

Data Types in Java

CPSC 2150

Primitive Variables

- 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 (i.e., number of bytes)
 - Also defines “default initial values”, but these default values are *not* used for local variables!
- You can cast from one type to another using the data type you wish to cast to in parenthesis
 - `a = b / (double) c;`

Primitive Variables

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 $\pm 10^{\pm 38}$, 7 significant digits
double	8	about $\pm 10^{\pm 308}$, 15 significant digits
char	2	Unicode UTF-16 code unit

Constant Values

- Boolean
 - `true`, `false`
- Character
 - With single quotes, e.g., `'Q'`
 - `\n`, `\t`, `\\`, `\'`, `\"`, `\uXXXX` (for Unicode)
- Integer
 - `29`, `035`, `0x1D` (i.e., 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"`
 - **Note: not a primitive data type**

Good Practice: Use Upper Case 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'

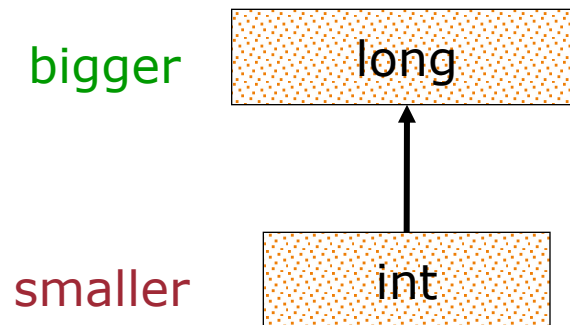
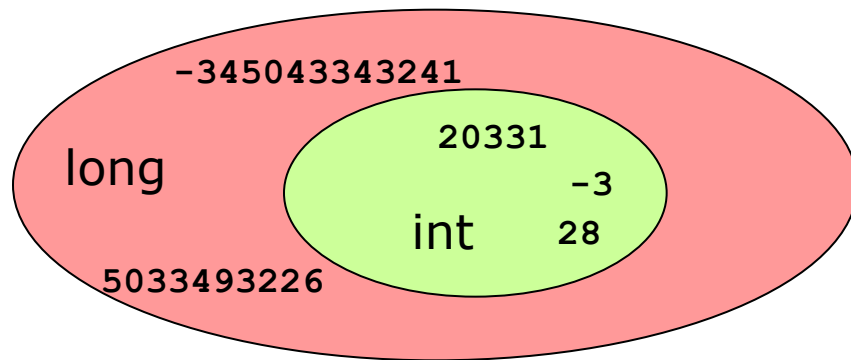
- Common usage, however, is lower case 'f'

```
float t = 1.0f; // no confusion
```

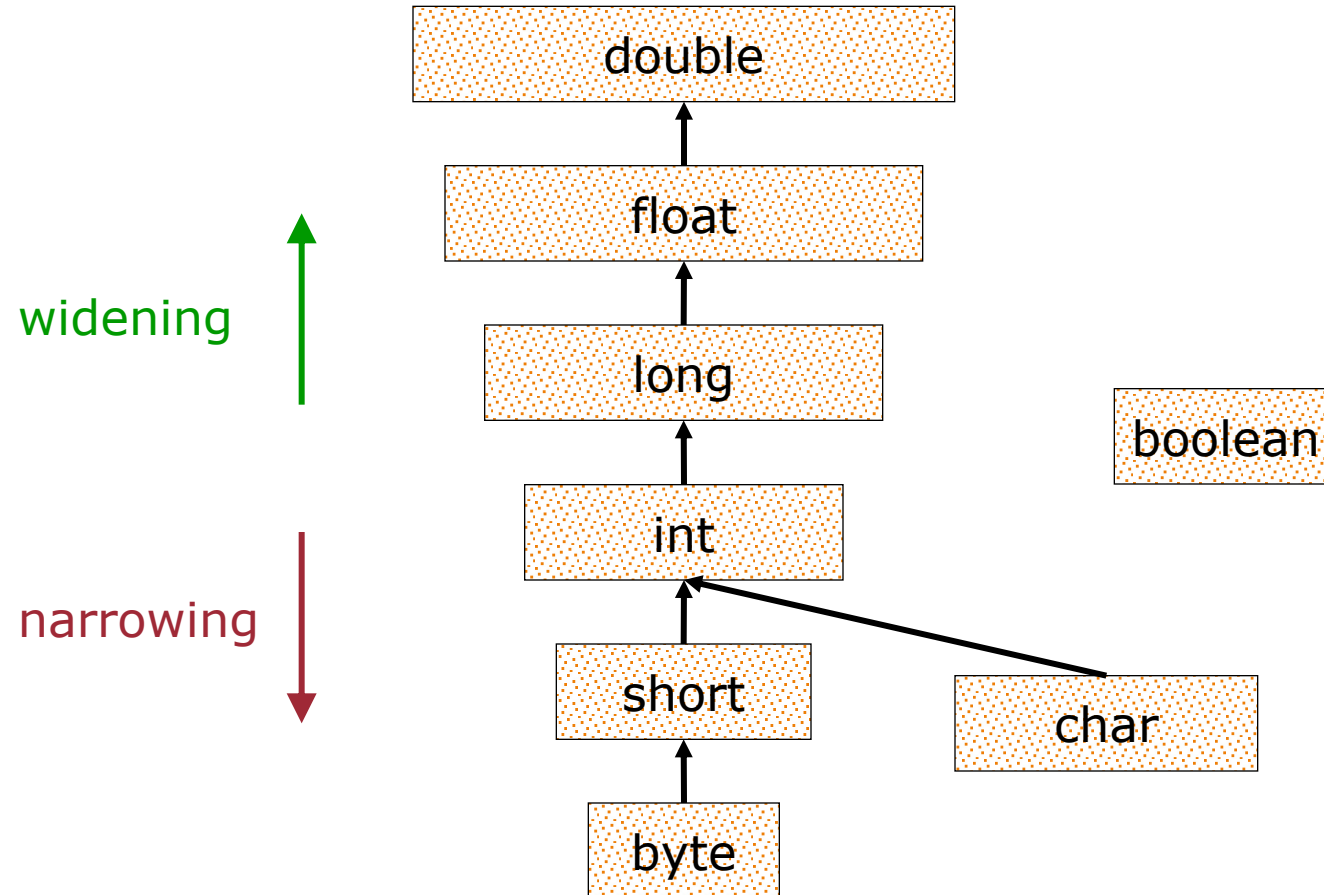
- Less important since lower case version does not create confusion

Primitive Data Types

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



Hierarchy of Primitive Data Types



Casting and Widening

- Widening is automatic when needed (i.e., 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
 - You as the programmer are saying that it 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!
 - A `float` uses some of its 32 bits for the exponent, while a `long` does not. A `float` can contain the magnitude of any `long`, but may lose some precision

References and Pointers

- In C and C++, we often used pointers
 - These are a reference data type
 - The value of a pointer is the memory location where the data is stored
- Java has reference data types, but not in the same way that C and C++ do
 - Anything that is not one of the 8 primitive data types will be a reference data type
 - This means that classes and arrays are reference types
 - You do not have to specify that it is a pointer in Java, like you do in C++
 - In Java, it is automatically a reference data type
 - Java does not have the dereferencing operator or the `->` operator
 - That is handled automatically
- Referencing and dereferencing is handled automatically in Java, but you still need to be aware of when you are using reference data types because of aliasing errors that can occur

Aliasing

- Aliasing can occur when we use reference types.
- When we use the assignment operator (`=`) with reference types, we are copying the memory location, so both reference variables will be pointing to the same memory location
- The equivalence operator (`==`) will compare to see if the memory locations are the same
 - Use `.equals()` instead (**Note:** More on this in a future in-class lecture)
 - **Ex:** `object1.equals(object2)` will return `true` if both objects are equal (based on the definition of the `equals` method)
- Consider a class called `SavingsAccount`. It has a balance and methods to deposit and withdraw money.
 - It's `equals()` method will return *true* if the two accounts have the same balance

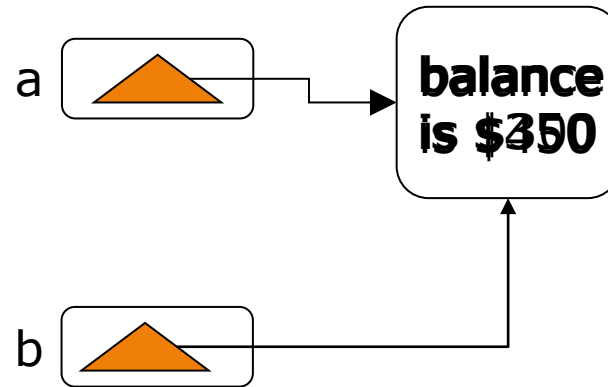
➡ `SavingsAccount a = new SavingsAccount(300);`

➡ `SavingsAccount b;`

➡ `b = a;`

➡ `b.deposit(150);`

➡ `a.withdraw(100);`



What will the value of x be at the end?

```
int x = 0;
SavingsAccount a = new SavingsAccount(300);
SavingsAccount b = new SavingsAccount(300);

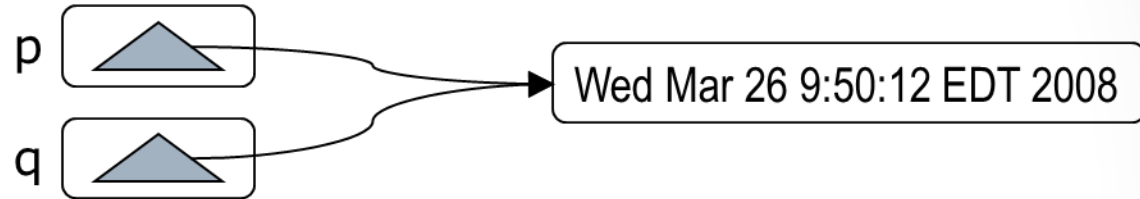
if (a == b) {
    x = 7;
}
```

x will still have a value of 0!

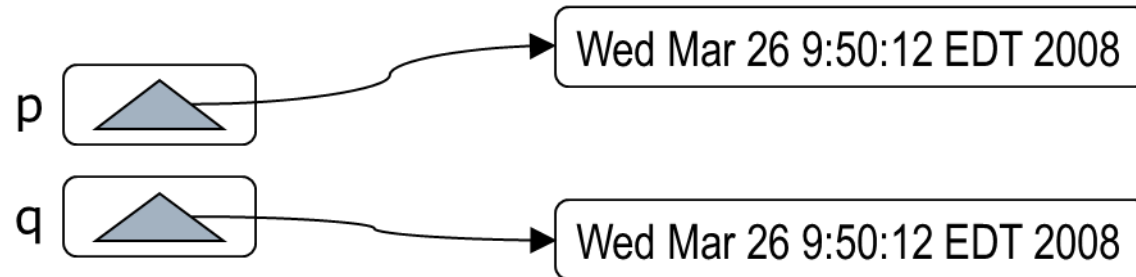
a == b is *false*, because they are looking at two different memory locations!

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()` (**Note:** More on this in a future in-class lecture)
 - `p.equals(q)`



What will the value of x be at the end?

```
int x = 0;
SavingsAccount a = new SavingsAccount(300);
SavingsAccount b = new SavingsAccount(300);

if (a.equals(b)) {
    x = 7;
}
```

x will have a value of 7!

a.equals(b) will return *true*, since they have the same balance

Wrapper Data Types

- Each of our primitive data types has a corresponding **wrapper** class
 - Noted with an upper-case letter
 - Examples `int -> Integer`, `long -> Long`, `double -> Double`
- Provide helpful functionality through static methods

```
int a = Integer.parseInt("35778");
String str = Double.toString(123.76);
```
- Provide useful static constants

```
Integer.MAX_VALUE
Integer.MIN_VALUE
```

 - Useful to avoid overflow errors
- Our wrapper classes are reference data types!
- Our wrapper classes are immutable

Immutability

- An immutable object is one whose (abstract) value can never change
 - Constructor allows initialization to different values
 - Changes the pointer to a new memory location, not the value in the old memory location
 - No **mutator** methods
- Why would we want such a thing?
- Because aliasing an immutable is safe!
- Why do we want immutable wrappers for primitive types instead of just using primitive types?
 - Because sometimes reference types need to be passed as parameters (e.g., to Java collections)
 - Add a variable `x` to a `List`. Later the value of `x` changes. Immutability allows `x` to change while the value in the `List` does not change

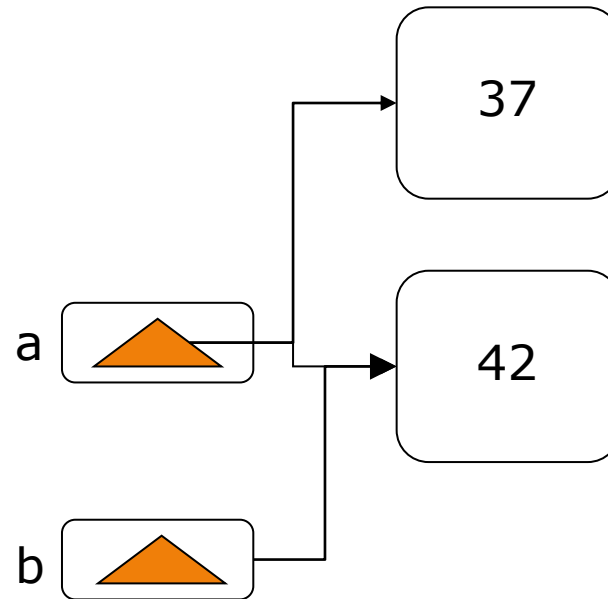
Immutability

➡ Integer a = 42;

➡ Integer b;

➡ b = a;

➡ a = 37;



Boxing and Unboxing

- Our wrapper types are just classes that contain our primitive types with some helpful functionality
- Immutability is added to avoid some aliasing issues since we have converted a primitive type into a reference type
- We refer to taking a primitive type and storing it in its associated wrapper type as ***boxing***.
 - Pass the primitive value into the constructor to initialize the object
- We refer to extracting the base primitive type from its wrapper type as ***unboxing***.
 - Call a getter function to retrieve the primitive value
- For these wrapper types, Java actually handles most of it for us automatically
 - The equality operator will still give issues though
 - Checks the pointer for equality
 - Use the `equals()` method instead

Boxing and Unboxing

- **Boxing:** primitive --> wrapper

```
Integer integerObject = new Integer(42);
```

- **Unboxing:** wrapper --> primitive

- `int i = integerObject.intValue();`

- Java does this *automatically* for you

```
Double price = 499.99; // auto-box
```

```
price = price + 19.90; // auto-unbox then box
```

- But be very careful...

```
Integer i = new Integer(2);
```

```
Integer j = new Integer(2);
```

```
assert (i >= j); // success (unboxing)
```

```
assert (i <= j); // success (unboxing)
```

```
assert (i == j); // failure! (no unboxing)
```

String variables

- `String` variables are not a primitive type in Java
 - They are a user defined class
 - This is also the case in C++, strings are a class that stores an ordered array of characters
- This means that `Strings` in Java are a reference type!
- Boxing and unboxing still happens for us, but be careful with equality

```
String str1 = "hello";  
String str2 = "hello";  
if(str1 == str2) {  
    // returns false in some versions of Java  
    // It's checking the pointers  
    // use str1.equals(str2) for the comparison
```

final variables

- In C++, we have constants.
- In Java, they are referred to as `final` variables
- A variable defined as `final` can be initialized once, and then cannot be changed
 - Initialization does not need to be on the same line as the declaration
- **Note:** that for a reference variable, `final` just means that the reference itself (i.e., the memory location) cannot change. The value stored at that location can still change.

```
// Example 1
final int x = 7;
x = 5; // X
```

```
// Example 2
final int x;
x = 5; // OK
```

```
// Example 3
final SavingsAccount a = new SavingsAccount(300);
final SavingsAccount b = new SavingsAccount(450);
a.deposit(50); // OK, the reference does not change
b = a; // X
```

Arrays

- In C++, when you declared an array you declared it with the size of the array so the array could be allocated in memory

```
int arr[10]; // declares an array of length 10
```

- In Java, arrays are references. They are just a pointer to an array

- Note that the square brackets are now before the variable name

```
int[] arr; // arr is a pointer that will point to an array of integers  
arr = new int[10]; // now a points to an array of 10 integers
```

- In Java, a default initial value is assigned to each element of the array when it is initialized, depending on the data type:
 - `boolean` defaults to *false*
 - `char` defaults to the character with Unicode 0000
 - Numeric primitives default to 0
 - All reference types default to `null`

Arrays

- Accessing an index in an array is the same in Java and C++

```
arr[0] = 5; // set the first element of the array to 5
```

- In C++ and Java, arrays can be initialized with values in curly braces

```
int[] arr = {1, 2, 3};
```

- Java arrays have the length attribute which can be called at any time

```
int[] arr = new int[10];
```

```
int j = arr.length; // j is set to 10
```

Multidimensional Arrays

- Like in C++, Java arrays can be multi-dimensional
- Two-dimensional arrays do not need to be rectangular
 - Each row can be a different size

```
int[][] arr;           // arr is a pointer to a two-dimensional array
arr = new int[5][];    // arr now has 5 rows, but no columns yet
arr[0] = new int[1];   // arr's first row has 1 column
arr[1] = new int[2];   // arr's second row has 2 columns
arr[2] = new int[3];   // arr's third row has 3 columns
arr[3] = new int[5];   // arr's fourth row has 5 columns
arr[4] = new int[5];   // arr's fifth row also has 5 columns
```

The End

- Continue on to the discussion on Logic in Java