Fundamental Constructs

Chapter 3

Copyright © 1998-2003 Delroy A. Brinkerhoff. All Rights Reserved.

Chapter 3 Slide 1 of 24

Intrinsic Types

Simple, built-in data types

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
<th>range values</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>0</td>
<td>true, false</td>
<td>return type</td>
</tr>
<tr>
<td>boolean</td>
<td>1?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>1</td>
<td>-128 to 127</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>2</td>
<td>0 to 65535</td>
<td>unicode</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>-32768 to 32767</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>-2147483648 to 2147483647</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>8</td>
<td>~ -9.223×10^18 to ~9.223×10^18</td>
<td>15 sig digs</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>~ -3.40×10^38 to ~3.40×10^38</td>
<td>6 sig digs</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Size and byte-order are Java-defined
- int is the smallest data type supporting arithmetic
- Arithmetic on char is not allowed
- Signed -2bits-1 to 2bits-1 - 1; Unsigned 0 to 2bits - 1

Chapter 3 Slide 2 of 24

Type Promotions and Casts

Type promotions (see illustration, p. 48)

- Rules of automatic type promotions
  - Automatic when going from "narrow" to "wide" data types
  - Wider means a greater range, not number of bytes
  - The wide type must encompass the full range of the narrow type
  - Expression type is type of widest operand
  - byte ≤ short ≤ int ≤ long (8 bytes) ≤ float (4 bytes) ≤ double
  - int is the smallest data type supporting arithmetic
  - Ordinal constants are int and floating point constants are double
  - Explicit cast required to convert from "wide" to "narrow" type
    - <i>int i = (int) 3.14;</i>
    - <i>double d = (double) 1 / 3;</i>
- Boolean values are not castable (period! see C++ note p. 49)
- Can't cast to or from String (must use methods; see p. 56)

Chapter 3 Slide 3 of 24

Java™ “Library” Features

Symbolic or named constants and methods (functions)

- Everything is in a class
- Usually in lang or util
- "static" makes one copy
- "final" prevents change
- Symbolic constants
  - <C/C++> #define enum
  - Java public static final
  - Names are upper case

```java
class Math {
    private Math() { }
    public static final double PI = 3.14159;
    public static final double sin(double x) {
        ...
    }
}
x = 2 * Math.PI;
y = Math.sin(x/2);
```

Chapter 3 Slide 4 of 24

Wrapper Classes

Object versions of intrinsic data types in the java.lang package

- Java™ predefines classes for some of the simple data types
  - java.lang.Integer
  - java.lang.Float
  - java.lang.Char
  - java.lang.Long
  - java.lang.Double
- Methods
  - int i = Integer.parseInt("123");
  - double x = Double.parseDouble("3.14159");
  - boolean equals(Object o)

```java
String x = Integer.toString(i);
String x = i + " ";
```

Chapter 3 Slide 5 of 24

String Class

java.lang.String

- Instantiation
  - <String greeting = "Hello world";> // preferred
  - <String greeting = new String("Hello world");> // not preferred
- Methods
  - <String compareTo("hello world");> // like strcmp
  - <String equals("hello world");> // boolean
  - <String equalsIgnoreCase("hello world");> // boolean
  - <String length();> // 11
  - <String substring(1,5);> // "ello"
  - <String indexOf(o");> // 4
  - <String indexOf("world");> // 6
  - <String concat( from Java™);> // String concatenation
  - <String + from Java™;> // String concatenation and assignment

Chapter 3 Slide 6 of 24
StringBuffer Class

- Analogous to C++'s `strstream` classes
- Constructors
  - `StringBuffer SB = new StringBuffer();`
  - `StringBuffer SB = new StringBuffer(11);`
  - `StringBuffer SB = new StringBuffer("Hello world");`
- Dynamically sized (but changing size may be expensive)
- Methods (some)
  - `SB.append(" from Java");` // may also be numeric or char
  - `SB.capacity();`
  - `SB.length();`
  - `SB.insert(11, "");`
  - `SB.insert(11, "********");`

```
Chapter 3 Slide 7 of 24
```

StringTokenizer Class

- `java.util.StringTokenizer` vs `strtok`
- `StringTokenizer(String s [,String delim [,boolean rd ]])`
  - Parameters
    - `s` string to tokenize (not changed)
    - `delim` string of delimiter characters (separates tokens, white space by default)
    - `rd` return delimiter (nextToken returns delimiter character)
- Methods
  - `hasMoreTokens();`
  - `nextToken(String delim);`

```
Chapter 3 Slide 8 of 24
```

Formatting Numbers

- `java.text.NumberFormat`
  - `NumberFormat f;`
  - `f = NumberFormat.getInstance();`
  - `f = NumberFormat.getCurrencyInstance();`
  - `f = NumberFormat.getNumberInstance();`
  - `f = NumberFormat.getPercentInstance();`
- Methods
  - `String f.format(number);`
  - `setMinimumIntegerDigits / setMaximumIntegerDigits`
  - `setMinimumFractionDigits / setMaximumFractionDigits`
  - `Locale[] locales = NumberFormat.getAvailableLocales();`
- `java.util.Locale`
  - Constants: `Locale.CANADA`, `Locale.JAPAN`, `Locale.US`, etc.
  - `static void setDefault(Locale newLocale)`
  - `static Locale getDefault();`

```
Chapter 3 Slide 9 of 24
```

Objects

- Created (instantiated) from classes
  - Objects are instances of classes
    - All objects are instantiated dynamically on the heap with `new` (3 exceptions: String constants, statically initialized arrays, and object factories-- static methods that create and return objects)
    - Attributes of each object are separate and distinct
    - Objects are named with reference variables allocated on the stack
    - Anonymous objects are not named (i.e., not assigned to a reference)
  - `instanceof` operator left hand operand is an object, right hand operand is a class; boolean valued
  - Examples
    - `Circle C;` // define reference var
    - `C = new Circle(10,0,0);` // instantiation only
    - `Circle C = new Circle(10,0,0);` // both in 1 statement
    - `new Circle(10,0,0);` // anonymous object

```
Chapter 3 Slide 10 of 24
```

Java™ Collects Garbage

- No explicit object deallocation (i.e., no delete operator)
  - Java™ provides automatic garbage collection (there is no delete operator)
    - Java™ uses a mark-and-sweep algorithm
      - Implemented as a low-priority thread
      - Suspends execution of user program
      - Mark all allocated heap memory as unused
      - Follow pointers and mark reached memory as used
      - Reclaim unmarked memory
  - C++ programmers know that memory management is too important to leave up to the system; Smalltalk [Java] programmers know that memory management is too important to leave up to programmers”
    - Bjarne Strutsrudp
  - Note that garbage collection cannot be forced (System.gc() and Runtime.gc() not withstanding)

```
Chapter 3 Slide 11 of 24
```

Arrays

- Always are instantiated objects (see p. 96)
  - Variable definition is distinct from object instantiation
    - `int monLength[];` // variable definition
    - `int monLength = new int[12];` // object instantiation
    - `int monLength[] = new int[12];` // both
  - Static instantiation and initialization
    - `int monLength[] = { 31,28,31,30,31,30,31,31,30,31,30,31 };`
    - `Indexed 0 .. size - 1`
    - `Index is bounds checked (errors throw an exception)`
    - `monLength.capacity();` // capacity, not filled slots
  - Array declaration syntax
    - `int[] monLength;` or `int monLength[];`
    - `The first is useful for method return types: int[] func(void)`

```
Chapter 3 Slide 12 of 24
```
**Arrays of Objects**

Arrays of objects are really arrays of pointers to objects.

```java
Employee emp[];
emp = new Employee[5];
for (int i = 0; i < emp.length; i++)
emp[i] = new Employee();
```

**AND and OR Operators**

Logical and bitwise

- Logical operators (left and right hand operands are boolean)
  - `&&`: short-circuit if `E1` is false, `E2` not evaluated
  - `&`: non-short-circuit `E1 & E2` always evaluated
- Logical operators (left and right hand operands are boolean)
  - `||`: short-circuit if `E1` is true, `E2` not evaluated
  - `|`: non-short-circuit `E1 | E2` always evaluated

- Bitwise operators (left and right hand operands are numeric)
  - `a & b`
  - `a | b`
  - `a ^ b`
  - `~a`

**Bit-Shifting Operations**

- Bit-shifting
  - `<<` right hand operand is mod the number of bits of left hand operand
  - `>>` each bit shifted is equivalent to `× 2`
  - `>>>` always pads on the left with 0; each bit shifted is equivalent to `÷ 2`

**Simple Program Structure**

Method order within class is not fixed

```java
public class Demo
{
    private double value;
    public Demo(String input)
    {
        value = Double.parseDouble(input);
    }
    public double sqr()
    {
        return value * value;
    }
    public static void main(String[] args)
    {
        Demo d = new Demo(args[0]);
        double y = d.sqr();
    }
}
```

**Sending Messages**

Objects communicate/cooperate/collaborate by exchanging messages

- Every message has a sending object and a receiving object
- A message is a high-level concept
  - Equivalent to an operation or behavior (i.e., what an object can do)
  - Implemented by one or more methods or functions

**Method Definition Syntax**

- Always defined inside a class but cannot be nested
- Header: public class `Demo`
  - `private double value;`
  - `public Demo(String input)`
  - `public demo = new Demo(args[0]);`
  - `double y = demo.sqr();`
- `class Square`
**Method Call Syntax**

Summary / review

- Does not include modifiers or typing information
- Non-static methods are called through an object
- Static methods are called through the class

Employee worker = new Employee();
Manager boss = worker.getBoss(416, new Date());
int number = Employee.getPhoneNumber();

public class Bar {
    public Bar() {
        someMethod(); // calling object is new Bar object being built
    }
}

**Objects and Methods**

Summary / review

- Non-static methods are always called through an object
  - The object is a default target for the method
  - The object is bound to the method through the "this" reference

public class Foo {
    private int number;
    public Foo(int n) {
        number = n;
    }
    public int bar(Foo f) {
        return number + f.number;
    }
}

Foo f1 = new Foo(10);
Foo f2 = new Foo(20);
int count = f1.bar(f2);

**Object Assignment and Reference**

Creating object aliases

- Object names are references that "point" to an object
- Multiple names (aliases) may reference the same object
  - Created by assignment
  - Created by method calls

Circle c = new Circle(100,20,10);
Circle t = c; // doesn't duplicate
void draw(Circle s) { }
if (c == null) . . .

**Pass By Value: Simple Types**

Illustrated with int

int x = 100;
void inc(int a) {
    a += 100;
}

**Pass By Value: Objects**

Assume class Employee

- Java™ only has pass by value
- The value passed is an address

Employee x = new Employee("Fred");
update(x);
void update(Emp a) {
    a = new Employee("Sid");
}

**Pass By Value: Updating Objects**

Class Employee has an instance variable name

Employee x = new Employee("Fred");
update(x);
void update(Emp a) {
    a.name = "Sid";
}

Chapter 3