, the types of values in an expression change: `3<4 && 1<2`

Type Promotion

```
public class TypePromotion {  
    public static void main(String[] args) {  
        int x = 1;  
        long y = 2;  
        long z = x + y; // This is OK  
        double d = z + y; // This is OK  
        System.out.println(z);  
        System.out.println(d);  
    }  
}
```

- If Java sees an operator (like +) with different types on either side, it does a *type promotion*
- For example, x is promoted to long before the operator is evaluated
- The resulting value will have the promoted type

Type Promotion

- `char`, `byte`, `short` are promoted to `int`
- `int` is promoted to `long`
- Any integer type is promoted to `float`
- `float` is promoted to `double`

Type Promotion and char

```
public class TypePromotion2 {  
    public static void main(String[] args) {  
        char a = 'a';  
        char b = 'b';  
        // Would only cast a!  
        // char ab = (char) a + b;  
        char ab = (char) (a + b);  
        char c = (char) (b + 1);  
        char z = 'y' + 1; // This is a special case!  
    }  
}
```

- char is (almost) always promoted to int, even if both sides are char!
- Since int cannot be converted to char, we need to *cast* it
- Observe that casting has very high precedence
- There is a special case for constant expressions!

A Note on Parentheses

- Parentheses in Java expressions work the same way as in maths
- They circumvent precedence and group things together
- $(a + b) * (c + d)$