Some Interpreted Languages are Compiled

Java Source | Java Bytecode | App

Why do this?
Speed is part of the answer, but only part

Issues with Preconditions/Invariants

- We use assert statements
  - Raise error if violated
  - Stops the program
- **Does not find all errors**
  - Only when used wrong
  - What if error in seldom used portion of code?
- **Performance overhead**
  - Remember Assignment 6
  - But when needed most!

```
class Fraction(object):
    _numerator = 0    # int, hidden
    _denominator = 1  # int > 0, hidden
    ...
    @property
    def numerator(self):
        """Numerator value of Fraction
        Invariant: must be an int"""
        return self._numerator

    @numerator.setter
    def numerator(self,value):
        assert type(value) == int
        self._numerator = value
```

Types of Programming Errors

Syntactic Errors

- Compiler can find for you
  - Never need to "run" code
  - Finds error without inputs
- Limited to bad "grammar"
  - Fast, but not thorough
- **Examples:**
  - Unknown function/variable
  - Function call w/ wrong type

Runtime errors

- Can only check at run time
  - Found by the interpreter
  - Or OS if using machine code
- Can catch all errors
  - But just saw the problems
- **Examples:**
  - Variable outside of range
  - Access field of null variable

Java is a Statically Typed Language

- A **variable** is
  - a **named** memory location (**box**),
  - a **value** (in the box), and
  - a **type** (limiting what can be put in box)

Different from Python

x 5 **int**

Here is variable **x**, with value 5.
It can contain an **int** value.

Variable names must start with a letter

area 20.1 **double**

Here is variable **area**, with value 20.1.
It can contain a **double** value.

Variable Declarations

- A *declaration of a variable* gives the **name** of the variable and the **type** of value it can contain

int x;
Here is a declaration of x, indicating that it contain an **int** value.

double area;
Here is a declaration of area, indicating that it can contain a **double** value.

Assignment Statements

- *Execution of an assignment statement* stores a value in a variable

To execute the assignment
<var>= <expr>;
evaluate expression <expr> and store its value in variable <var>

x= x + 1;
Evaluate expression x+1 and store its value in variable x.

Initialization: Declaration+Assignment

- Can combine declaration and assignment

int x = 3;
Here is a declaration of x, indicating that it contain an **int** value.
It starts with a value of 3.

double area = 2.3;
Here is a declaration of area, indicating that it can contain a **double** value.
It starts with a value of 2.3.

- In Python these are one and the same
  - But Java separates them into two separate commands
  - Particularly relevant with fields/attributes in classes

Python Versus Java

```
class Fraction(object):
    _numerator = 0    # int, hidden
    _denominator = 1  # int > 0, hidden
    ...
    @property
    def numerator(self):
        """Numerator value of Fraction
        Invariant: must be an int"""
        return self._numerator

    @numerator.setter
    def numerator(self,value):
        assert type(value) == int
        self._numerator = value
```

```
public class Fraction {
  // Field declarations; NOT assignments
  private int numerator;    // invariant: int
  private int denominator;  // invariant: > 0
  ...
  /** Numerator value of Fraction (getter)
   *  Invariant: must be an int */
  public int getNumerator() {
    return this.numerator;
  }
  /** Numerator value of Fraction (setter) */
  public void setNumerator(int value) {
    this.numerator = value;
  } ... }
```

Hidden; Not Hidden; "docstring"; Keyword, not argument; Return Type; Param Type; Curly braces, not indentation

Other Differences with Python

- Everything in Java **must be** inside of a class def
- Functions are static
  - Lives in class, not folder
  - Do not need an object
- Function call has form <class>.<function-call>
  - Treat the class just like a you would a module
  - **Lecture 8 Example**: Anglicize.anglicize(100)

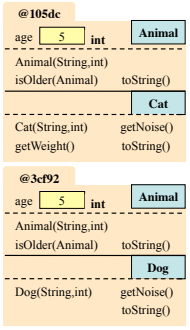
```
public class Anglicize {

  /** Returns: English equivalent of n. */
  public static String anglicize(int n) {
    if (n < 1000) {
      return anglicize1000(n);
    }
    // n >= 1000
    String suffix = "";
    if (n % 1000 != 0) {
      suffix = " "+anglicize1000(n % 1000);
    }
    return anglicize1000(n/1000) + suffix;
  }
  ...
```

Static Type vs. Dynamic Type

- Types need not agree
  - `Animal a = Cat("Felix",5);`
  - But contents must be an **instance of** variable type.

    a **@105dc** Animal
- Compiler only knows static type
  - Dynamic is runtime property
  - Uses static type to check for errors
  - This is illegal (will not compile):

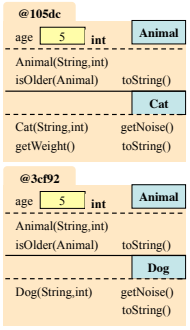
    `a.getWeight()`



Casting Up and Down the Class Hierarchy

- How casting works in Java
  - (int) (5.0 / 7.5)
  - (double) 6
  - double d = 5; // automatic cast
- Can also cast class types:
  - `Animal h = new Cat("N", 5);`
  - `Cat c = (Cat) h;`

The Class Hierarchy
(→ means "extends" or "is a kind of")



Static Type vs. Dynamic Type

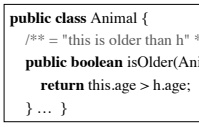
```
public class Animal {
  /** = "this is older than h" */
  public boolean isOlder(Animal h) {
    return this.age > h.age;
  } … }
```

```
Cat c = new Cat("C", 5);
Dog d = new Dog("D", 6);
c.isOlder(d)    ?????
```

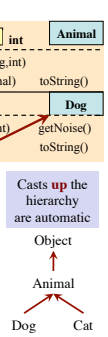
Dynamic type of h:
- Type of the object/folder
- Matches call to definition

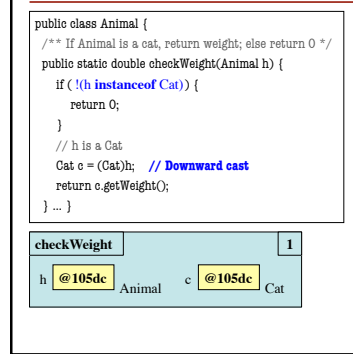
Static type of h:
- Type that is declared
- Checks if call is legal

Casts **up** the hierarchy are automatic



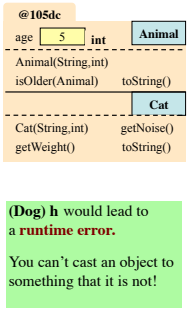
cast from Dog to Animal, automatically





Casting Down the Class Hierarchy

```
public class Animal {
  /** If Animal is a cat, return weight; else return 0 */
  public static double checkWeight(Animal h) {
    if ( !(h instanceof Cat)) {
      return 0;
    }
    // h is a Cat
    Cat c = (Cat)h;   // Downward cast
    return c.getWeight();
  } ... }
```

(Dog) h would lead to a **runtime error.**

You can't cast an object to something that it is not!