Defensive Programming: Part I. Types, Conditionals, Assertions

Atul Prakash
Reading: Chapter 2-6 Downey. Sun’s Java tutorials as referenced in the slides
You should know from a previous programming course...

- Variables have types: integer, float, boolean, string, ...
- Operators: +, -, =, *, /, %, **, comparison operators, etc.
- if-then-else statements
- loops, e.g., while statements
Some surprises

• We will show that computer programs can behave in unexpected ways:
  • $x + 1 < x$ is possible
  • $(x == x)$ can compute to false
  • $(x != x)$ can compute to true
  • $x < y, x > y, x == y$ can all be false.

• This lecture: overview of Java, along with discussion of nuances of types, conditional statements, and loops
Java Language Fundamentals

• The language syntax is similar to C/ C++
• We will contrast Java with Python where necessary
Keywords

- Keywords are reserved words
- ANSI C has around 32, Java around 50
- Keywords in the Java Language

abstract  continue  for  new  switch
assert ***  default  goto *  package  synchronized
boolean  do  if  private  this
break  double  implements  protected  throw
byte  else  import  public  throws
case  enum ****  instanceof  return  transient
catch  extends  int  short  try
char  final  interface  static  void
class  finally  long  strictfp **  volatile
const *  float  native  super  while

* not used
** added in 1.2
*** added in 1.4
**** added in 5.0
Programs

- Python: you can just type in code. Runs as you type:
  - \(2 + 3\)
  - \(x = "hello"\)
  - `print x`

- Java: programs are compiled. Always start from a "main" function in a class

```java
// HelloWorld.java
public class HelloWorld {
    public static void main(String[] args) {
        String x = "hello";
        System.out.println(x);
    }
}
```
Variables and Types

- All variables must be given a type at start.
- Variable type cannot change (unlike Python)

```java
public class MyFirstApp {
    public static void main(String[] args) {
        String x = "hello";
        int y = 10;
        // prints out hello10
        System.out.println(x + y);
    }
}
```

```java
public class BadMyFirstApp {
    public static void main(String[] args) {
        String x = "hello";
        y = 10;  // illegal. No type given for y.
        x = 10;  // illegal. x is a String
        int x = 10;  // illegal. x is already a String
    }
}
```
Common Types

- short, int, long (integers of various max. lengths)
- float, double (floating point values)
- char (single unicode character) e.g., 'a', '
'
- boolean: true/false
- String: immutable. Use double quotes.

```java
public class App2 {
    public static void main(String[] args) {
        String x = "hello";
        int y1 = 10;
        int y2;
        char z = 'a';
        double w = 2.3;
        y2 = y1/3;  // y2 gets a value 3
        w = y1/3; // Integer division. w becomes 3.0.
        System.out.println("y2: "+ y2);
        System.out.println("w: "+ w);
    }
}
```
## Data Types have limited range

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Type Value</th>
<th>Size</th>
<th>Range</th>
<th>Example literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>true/false</td>
<td>1 byte</td>
<td>-</td>
<td>true, false</td>
</tr>
</tbody>
</table>
| int       | integer    | 32-bit (4-byte), signed, two’s-complement | -2³¹… 2³¹-1 | • decimals: 100, -2  
                        |           |      | -2147483648…   | • Octal: 07, 05  
                        |           |      | -2147483647    | • Hexadecimal: 0x1, 0xA9 |
| long      | integer    | 64-bit (8-byte), signed, two’s-complement | -2⁶³… 2⁶³-1 | • decimals: 10000L, -212L  
                        |           |      | -9,223,372,036,854,775,808… | • Octal: 07123L, 0125L  
                        |           |      | 9,223,372,036,854,775,807 | • Hexadecimal: 0x1D3L, 0xA9L |
| byte      | integer    | 1 byte | -128…127 | - |
| short     | integer    | 16-bit (4-byte), signed, two’s-complement | -2¹⁵… 2¹⁵-1 | - |
                        |           |      | -32,768… | |
                        |           |      | 32,767   | |
# Data Type Ranges

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Type Value</th>
<th>Size</th>
<th>Range</th>
<th>Example literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>floating-point</td>
<td>64-bit (8-byte), described in IEEE reference 754</td>
<td>+/-1.7676931348 x 10 +308... 6231570 x 10 +308...</td>
<td>1e1, 2., .3, 3.14, 56.3e_45d</td>
</tr>
<tr>
<td>float</td>
<td>floating-point</td>
<td>36-bit (4-byte), described in IEEE reference 754</td>
<td>-3.40282347 x 10 +38... -1.40239846 x 10 -45</td>
<td>1e1f, 2.f, .3f, 3.14f, 56.3e_4f</td>
</tr>
</tbody>
</table>
| char      | Single char  | 16-bit (2-byte), signed           | 0...65535                                  | • Single char: ‘T’  
• Escapes: ‘\n’, ‘\r’, ‘\t’  
• Unicode escape: ‘\u0041’ (A) |
Operators

• Operators are symbols that perform an operation on a set of operands (one, two, three)
  – Most operators require two operands - binary operator. For example, +, -, *, /, ** (power), as in:
    • \( z = x + y; \ z = x * y; \ z = x - y; \ z = x**y; \)
  – Some unary operators:
    • ++: increment operator for integers.
    • Two forms: pre-increment and post-increment
      – int i = 10; int j = ++i;  // increment i, then assign.
      – int i = 10; int j = i++;  // assign i, then increment.
  – One ternary operator
    • op1 ? op2 : op3, e.g., (x==y) ? x = 9 : x = 99;
    • It means that if op1 is true, then the result is op2, else op3.
Conditions

• &&: anding; ||: oring; ! used for negation.
• == for equality check. != for non-equality
• >, >, <, <= are additional comparison ops.

```java
public class App4 {
    public static boolean isequilateral(int x, int y, int z) {
        if (x == y && x == z) {
            return true;
        } else {
            return false;
        }
    }
    public static void main(String[] args) {
        boolean ans1 = isequilateral(3, 3, 4);
        boolean ans2 = isequilateral(4, 4, 4);
        System.out.println("ans1: " + ans1);
        System.out.println("ans2: " + ans2);
    }
}
```
Maximum and Minimum integers

• Integers:
  – Integer.MAX_VALUE: largest positive integer
  – Integer.MIN_VALUE: most negative integer

• Similar values for short and long:
  – Short.MAX_VALUE, Long.MIN_VALUE, etc.
Integer Wraparound Problem

• int/short/long values wrap around.
  – Integer.MAX_VALUE + 1 -> wraps around to the Integer.MIN_VALUE.
  – Integer.MIN_VALUE - 1 -> wraps around to the Integer.MAX_VALUE
  – Same principle for short and long

• This has some unexpected implications
  – It is possible that i + 1 < i
  – It is possible that i > 0 and j > 0 but i + j < 0

• Need to be aware of this possibility
Testing Overflows

• Try out the Overflow.java on Ctools

```java
public class Overflow {
    public static void main(String[] unused) {
        // Demonstrates that shorts, ints, and longs wraparound.
        // Floats and Doubles do not.
        test_shorts();
        test_ints();
        test_floats();
        test_longs();
        test_doubles();
    }

    public static void test_shorts() {
        short i = Short.MAX_VALUE;
        System.out.println("i initial value = " + i);
        i += 1;
        System.out.println("i after incrementing = " + i);
        i -= 1;
        System.out.println("i after decrementing = " + i);
    }

    public static void test_ints() {
        int i = Integer.MAX_VALUE;
        System.out.println("i initial value = " + i);
        i += 1;
        System.out.println("i after incrementing = " + i);
        i -= 1;
        System.out.println("i after decrementing = " + i);
    }

    public static void test_floats() {
    }
```
Float/Double

• They have a finite range as well.
  – But, *no wraparound* fortunately.
• Instead, these values overflow to +infinity or -infinity (after rounding).
• Special values:
  • `Float.MAX_VALUE`: largest floating point value
  • `Float.MIN_VALUE`: most negative float
  • `Float.POSITIVE_INFINITY`, `Float.NEGATIVE_INFINITY`
  • `Double.MAX_VALUE`, etc. for double values
Float NaN: Not-A-Number

• For floats and doubles, there is a special value NaN, or Not-a-Number. 0.0/0.0 gives a NaN.
  – Arithmetic operations on NaN give a NaN
  – NaN is not ordered. All comparison operations on NaN, except for !=, give false. Some surprises as a result:
    • NaN == NaN gives false.
    • NaN != NaN gives true
  – Within code, Float.NaN and Double.NaN are the floating point and double NaN values.
NaN

• Some properties:
  – NaN is the only number for which \( x \neq x \). Can serve as a test for NaN.
  – Need to be careful if your computations can give a NaN. Some non-intuitive things are possible:
    • Both \( x > y \) and \( y > x \) can give false if either \( x \) or \( y \) is a NaN.

• Why is NaN there?
  – Numerical experts deemed it necessary to handle erroneous math, such as \( 0.0/0.0 \).
Testing Floats

• Try out TestFloat.java on Ctools

```java
class TestFloat {
    public static void main(String[] args) {
        // An example of overflow:
        double d = 1e308;
        System.out.print("overflow produces infinity: ");
        System.out.println(d + "*10^" + d*10);

        System.out.print("Dividing 1.0 by 0 produces infinity: ");
        System.out.println(1.0f/0);

        try {
            System.out.println("But, integer division by 0 produces an exception: ");
            int i = 1/0;
        } catch (Exception e) {
            System.out.println("Exception caught: " + e);
        }

        // An example of NaN:
        System.out.print("0.0/0.0 is Not-a-Number: ");
        d = 0.0/0.0;
        System.out.println(d);
        boolean eq = (d == d);
        boolean neq = (d != d);
        System.out.println("equality comparison on two NaNs = " + eq);
        System.out.println("non-equality comparison on two NaNs = " + neq);

        // An example of inexact results and rounding:
        System.out.print("values i for which (1.0/i) * i != 1 with float:");
        for (int i = 0; i < 100; i++) {
            float z = 1.0f / i;
            if ((z * i) != 1.0f)
                System.out.print(" " + i);
        }
        System.out.println();
    }
}
```
Type Conversions

• Generally, if you are doing:
  
  • a = b

• Then, a and b must of compatible types.

```java
public class App3 {
    public static void main(String[] args) {
        String x;
        int y = 2;
        float z = 3.5f;
        y = z;  // illegal
        z = y;  // legal.
        y = (int) z;  // legal. Called casting.
        String s = 10;  // illegal.
    }
}
```
Casting

• *Conversion to more general types* generally automatic. E.g.
  
  • double z = 3; // works
  
  • Conversion to a *narrower type* requires a "cast" to tell the compiler that this is intentional.
  
  • int x = 3.4; // fails
  
  • int x = (int) 3.4; works. Value truncated.
  
  • But non-sensical casts fail, as expected

```java
class App3 {
    public static void main(String[] args) {
        int y = 2;
        double z = 3.5;
        y = (int) z; // legal cast.
        System.out.println(y); // prints 3
        String x = (String) y; // illegal cast.
    }
}
```
Statements

• Functions, like main, consist of a sequence of statements

\[ x = 3; \]

• Each statement terminated by a semi-colon

\[ x = 3; \]

is same as
Conditionals

- Syntax: `if (cond) stmt`
- Optional: `else if` and `else` followed by a statement

```java
class ConditionalDemo {
    public static void main(String[] args) {
        int x = 4;
        int y = 5;
        // Syntax:
        // if (cond) stmt
        // [else if (cond) stmt]
        // [else if (cond) stmt]
        // [...]
        // [else stmt]

        if (x > y) System.out.println("x is larger than y");
        else if (x == y) System.out.println("x and y are equal");
        else System.out.println("y is larger than x");
    }
}
```
Compound Statements

• What if we want to do more than one thing in an if statement?
• Use a compound statement to treat multiple statements as one statement:
  • \{ stmt1 ... stmtN \}
Example

class ConditionalDemo2 {
    public static void main(String[] args) {
        int x = 4;
        int y = 5;

        if (x > y) { // Compound statement
            System.out.println("x is larger than y");
            x = 5;
        } else System.out.println("y is larger than x"); // simple statement
    }
}
Be wary of null statement

• A semi-colon by itself is a null statement. It does not do anything.

• The following is legal:
  • if (a > b);  // Note: null statement

• It means do nothing if a is greater than b
This code runs, but has a bug

class ConditionalDemo3 {
    public static void main(String[] args) {
        int x = 4;
        int y = 5;

        if (x > y);
            System.out.println("x is greater than y");
        System.out.println("Done");
    }
}
How Compiler Views the Code

• if \((x > y)\) execute the null statement (;)
• Since no else part, if statement is done.
• Print “x is greater than y”
• Print “Done”
Style Issues

- If conditions are mutually exclusive, use:
  - if, followed by a sequence of else ifs, followed by else.
- Safety: Generally, should include an else, even if it is impossible. Can print an error there if the case is not possible. Only omit it if there would be a null statement.
Example

• Bad style:

```java
if (x < 100) {
    // do something with x
}
// Bad to omit else if x > 100 is an error or assumed to not happen
```

• Uncommon but OK

```java
// Not a common idiom:
if (x < 100) {
    // do something with x ...
} else; // do nothing
```

• Better style:

```java
if (x < 100) {
    // ... do something with x ...
} else System.err.println("Unexpected value of x");
```

• OK, but add comment

```java
// OK to omit else in this case:
if (x < 100) {
    // do something with x ...
} // nothing to do if x >= 100.
```
Switch Statements

- More convenient for a series of equality conditional checks than a sequence of ifs.

```java
int month = 8;
switch (month) {
    case 1: System.out.println("January"); break;
    case 2: System.out.println("February"); break;
    case 3: System.out.println("March"); break;
    case 4: System.out.println("April"); break;
    case 5: System.out.println("May"); break;
    case 6: System.out.println("June"); break;
    case 7: System.out.println("July"); break;
    case 8: System.out.println("August"); break;
    case 9: System.out.println("September"); break;
    case 10: System.out.println("October"); break;
    case 11: System.out.println("November"); break;
    case 12: System.out.println("December"); break;
    default: System.out.println("Invalid month."); break;
}

// Equivalent using if then else:
if (month == 1) System.out.println("January");
else if (month == 2) System.out.println("February");
else if (month == 3) System.out.println("March");
else if (month == 4) System.out.println("April");
else if (month == 5) System.out.println("May");
else if (month == 6) System.out.println("June");
else if (month == 7) System.out.println("July");
else if (month == 8) System.out.println("August");
else if (month == 9) System.out.println("September");
else if (month == 10) System.out.println("October");
else if (month == 11) System.out.println("November");
else if (month == 12) System.out.println("December");
else System.out.println("Invalid month.");
```
Breaks in Switch

- A case continues to next case, unless there is a break. Following will print incorrect output for months 1-9.

```java
switch (month) {
    case 1: System.out.println("January");
    case 2: System.out.println("February");
    case 3: System.out.println("March");
    case 4: System.out.println("April");
    case 5: System.out.println("May");
    case 6: System.out.println("June");
    case 7: System.out.println("July");
    case 8: System.out.println("August");
        System.out.println("Break deleted here");
    case 9: System.out.println("September");
    case 10: System.out.println("October");
        break;
    case 11: System.out.println("November");
        break;
    case 12: System.out.println("December");
        break;
    default:
        System.out.println("Invalid month.");
        break;
}
```
Style - Avoid duplicate code

// bad style
switch (month) {
    case 9: System.out.println("Fall semester"); break;
    case 10: System.out.println("Fall semester"); break;
    case 11: System.out.println("Fall semester"); break;
    case 12: System.out.println("Fall semester"); break;
    case 1: System.out.println("Winter semester"); break;
    case 2: System.out.println("Winter semester"); break;
    case 3: System.out.println("Winter semester"); break;
    case 4: System.out.println("Winter semester"); break;
    default:
        System.out.println("Spring/Summer semester");
} }  

// better style
switch (month) {
    case 9:
    case 10:
    case 11:
    case 12:
        System.out.println("Fall semester"); break;
    case 1:
    case 2:
    case 3:
    case 4:
        System.out.println("Winter semester"); break;
    default:
        System.out.println("Spring/Summer semester");
}
Avoid duplicate code

// bad style
if (a + b > c) System.out.println("it is a triangle");
else if (a + c > b) System.out.println("it is a triangle");
else if (b + c > a) System.out.println("it is a triangle");
else System.out.println("it is not a triangle");

// better style. Eliminate duplicate code in condition
if (a + b > c || a + c > b || b + c > a)
    System.out.println("it is a triangle");
else System.out.println("it is not a triangle");
Defensive programming

- Use either single-line if statement or use compound statement

```java
// Correct, but style can be improved
if (a + b > c || a + c > b || b + c > a)
    System.out.println("it is a triangle");
else
    System.out.println("it is not a triangle");

// Safer style - in case additional statements need to added
// to the if or else part in the future.
if (a + b > c || a + c > b || b + c > a) {
    System.out.println("it is a triangle");
}
else {
    System.out.println("it is not a triangle");
}

// Or Use this. But don't put an additional statement after semi-colon.
if (a + b > c || a + c > b || b + c > a) System.out.println("it is a triangle");
else System.out.println("it is not a triangle");
```
Assert statements

• Assert statements are a way to state assumptions about the code. Code will stop execution if assertion is false

```java
a = -1; b = 4; c = 4;
assert (a > 0 && b > 0 && c > 0);
if (a + b > c || a + c > b || b + c > a) {
    System.out.println("it is a triangle");
} else {
    System.out.println("it is not a triangle");
}
javac SwitchDemo.java
java -ea SwitchDemo

Exception in thread "main" java.lang.AssertionError
    at SwitchDemo.main(SwitchDemo.java:153)
```
Enabling Assertions

• By default, assert statements are ignored by the compiler.

To enable them for debugging, add

"-ea" to the java command (not to javac)

In Eclipse, do Run-> Run Configurations… -> Arguments.

Add -ea to the VM argument.
Asserts to express internal invariants

Initial code

```java
if (i % 3 == 0) {
  ...
} else if (i % 3 == 1) {
  ...
} else { // We know (i % 3 == 2)
  ...
}
```

Better code with assertion

```java
if (i % 3 == 0) {
  ...
} else if (i % 3 == 1) {
  ...
} else { // We know (i % 3 == 2)
  assert (i % 3 == 2);
}
```

Note: % is the mod operator

Example source: [http://java.sun.com/j2se/1.5.0/docs/guide/language/assert.html](http://java.sun.com/j2se/1.5.0/docs/guide/language/assert.html)
Review Sun’s Docs on Asserts

- [http://java.sun.com/j2se/1.5.0/docs/guide/language/assert.html](http://java.sun.com/j2se/1.5.0/docs/guide/language/assert.html)
- Internal invariants
- asserts in else/default
- control flow invariants
Another Example

Initial code

```java
switch(suit) {
    case Suit.CLUBS:
        ...
        break;
    case Suit.DIAMONDS:
        ...
        break;
    case Suit.HEARTS:
        ...
        break;
    case Suit.SPADES:
        ...
}
```

No other suit value assumed to be possible

Better code with default/assert

```java
switch(suit) {
    case Suit.CLUBS:
        ...
        break;
    case Suit.DIAMONDS:
        ...
        break;
    case Suit.HEARTS:
        ...
        break;
    case Suit.SPADES:
        ...
        break;
    default:
        assert false;
}
```
Control-flow invariant

Initial code

```c
void foo() {
    for (...) {
        if (...) {
            return;
        }
        // Execution should never reach this point!!!
    }
    // Execution should never reach this point!!!
}
```

Better code with assertion added in

```c
void foo() {
    for (...) {
        if (...) {
            return;
        }
        assert false; // Execution should never reach this point!
    }
    // Execution should never reach this point!!!
}
```
For and while loops

- For loop:
  ```
  for (int i = 0; i < 10; i++) {
    statement1;
    ...
    statement n;
  }
  ```

- Equivalent while loops:
  ```
  int i = 0;
  while (i < 10) {
    statement1;
    ...
    statement n;
    i++;
  }
  ```

- Python equivalent:
  ```
  for i in range(10):
    statements
  ```

- Continuing, advancing i after each iteration
- Initialize, condition, each iteration

Python equivalent:
```python
for i in range(10):
    statements
```
Following for loops are equivalent - study them

```java
int k;
int sum;
int i;

k = 100;

sum = 0;
for (i = 0; i < k; i++) {
    sum = sum + i;
}
System.out.println("Sum of numbers from 0 to 99 is " + sum);

// equivalent to above
for (i = 0, sum = 0; i < k; i++) {
    sum = sum + i;
}
System.out.println("Sum of numbers from 0 to 99 is " + sum);

// yet another way to write
for (i = 0, sum = 0; i < k; i++) sum = sum + i;
System.out.println("Sum of numbers from 0 to 99 is " + sum);

// yet another way. Note the null statement within the for loop.
for (i = 0, sum = 0; i < k; sum = sum + i, i++);
System.out.println("Sum of numbers from 0 to 99 is " + sum);
```