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

<table>
<thead>
<tr>
<th>Type</th>
<th>Size (Bytes)</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1 bit</td>
<td>true or false</td>
</tr>
<tr>
<td>byte</td>
<td>1</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>-32768 to 32767</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>-2147483648 to 2147483647</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
<td>-9223372036854775808 to 9223372036854775807</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>about ±10(^{38}), 7 significant digits</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>about ±10(^{308}), 15 significant digits</td>
</tr>
<tr>
<td>char</td>
<td>2</td>
<td>Unicode UTF-16 code unit</td>
</tr>
</tbody>
</table>

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

### Literals (ie Constants)
- **Boolean**
  - `true, false`
- **Character**
  - With single quotes, eg ‘Q’
  - \n, \t, \v, \", \\, \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”

### 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

- double
- float
- long
- int
- byte
- short
- char
- boolean

Widening:
- int i = 13; //no type conversion
- long x = 12; //int to long (widening)
- long y = 1; //int to long (widening)

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

Widening is automatic when needed (i.e. implicit)
- int i = 13;
- long x = 12;
- long y = 1;

Widening can be forced by an explicit cast
- int sum = 76;
- int count = 10;
- float average = sum / count;

Casting and Widening

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 the 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
- Declaration binds the variable name to a memory location
- Assignment copies the contents of memory

Value Semantics: Assignment

- Assignment is a copy
- Example: What is the final value of balanceA? balanceB?
  - int balanceA = 300;
  - int balanceB = balanceA;
  - balanceB = balanceB + 150;
  - balanceB = balanceB + 150;

- counter = start;
- counter = start;
- balanceB = balanceB + 150;
- balanceB = balanceB + 150;

- counter = start;
- counter = start;
- balanceB = balanceB + 150;
- balanceB = balanceB + 150;
Value Semantics: Parameters

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

```java
void increaseByOneFifty(int cash) {
    cash = cash + 150;
    //balanceA 300
    int balanceA = 300;
    //balanceA 300
    increaseByOneFifty(balanceA);
    //balanceA 450
}
```

```java
int balanceA = 300;
increaseByOneFifty(balanceA);
```

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: 609776 or Zoo 6b97fd

d null
java.util.Date d;

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

```java
Savings accountA;
Animal[] zoo;
```

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

```java
Savings accountA = new Savings(300);
Animal[] zoo = new Animal[50];
```

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

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

Using Arrays

- **An array type does not include the length**

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

- **Array length**
  - Set at run time, can not change after initialization
  - Available as a property with .length

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

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

Parameter Passing Creates an Alias

- **Parameter passing copies the pointer**

```java
void increaseByOneFifty(Savings cash) {
    cash.deposit(150);
}
```

```java
increaseByOneFifty(accountA);
```

Assignment Creates an Alias

- **Assignment copies the pointer**
  - **Example:** What is the final balances in `accountA`? `accountB`?

```java
//accountA has a balance of $300
Savings accountB = accountA;
accountB.deposit(150);
//accountA has a balance of $300
accountB balance is $300
```
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”

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