

# Primitive/Reference Types and Value Semantics

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

Type	Size (bytes)	Range
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 $\pm 10^{+38}$ , 7 significant digits
double	8	about $\pm 10^{+308}$ , 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

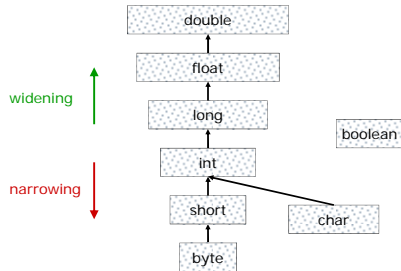
- When writing a long constant, use an upper case 'L'
  - `long x = 13L;`
- Lower case 'l' is syntactically correct, but potentially confusing
  - `long y = 13l; //y is 13. surprise!`
- For consistency, prefer 'F' to 'f'
  - Common usage, however, is lower case 'f'
    - `float t = 1.0f; //no confusion`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

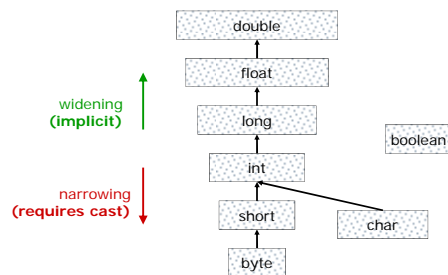
```
int sum = 76;
int count = 10;
float average = sum/count;
//no type conversion, result is 7
average = sum/(float)count;
//int to float (widening), result is 7.6
```

## 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 [ 8.65 ]
```
- Declaration *binds* the variable name to a memory location
 

```
short counter;
counter [ ? ]
```
- Assignment *copies* contents of memory
 

```
counter [ ? ] start [ 14 ]
counter = start;
counter [ 14 ] start [ 14 ]
```

## Value Semantics: Assignment

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

```
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);
    
```

## 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!

zoo (6b97fd) or zoo 6b97fd

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

```

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

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

```

java.util.Date d = new java.util.Date();   d: 44ae03
Savings accountA = new Savings(300);     accountA: 934b2b
Animal[] zoo = new Animal[50];          zoo: 6b97fd
    
```

## 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 balances in accountA? accountB?

```

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

Savings accountB = accountA;
accountB  accountA  balance is $300

accountB.deposit(150);
accountB  accountA  balance is $450
    
```

## Parameter Passing Creates an Alias

- Parameter passing *copies the pointer*
- Example: What is the final balance of accountA?

```

void increaseByOneFifty(Savings cash) {
    cash.deposit(150);
}
...
//accountA has a balance of $300
increaseByOneFifty(accountA);
accountA  balance is $450
    
```

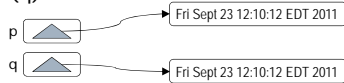
## Testing for Equality

Computer Science and Engineering @ The Ohio State University

- 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

Computer Science and Engineering @ The Ohio State University

- Some "references" explaining parameter passing in Java
  - <http://javadude.com/articles/passbyvalue.htm>
  - <http://www.yoda.arachsys.com/java/passing.html>
  - <http://www.javaworld.com/javaworld/jvqa/2000-05/03-qa-0526-pass.html>

## Summary

Computer Science and Engineering @ The Ohio State University

- 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