

Primitive/Reference Types and Value Semantics

Computer Science and Engineering ■ College of Engineering ■ The Ohio State University

Lecture 2

Primitive Types

- Java contains 8 primitive types
 - boolean, byte, short, int, long, float, double, char
- Variable declaration
 - `<type> <identifier> { = <expression> } ;`
 - `short index;`
 - `boolean isDone = true;`
 - `int counter = 3;`
 - `float tip = cost * 0.15;`
- Language defines size and range of each type (ie number of bytes)
 - Also defines “default initial values”, but these default values are *not* used for local variables!

Size and Range of Primitive Types

<i>Type</i>	<i>Size (bytes)</i>	<i>Range</i>
boolean	1 bit	true or false
byte	1	-128 to 127
short	2	-32768 to 32767
int	4	-2147483648 to 2147483647
long	8	-9223372036854775808 to 9223372036854775807
float	4	about $\pm 10^{\pm 38}$, 7 significant digits
double	8	about $\pm 10^{\pm 308}$, 15 significant digits
char	2	Unicode UTF-16 code unit

Literals (ie Constants)

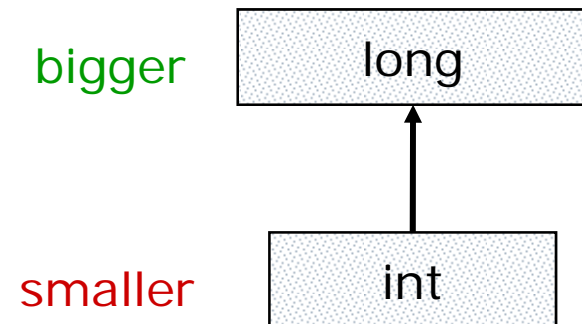
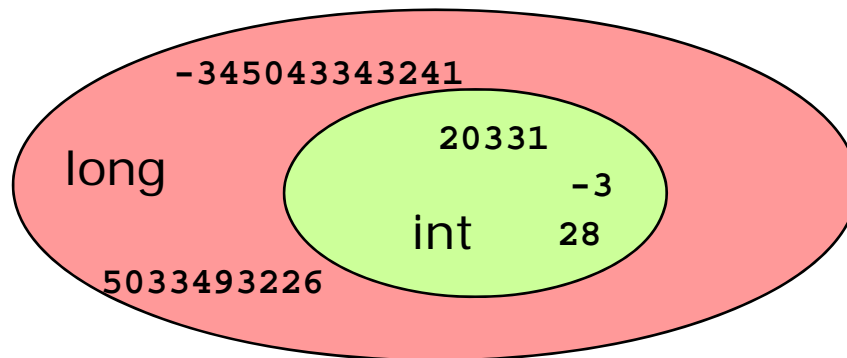
- Boolean
 - `true`, `false`
- Character
 - With single quotes, eg `'Q'`
 - `\n`, `\t`, `\\`, `\'`, `\"`, `\uXXXX` (for unicode)
- Integer
 - `29`, `035`, `0x1D` (ie decimal, octal, hexadecimal)
 - Sizes: `29` vs `29L` (default int vs long)
- Floating-point
 - `18.`, `18.0`, `1.8e1`, `.18E+2`, `180.0e-1`
 - Sizes: `18.0` vs `18.0F` (default double vs float)
- String
 - With double quotes, `"like this"`

Good Practice: Upper Case L for Long

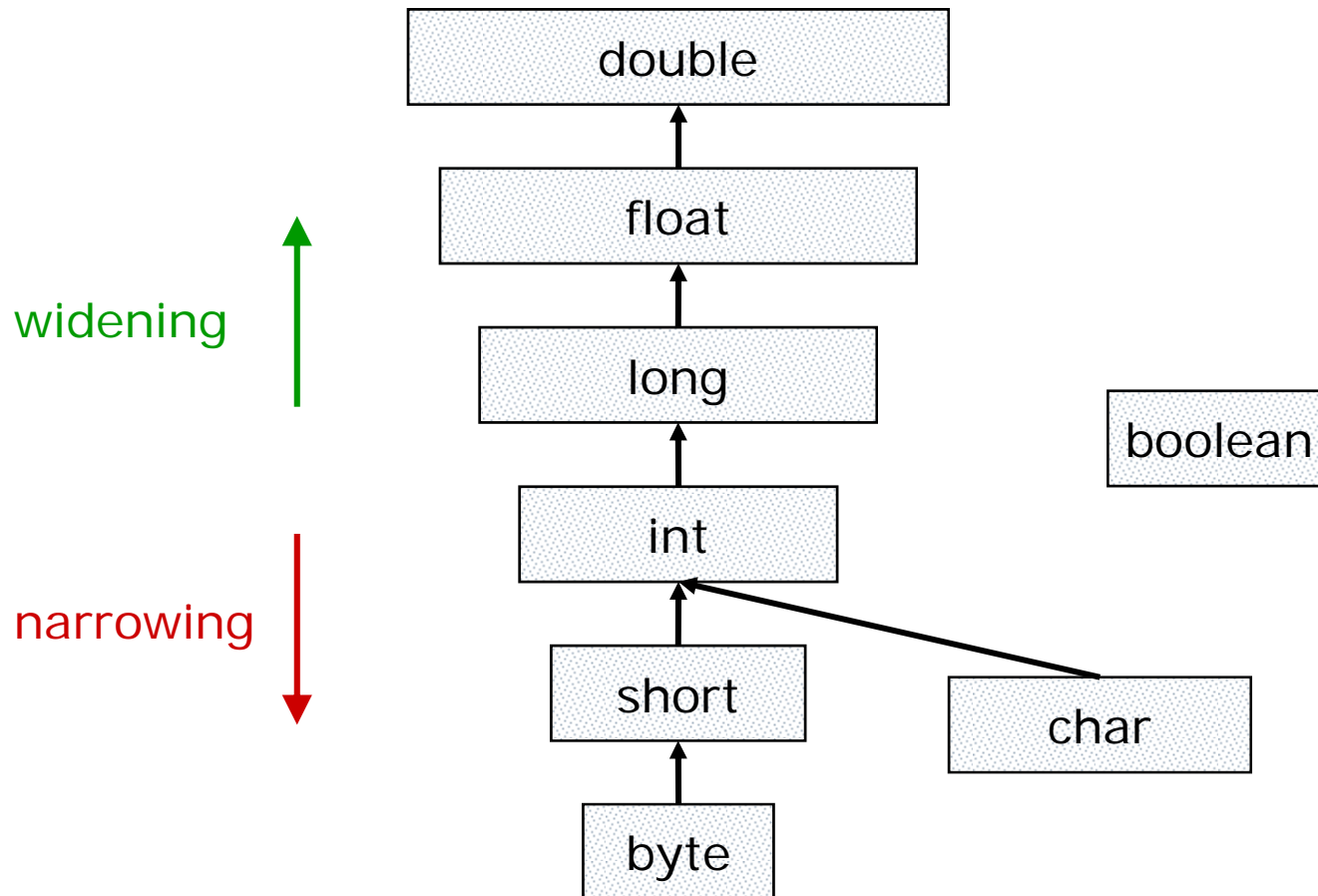
- When writing a long constant, use an *upper case 'L'*
`long x = 13L;`
- Lower case 'l' is syntactically correct, but potentially confusing
`long y = 13l; //y is 13. surprise!`
- For consistency, prefer 'F' to 'f'
 - Common usage, however, is lower case 'f'
`float t = 1.0f; //no confusion`
 - Less important since lower case version does not create confusion

Hierarchy of Primitive Types

- A type is a *set* of possible values
- Some types are “bigger” (ie have more possible values) than others
 - Every int is a long, so long is a “bigger” type
 - Subset inclusion



Hierarchy of Primitive Types



Casting and Widening

- Widening is automatic when needed (ie implicit)

```
int i = 13;           //no type conversion
long x = 12;         //int to long (widening)
long y = i;          //int to long (widening)
```

- Widening can be forced by an **explicit cast**

```
int sum = 76;
int count = 10;
float average = sum/count;
    //no type conversion, result is 7
average = sum/(float)count;
    //int to float (widening), result is 7.6
```

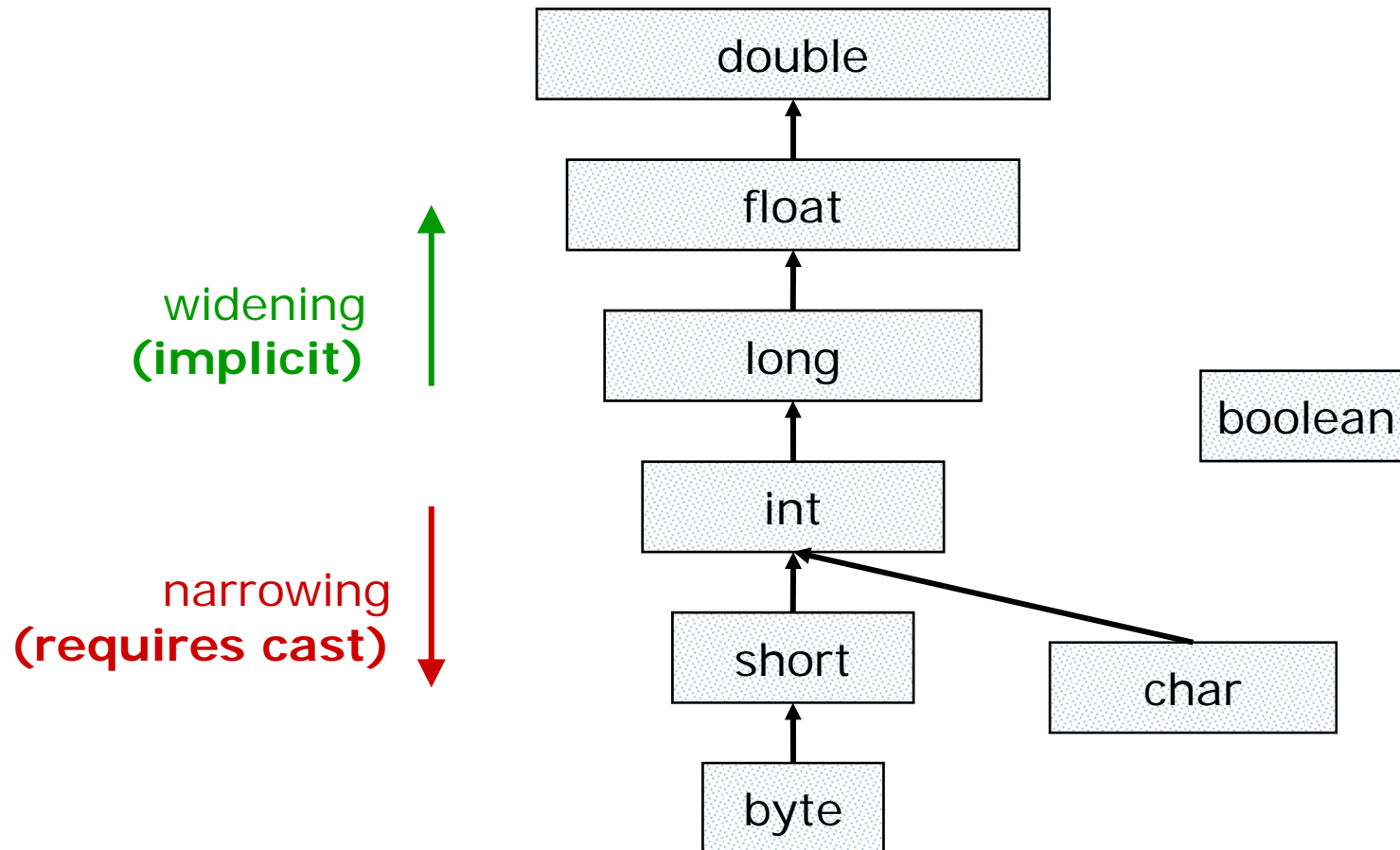

Casting and Narrowing

- Narrowing requires explicit cast

```
int i = 12L;           //error: requires cast
int i = (int) 12L;    //long to int (narrowing)
byte j = (byte) i;    //int to byte (narrowing)
```

- Cast is a promise by program that the narrowing type conversion is ok
- May result in loss of information
 - Casting float to int truncates decimals
 - Casting long to int discards top bytes
- Warning: *Widening* can lose information too!
 - How?

Hierarchy of Primitive Types



Value Semantics

- A variable is the name of a memory location that holds a value

tip

- Declaration *binds* the variable name to a memory location

```
short counter;
```

counter

- Assignment *copies* contents of memory

```
counter = start;
```

counter

start

counter

start

Value Semantics: Assignment

- Assignment is a *copy*
- Example: What is the final value of balanceA? balanceB?

```
int balanceA = 300;
```

balanceA 300

```
int balanceB = balanceA;
```

balanceB 300 balanceA 300

```
balanceB = balanceB + 150;
```

balanceB 450 balanceA 300

Value Semantics: Parameters

- Parameters are *copied*
- Example: What is the final value of `balanceA`?

```
void increaseByOneFifty(int cash) {  
    cash = cash + 150;  
}  
...  
int balanceA = 300;  
increaseByOneFifty(balanceA);
```

cash 300

cash 450

balanceA 300

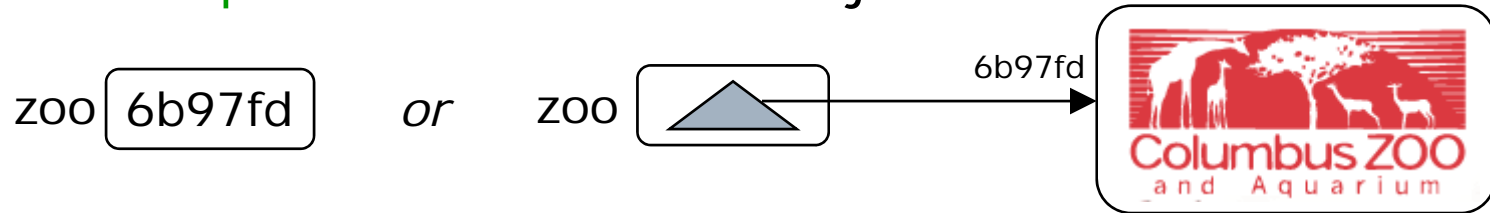
balanceA 300

Reference Types

- Class types, provided by:
 - Java standard libraries
 - String, Integer, Date, System, ...
 - Programmer
 - Person, Animal, Savings, HelloWorldApp
- Arrays
 - Can contain primitive or reference types
 - int[], float[], String[], ...
 - Indexed starting from 0
- Just one literal for references: **null**

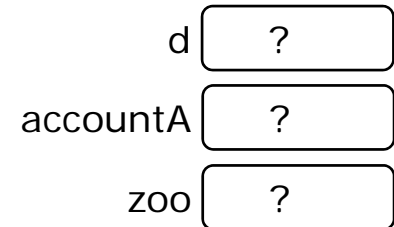
Value Semantics (of References!)

- Recall: A variable is the name of a memory location that holds “a value”
 - For reference types, the “value” in the memory location is a **pointer** to the actual object!



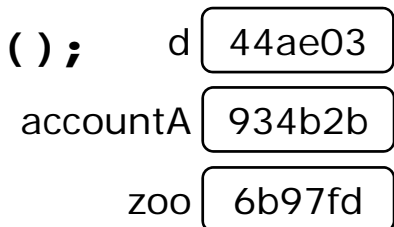
- Declaration binds the variable to a memory location (which contains a **pointer**)

```
java.util.Date d;  
Savings accountA;  
Animal[] zoo;
```



- Explicit object creation with **new()**

```
java.util.Date d = new java.util.Date();  
Savings accountA = new Savings(300);  
Animal[] zoo = new Animal[50];
```



Using Arrays

- An array type does not include the length

```
int[] ids = new int[rosterSize];  
int searchRoster(int[] students) { ... }
```

- Array length

- Set at run time, can not change after initialization

```
int[] ids = new int[rosterSize];
```

- Available as a property with *.length*

```
void examine (int[] ids) {  
    for (int i = 0; i < ids.length; i++) {...}
```

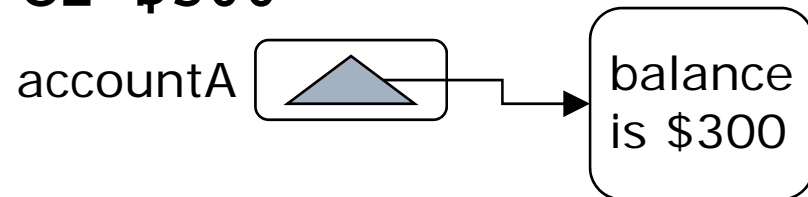
- Iteration: “foreach” loop (keyword is still *for*)

```
int sum = 0;  
for (int a : ids)  
    sum += a;  
float average = sum/(float)ids.length
```


Assignment Creates an Alias

- Assignment *copies the pointer*
- Example: What is the final balance of (the object pointed-to by) accountA? accountB?

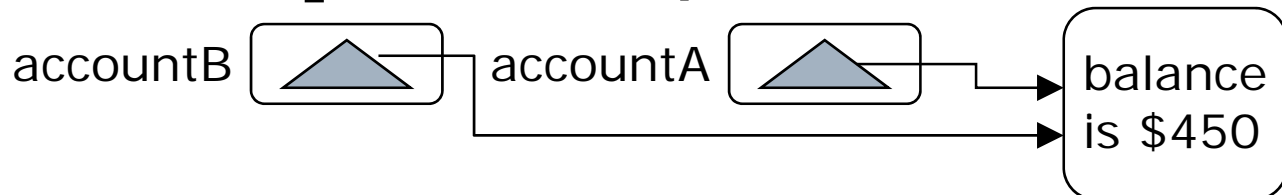
```
//(the object pointed-to by) accountA  
has a balance of $300
```



```
Savings accountB = accountA;
```



```
accountB.deposit(150);
```



Parameter Passing Creates an Alias

- ❑ Parameter passing *copies the pointer*
- ❑ Example: What is the final balance of (the object pointed-to by) accountA?

```
void increaseByOneFifty(Savings cash) {
```

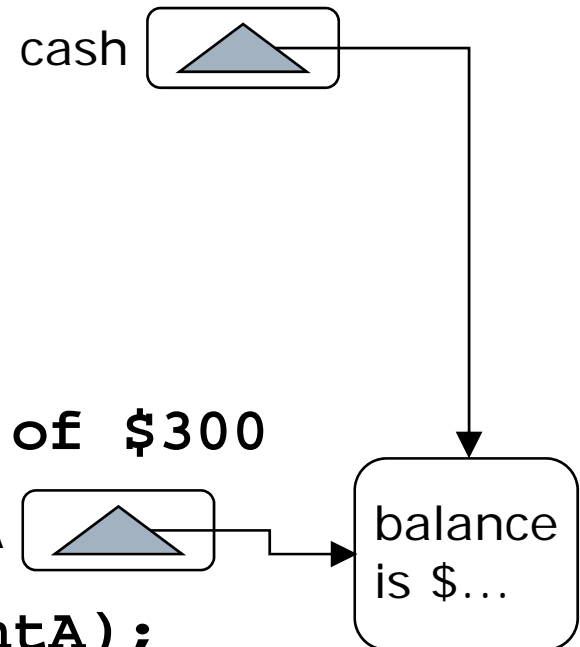
```
    cash.deposit(150);
```

```
}
```

```
...
```

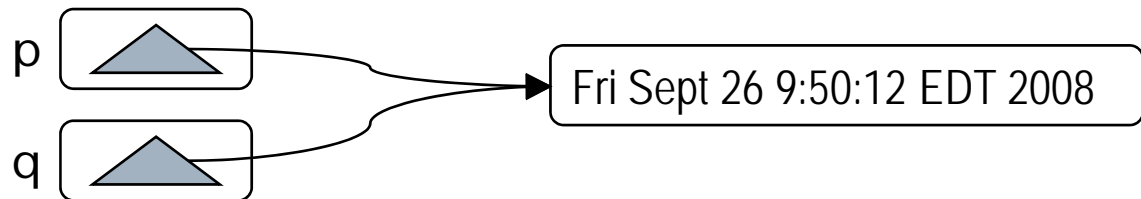
```
//accountA has a balance of $300
```

```
increaseByOneFifty(accountA);
```

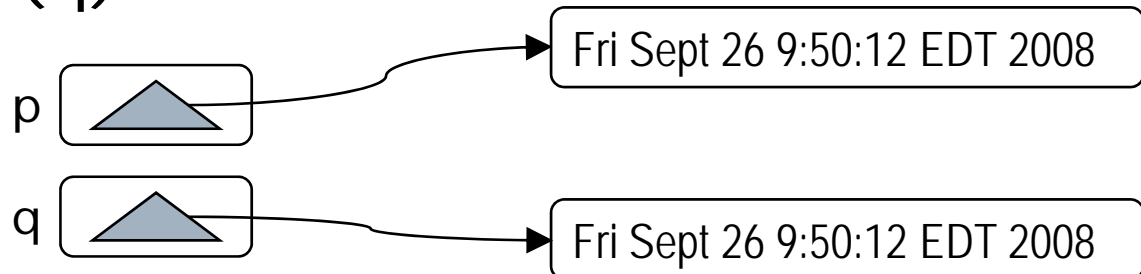


Testing for Equality

- For references p , q consider: $p == q$
 - Compares *pointers* for equality
 - Do they refer to the same object?



- How do we test if *objects* are equal?
 - Define a boolean method `equals()`
 - `p.equals(q)`



Supplemental Reading

- IBM developerWorks paper
 - “Pass-by-value semantics in Java applications”
 - <http://www.ibm.com/developerworks/java/library/j-passbyval/>

Summary

- Primitive Types and operators
- Type conversions with casting
 - Widening is implicit
 - Narrowing requires an explicit cast
- Value Semantics
 - Assignment operator performs a *copy*
 - Parameters are “pass by value” (ie *copied*)
- Reference Types
 - Reference and referent (ie object)
 - Variable is the reference, not the referent
 - Assignment copies reference, creates alias
 - Parameter passing copies reference, creates alias