TYPES, VARIABLES, AND REFERENCES

Prof. Chris Jermaine
cmj4@cs.rice.edu

Java Is A Strongly-Typed Language

- What does "strongly typed" mean?

- "I know it when I see it."

    — US Supreme Court Justice Potter Stewart, Jacobellis v. Ohio (1964)

- Everyone agrees Java is strongly-typed

    — My definition: only a small set of very well-defined type conversions are built in:

    ```
    int a = 34;
    long b = 454;
    b += a;
    ```
    — Everything else is disallowed:

    ```
    int a = 34;
    if (a) {
        // do some stuff
    }
    ```

2

Java Is A Statically-Typed Language

• What does "statically typed" (as opposed to "dyn. typed") mean?

• Means most/all type checking is done at compile time

    — Dynamically typed means type checking done as variables are used

    — Python, for example, is dynamically typed

    — As an aside, Java is not **purely** statically typed...

    — But you should try to avoid use of dynamic typing! (more later in the class)

Java does not do "type inference"

- In type inference...

    — Programmer just uses vars and the compiler figures out the type

- Lack of type inference

    — Means all vars must be declared with type info before use

So What Does This Mean For You?

- Declare a variable before you use it

- State the type

- Or the compiler will yell at you

What Types Are There in Java?

- This is a bit complicated...

- A variable itself can only be of one of 8 + 1 types

- The eight "primitive types" are byte, short, int, long, float, double, boolean, char

- A variable can also be a **reference** (like an address... more later!)

  — Var declared to be a type other than one of the 8 primitives is **actually** a reference

  — So "String temp" gives you a reference to an **object** of type String

- Each one of these 8 + 1 types has a specific memory footprint

- When you declare a variable...

  — Java allocates enough memory to hold data of the type associated with the var

  — Puts the default value into the var

Objects

- References always point to **objects**

- Objects can only be created via the "new" command

  ```
  String temp = new String ("foo");
  ```

- Just like variables, objects also have "types"

  — We say the object is an "instance" of a class

  — Ex: the object referenced by temp is an instance of the String class

- Just like variables, objects also have memory footprints

  — Depends upon the member variables and methods

- When object has no more references, can be **garbage collected**

  ```
  String temp = new String ("foo");
  temp = null; // now JVM can garbage collect the object
  ```

References

- When declared, the value is "null"

```
// this code will output "foo"
String temp;
if (temp == null)
    System.out.println ("foo");
```

- Any attempt to use a null reference will crash (or not compile)

```
// this code will crash
String temp;
System.out.println (temp);
```

- This is why we are always calling "new"... creates a new **object**

```
// this code will output "foo"
String temp = new String("foo");
System.out.println (temp);
```

References

- Can assign references to one another

    — If both "point" to objects of the same type

    ```
    // this code will output "foo foo"
    String temp1 = new String ("foo");
    String temp2 = temp1;
    System.out.format ("%s %s\n", temp1, temp2);
    ```

    — This is called "aliasing" and is quite dangerous... why?

    — In my perfect language, reference assignments would not be allowed

    — Another way to avoid problems is to never change the state of an object

    — This is called **functional programming**

    — More on this later in the semester...

Method Calls in Java

- All methods calls are "by value"

- So if I say:

```
String temp = new String ("foo");
someObject.someMethod (temp);
```

- And we have:

```
public void someMethod (String input) {
    // some code
```

- This is more-or-less the same as:

```
String temp = new String ("foo");
someObject.someMethod (temp);
String input = temp;
// some code
```

Call-By-Value

- So what happens if I have:

```
String temp;
someObject.stringFactory (temp);
System.out.println (temp);


...
public void stringFactory (String input) {
    input = new String ("foo");
}
```

Call-By-Value

- So what happens if I have:

```
String temp;
someObject.stringFactory (temp);
System.out.println (temp);

...
public void stringFactory (String input) {
    input = new String ("foo");
}
```

- Program will crash! Why?

   — "input" was really just a local var, with value of "temp" copied into it

   — Value of "temp" was null when it was copied over

   — You can't change "temp" by putting a non-null value into "input"

Call-By-Value

- So what happens if I have:

```
String temp = new String ("foo");
someObject.stringModifier (temp);
System.out.println (temp);

...
public void stringModifier (String input) {
    input = new String ("bar");
}
```

Call-By-Value

- So what happens if I have:

```
String temp = new String ("foo");
someObject.stringModifier (temp);
System.out.println (temp);


...
public void stringModifier (String input) {
    input = new String ("bar");
}
```

- This'll print out "foo". Why?

    — Again, by changing **what** "input" points to, I can't affect "temp"

- In fact, it's impossible to modify temp under call-by-value

    — Though it might be possible to modify object pointed to by temp via method call

14

Casting

- In Java, can (try to) change types using "casting"

```
int x = 1234;
long y = 5678;
x = y; // compiler won't like this... loss of precision

x = (int) y; // compiler will be fine with this
```

- With primitive types, must cast when assign may be dangerous

  — Loss of precision

- But some casts just not allowed

  — Can never cast boolean, for example

- Casting for references is much more complicated (bad idea!)

  — Will cover when we cover classes in detail

Primitive Types and Refs Revisited

- It's confusing having both references and primitive types

    — Sort of strange that these two lines are so fundamentally different:

    ```
    String temp1;
    int temp2;
    ```

    — First declares a reference (not a String!), second declares an int

- Plus there's stuff you can't even do with primitive types

    — Can't put 'em in generic containers (containers this week)

    — This is the reason for the following Java strangeness:

    ```
    Integer temp1; // declares a ref to an Integer object
    int temp2; // declares an int
    ```

    — What's going on here?

Primitive Types and Refs Revisited

- Why not just kill primitive types?

    — Then every variable is of type "reference"

    — Java already has object wrappers for all of the primitive types!

    — Would definitely make things much more beautiful

Primitive Types and Refs Revisited

- Why not just kill primitive types?

  — Then every variable is of type "reference"

  — Java already has object wrappers for all of the primitive types!

  — Would definitely make things much more beautiful

- Obvious reason is performance

  — Getting to data requires chasing a reference

  — Sorting (then traversing) an array of doubles vs. Doubles

- Plus there is the issue of memory usage

  — byte: 8 bits; Byte: 8 + 64 bits

- So we get something that's somewhat confusing...

  — ...but perhaps necessary