Java Fundamentals

CSE 413, Autumn 2002
Programming Languages

http://www.cs.washington.edu/education/courses/413/02au/

Readings and References

- Reading
  - » Chapter 3, Fundamental Programming Structures in Java, *Core Java Volume 1*, by Horstmann and Cornell
- Other References
  - » "Language Basics", Java tutorial
  - » http://java.sun.com/docs/books/tutorial/java/nutsandbolts/index.html

Java Primitive Data Types

`boolean`	true or false
`char`	'\u0000' to '\uFFFF' 16 bits (ISO Unicode)
`byte`	-128 to +127
`short`	-32,768 to +32,767
`int`	-2,147,483,648 to +2,147,483,647
`long`	-9,223,372,036,854,775,808 to + 9,223,372,036,854,775,807

Java Primitive Data Types

`float`	-3.40292347E+38 to +3.40292347E+38 (IEEE 754 floating point)
`double`	-1.79769313486231570E+308 to +1.79769313486231570E+308 (IEEE 754 floating point)

Object Wrappers for Primitive Types

Each primitive data type has an object "wrapper" with related functionality

- **`Boolean`**
- **`Byte`**
- **`Character`**
- **`Short`**
- **`Integer`**
- **`Long`**
- **`Float`**
- **`Double`**

Accessing Values In Wrappers

Integer.intValue()

```
Integer i = new Integer( 5 );
int j = i.intValue();
```

j is now primitive int with value 5

There are also useful general purpose functions defined in the wrapper classes

```
static int parseInt(String s, int radix)
static String toString(int i, int radix)
etc
```


Java Operators are Much Like C/C++

- Arithmetic +, -, *, /, %
- Preincrement and postincrement (++, --)
- Assignment (=, +=, -=., etc.)
- Relational comparison operators (==,<,>,<=,>=)
- Boolean logical operators (!, &&, ||)
- Bitwise operators (~,&,|,^)
- Shift operators (>>, <<,>>>)
- No programmer-defined operator overloading (java does overload + for string concatenation)

Integer division and remainder

- Recall this
  - » value = quotient * divisor + remainder
- The division operator is /

```
int x = 7;
int y = x / 2;
```

  - » **`y`** will have the value 3 at this point
- The remainder operator is %

```
int rem = x % 2;
```

  - » **`rem`** will have the value 1 at this point since 7-(3*2) is equal to 1

increment and decrement

- ++ and -- operators allow you to concisely indicate that you want to *use* and *increment or decrement* a variable's value
- pre-increment : ++i
  - » the value of i is incremented before being used in the expression
- post-increment: i++
  - » the value of i is incremented after being used in the expression
- in a statement by itself, makes no difference
  - » there is no expression of interest, just increment the value

```
Blob b = new Blob(count++,color,x,y);
```


Assignment Operators

- Sets a value or expression to a new value
- Simple uses

```
int a = 10;
```

- Compound +=, *= in form of *x op= y,* is short hand for *x = x op y*

```
a += 10;
a = a + 10; // equivalent
```


Relational operators

- Relational operators: boolean result
  - » < less than
  - » > greater than
  - » <= less than or equal
  - » >= greater than or equal
  - » == equivalence

Boolean Logical Operators

- Used to group, join and change boolean results of relationals
- && logical AND
- || logical OR
- ! logical NOT

Bitwise Operators

- Integers types only, produce int or long
- ~ bitwise not (reverses bits)
- & bitwise and
- | bitwise or
- ^ bitwise exclusive or

```
char aChar = 'c'; // 99 = 0x63 = 110 0011
int mask = 0xF;
int z = (aChar & mask);
```


Shift Operators

- Integers types only, produce int or long
- << (left shift): shifts bits to left
- >> (signed right shift): shifts bits to right, keeps the sign (+ value fills with zeros; - value fills with ones)
- >>> (unsigned right shift): shifts bits to right, fills with zeros regardless of sign

Identifiers

- Variable, method, class, or label
- Keywords and reserved words not allowed
- Must begin with a letter, dollar($), or underscore(_)
- Subsequent letters, $, _, or digits
- foobar // valid
- 3_node // invalid

Java Keywords

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

Keywords that are reserved but not used in Java

const goto

Literals - boolean, char, String

- true or false
  - » `boolean isBig = true;`
  - » `boolean isLittle = false;`
- character in an enclosing single quotes
  - » `char c = 'w';`
- Unicode
  - » `char c1 = '\u4567';`
- String
  - » `String s = "hi there";`

Literals - Integer types

- Expressed in decimal, octal, or hexadecimal
  - » 28 = decimal
  - » 034 = octal
  - » 0x1c = hexadecimal
- Default is 32 bits, to get a long specify a suffix of L
  - » 4555L

Literals - floating-point

- floating-point numeric value
- decimal point 16.55
- scientific notation, E or e: 4.33E+44
- 32-bit float, suffix F or f : 1.82F
- 64-bit double, suffix D or d: 12345d
- Default without F or D is 64-bit double

Sequence and Grouping

```
//Simple sequence
statement1;
statement2;
//Grouped -- can replace a single
//statement anywhere
{
   statement1;
   statement2;
}
```

The `if` statement

```
if (condition) {
    this block is executed if the condition is true
} else {
    this block is executed if the condition is false
}
```

- The condition is a logical expression that is evaluated to be **`true`** or **`false`**, depending on the values in the expression and the operators

`switch` statement

```
switch (integral type) {

   case value1 : {
      statement1;
      break; //Break out of switch
   }
   case value2 : {
      statement2;
      break;
   }
   default : {
      statement3;
   }

}
```

there are lots of limitations and potential bugs in using this, so be careful!

The `for` loop

- A counting loop is usually implemented with **`for`**
  - » The **`for`** statement is defined in section 14.13 of the Java Language Specification

initialize

check for termination

update loop control

```
for (i=0; i < count; i++) {
    System.out.println("i : "+i);
}
```

one or more statements in the loop body

`for` example

- a counting loop implemented with **`for`**

can declare variable here or use existing variable

check for termination
i runs from 0 to 19

update loop control
shorthand for `i=i+1;`

```
for (int i=0; i<20; i++) {
    testB.grow();
}
```

Looper.java

limited life of a loop control variable

- The scope of a local variable declared in the ForInit part of a for statement includes all of the following:
  » Its own initializer
  » Any further declarators to the right in the ForInit part of the for statement
  » The Expression and ForUpdate parts of the for statement
  » The contained Statement

from Java Language Specification, section 6.3

The `while` loop

- condition loop is usually implemented with **`while`**
  » The **`while`** statement is defined in section 14.11 of the Java Language Specification

initialize

check for termination

```
periods = 0;
toDate = base;
while (toDate < goal) {
    toDate = toDate+(toDate*rate);
    periods = periods+1;
}
```

one or more statements in the loop body

Note: reaching a limit by counting is satisfying a condition.
`for` loops can be rewritten as **`while`** loops, and vice versa

`while` example

- a condition loop implemented with **`while`**

any variable can be part of the controlling condition

check for termination
indeterminate

update loop control
operation of the loop causes changes that will eventually cause loop to terminate

```
boolean atEndOfFile = false;
while (!atEndOfFile) {
    read another line and set atEndOfFile if appropriate
    process the new line if needed
}
```

Looper.java

body of loop may not execute at all

- Notice that depending on the values of the control variables, it is quite possible that the body of the loop will not execute at all in both **`for`** and **`while`**

check for termination
`toDate` is already greater than **`goal`**, and so the entire loop is skipped

```
goal = 75;
...
periods = 0;
toDate = 100;
while (toDate < goal) {
    toDate += toDate*rate;
    periods++;
}
```

Early terminaton of the loop statement

- A loop is often used to look at all the elements of a list one after another
  - » all the Animals in a PetSet
  - » all the Shapes in a Car
- Sometimes we want to
  - » exit the loop statement early if we find some particular element or condition while we are looping
  - » ie, get out of the loop statement (for, while) entirely

break - jump to loop exit

```
public void snack() {
  for (int i=0; i<theBunch.size(); i++) {
    if (remainingFood <= 0) {
      System.out.println("No food left, so no more snacks.");
      break;
    }
    Animal pet = (Animal)theBunch.get(i);
    double s = Math.min(remainingFood,pet.getMealSize());
    pet.eat(s);
    remainingFood -= s;
  }
  // the break statement takes us here, out of the loop entirely
}
```


Early cycling of the loop

- Sometimes we want to
  - » Stop processing the item we are looking at right now and go on to the next one
- The loop statement (for, while) is still the controlling structure, but we just want to go to the next iteration of the loop

continue - jump to loop end

```
public void dine() {
  for (int i=0; i<theBunch.size(); i++) {
    Animal pet = (Animal)theBunch.get(i);
    double s = 2*pet.getMealSize();
    if (remainingFood < s) {
      System.out.println("Not enough food for "+pet+
        "'s dinner, so we'll skip to next animal.");
      continue;
    }
    pet.eat(s);
    remainingFood -= s;
    // continue takes us here, the end of this loop
  }
}
```

Short-Circuit Operators

- With && and ||, only as much of the logical expression as needed is evaluated
- Example:

```
int i=1;
if (false && (++i == 2))
   System.out.println(i); // doesn't print
if (true || (++i == 2))
   System.out.println(i); // prints 1
```

- Don't use increment operator in places where it might not get executed (as in this example)

boolean expressions and variables

- If you find yourself doing something like this

```
if (pageNumber == lastPage) {
  allDone = true;
} else {
  allDone = false;
}
```

- there is an easier way

```
allDone = (pageNumber == lastPage);
```

boolean variable

boolean expression

conditional operator (3 operands)

- If you find yourself doing something like this

```
if (score < 0) {
  color = Color.red;
} else {
  color = Color.black;
}
```

- there is an easier way

use this value if expression is true

```
color = (score < 0) ? Color.red : Color.black;
```

variable

boolean expression

use this value if expression is false

Appendix

Positional Notation

- Each column in a number represents an additional power of the base number
- in base ten
  - » $1=1*10^0$, $30=3*10^1$, $200=2*10^2$
- in base sixteen
  - » $1=1*16^0$, $30=3*16^1$, $200=2*16^2$
  - » we use A,B,C,D,E,F to represent the numbers between 9_{16} and 10_{16}

Binary, Hex, and Decimal

$2^8=256_{10}$	$2^7=128_{10}$	$2^6=64_{10}$	$2^5=32_{10}$	$2^4=16_{10}$	$2^3=8_{10}$	$2^2=4_{10}$	$2^1=2_{10}$	$2^0=1_{10}$	Hex_{16}	$Decimal_{10}$
							1	1	3	3
					1	0	0	1	9	9
					1	0	1	0	A	10
					1	1	1	1	F	15
				1	0	0	0	0	10	16
				1	1	1	1	1	1F	31
		1	1	1	1	1	1	1	7F	127
	1	1	1	1	1	1	1	1	FF	255

Binary, Hex, and Decimal

$Binary_2$	$16^4=65536_{10}$	$16^3=4096_{10}$	$16^2=256_{10}$	$16^1=16_{10}$	$16^0=1_{10}$	$Decimal_{10}$
11					3	3
1001					9	9
1010					A	10
1111					F	15
1 0000				1	0	16
1 1111				1	F	31
111 1111				7	F	127
1111 1111				F	F	255

Binary, Hex, and Decimal

$Binary_2$	Hex_{16}	$10^3=1000_{10}$	$10^2=100_{10}$	$10^1=10_{10}$	$10^0=1_{10}$
11	3				3
1001	9				9
1010	A			1	0
1111	F			1	5
1 0000	10			1	6
1 1111	1F			3	1
111 1111	7F		1	2	7
1111 1111	FF		2	5	5