UNIVERSITY OF
TORONTO
MISSISSAUGA

Java Basics

What is Java?

Version 1.0 released in May 23, 1995 by Sun Microsystems (now owned by Oracle)

Designer: James Gosling a.k.a Dr. Java (Canadian)

Adopted pretty quickly by many major players in software industry, such as IBM.

Some languages are compiled

Before running, the human code is translated into machine code instructions which can be understood by the hosting computer's hardware (and is hardware dependent).

When running, the computer hardware executes the machine code instructions.

Languages that are compiled include C, C++

Some languages are interpreted

Nothing done before running.

When running, the human code is translated and executed on the go, one statement after another.

Languages that are interpreted include Python, Perl

Java is a something in between

Write Once, Run Anywhere

All computers first install a **Java Virtual Machine (JVM)**

Before running, the human code is translated into **bytecode**, a.k.a. the machine code for the Java Virtual Machine. This byte code can be run on **any machine** that has a JVM installed.

When running, the JVM executes the bytecode instructions.

Because Java programs run within the JVM, we never need to worry about **hosting computer's hardware.**

Compiling Java

Note that you can compile Java code on your machine, too. Say you have a file HelloWorld.java. **Compile it into bytecode** by typing this at the command line of your terminal:

$ javac HelloWorld.java

This generates a bytecode file called HelloWorld.class.

Run the compiled code in the JVM by typing

$ java HelloWorld.class

Anyone else with a JVM installed can run this bytecode too!

Some other interesting Java facts

It’s still popular, check out tiobe index.

It’s faster than Python: link.

It’s used to write Web Applications (Spring), Desktop Applications (Acrobat Reader), Mobile Applications (Android), and can scale to enterprise applications.

It’s been designed to make it easy to:

- Work in groups
- Build software that’s malleable and extensible

You'll learn it quickly

You already know Python; that means you know most of the syntax already.

You just need to learn about the differences and you're ready to go!

8 months (CSC108 + CSC148) vs <= 8 weeks

UNIVERSITY OF TORONTO MISSISSAUGA

Differences between Python and Java

1. It's strongly typed

Python: one can change the type of a variable dynamically.

Java: every variable must be first declared with a particular **type**, and you're not supposed to change the type afterwards.

```python
# Python
x = 3
# x is an integer 3

x = “Hello World!”
# x is now a string
```

```java
// Java
int x = 3;
// x is declared as an integer

x = "Hello World!"
// Error: not OK!
```

What are some affordances of a strongly typed language?
What are some drawbacks?

Differences between Python and Java

2. It uses braces to define blocks

In Python, scopes are indicated using indentation.

In Java, we use braces {}.

```python
# Python
if a > b:
  a = b
  b = c
```

```java
// Java
if (a > b) {
  a = b;
  b = c;
}
```

Differences between Python and Java

3. The syntax for control structures is different

```java
if (condition1) {
    //statement;
} else if (condition2) {
    //statement
} else {
    //statement;
}

for (int i = 0; i < n; i++) {
    //statement;
}

do {
    //statement;
} while (condition1);

while (condition1) {
    //statement;
}
```

Java

```python
if condition:
    # statements
elif:
    # statements
else:
    # statements

for i in range(n):
    # statements

while condition:
    # statements
```

Python

UNIVERSITY OF TORONTO MISSISSAUGA

Differences between Python and Java

4. There's some variation in operators

```java
int a = 6;
int b = 0;

System.out.println("a value is " + a);
System.out.println("b value is " + b);

b= ++a; //what would happen if we replaced this line with b=a++ .... ?
System.out.println("a value is " + a);
System.out.println("b value is " + b);
```

Java supports prefix and postfix integer increments; Python does not! But both support compound assignments (i.e. a += 1).

Other operations also have subtly different syntax, e.g. Boolean and ternary operators:

Logical Operators in Java

```java
boolean x = true;
boolean y = false;

// Output: x and y is false
System.out.println("x and y is " + (x && y));

// Output: x or y is true
System.out.println("x or y is " + (x || y));

// Output: not x is false
System.out.println("not x is " + !x);
```

Logical Operators in Python

```python
x = True
y = False

# Output: x and y is False
print('x and y is',x and y)

# Output: x or y is True
print('x or y is',x or y)

# Output: not x is False
print('not x is',not x)
```

```java
// If foo is selected, assign selected foo to bar.
If not, assign baz to bar.
Object bar = foo.isSelected() ? foo : baz;
```

```java
// If a > b, assign x to result.
If not, assign y.
result = x if a > b else y
```

UNIVERSITY OF TORONTO MISSISSAUGA

Differences between Python and Java

5. Methods and attributes are associated with classes

```python
# Python
print("Hello, world!")
```

```java
// Java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

Java is fundamentally an object-oriented programming language, meaning **objects** are the basic “units” of programs we will write!

The template or blueprint for an object is called a **class**, and classes, once compiled, will create *.class* files.

Methods and **attributes** will be bottled up, or *encapsulated*, within these classes.

Some other differences between Java and Python:
http://anh.cs.luc.edu/331/notes/JavaVsPython.html

UNIVERSITY OF TORONTO MISSISSAUGA

More on Java's types

Primitive types:

byte (8-bit), short (16-bit), int (32-bit), long (64-bit), float (32-bit), double (64-bit)

char (16-bit)

boolean (represent 1 bit of information, but its size is not precisely defined)

Class types (e.g. types of *objects*, provided by Java or user defined):

e.g., String, Integer, Double, Balloon, Person...

Variables must respect their type!

- Will this line of code compile? Why or why not?

```
double x = 3;
```

- Will this line of code compile? Why or why not?

```
int y = 3.0;
```

Auto-conversion of primitive types

Literals like 3 are by default of type **short int**.

Literals like 3.0 are by default of type **double**.

Sometimes, when you assign a value of type A to a variable of type B, Java **automatically** converts the value to type B.

double x = 3; // x will be 3.0, auto-conversion: int -> double

You can auto-convert in some directions, but not others:

byte -> short -> int -> long -> float -> double

char -> int -> long -> float -> double

boolean -> no other types

Why do you think this might be?

Auto-conversion of primitive types

Any other conversion that does not follow the directions enumerated on the prior slide must be explicitly **casted**.

```
float x = (float)2.07;
long x = 207; short y = (short)x;
short x = 207; long y = x; // auto-converted, no need for casting
```

Auto-conversion of primitive types

Converting from **shorter** to **longer** types is done **without loss of information**

converting **byte** x = 7 to a **short int**:

0000 0111 -> 0000 0000 0000 0111

Converting from **longer** to **shorter** types involves **truncation**:

convert **short int** x = 259 to a **byte**:

0000 0001 0000 0011 (short int value 259) -> 0000 0011 (byte value 3)

Auto-conversion of Objects

This is possible, but it will help us to become more familiar with **classes of objects** first.

- Some predefined classes for objects include: String, Vector, Stack, Hashtable, Integer, and many others. You'll get more experience with some these in your lab next week.
- We can make our own classes for objects, too: Balloon, Person, etc.

All classes of object in Java are subclasses of the class **Object**:

https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/lang/Object.html

Auto-conversion of Objects

Auto-boxing of primitive types

The standard Java library includes objects, or classes, that act as **wrappers** around **primitive data types** (e.g. Integer, Boolean, etc).

Auto-boxing involves automatically converting primitive types to their corresponding object wrapper classes at compilation.

When the conversion goes the other way, this is called **unboxing**.

```
Integer myInt = 5; // autoboxing
int i = myInt; // unboxing
Integer anotherInt = new Integer( value: 5); // no autoboxing
```

What might be an advantage of an **Integer** object over a **primitive int**?
What might be an advantage of a **primitive int** over an **Integer** object?

See https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/lang/Integer.html

But wait, what's an Object?

An **object** is an **instance** of a **class**

A **class** is like a **blueprint**

A **class** defines an **abstract** (reusable) data type

A **class** can contain **attributes (aka variables)** and **methods**

A **class** can be **inherited** to create a subclass

Objects are Constructed

In Python, a **class constructor** is a method named **__init__**. In Java, it is a method with the **same name as the class**. It has **no return type**, not even void.

Within **constructors** and class **methods**, the keyword "**this**" functions like Python's "**self**"

```
public class SimpleObject {

    /**
     * A simple object class, to illustrate the basic anatomy of a Java Class
     */
    1 usage
    private int sumTo;

    /**
     * Class constructor that specifies an integer value for the sumTo attribute.
     */
    2 usages
    public SimpleObject(int sumTo) {
        this.sumTo = sumTo;
    }

}
```

Constructors initialize class **attributes (aka non-static* variables)**. Attributes and methods have **access modifiers.**

A **public** attribute or method can be accessed by objects outside of the class.

A **private** attribute or method cannot.

** We will talk more about the static keyword next class!*

Multiple Constructors

```
public Balloon(String color)
{
   this.amount = 0;
   this.capacity = 10;
   this.popped = false;
   this.color = color;
}
```

```
public Balloon(String color, int capacity)
{
   this.amount = 0;
   this.capacity = capacity;
   this.popped = false;
   this.color = color;
}
```

```
Balloon left = new Balloon("red");
Balloon right = new Balloon("red", 15);
```

A class can have **multiple constructors** with different signatures.

- Java will call the constructor with the matching signature.
- Constructors can be called outside of a class, to create an **instance** (an individual object member) of that class. The instan will be assigned to an **instance variable.**
- **this** is like Python's **self**

Cascading Constructors

```
public Balloon(String color, int capacity)
{
    this(color); // call the other  constructor
    this.capacity = capacity;

}
```

Each **constructor** can reference other constructors, to create a cascade of constructors.

Class Methods

```java
public class SimpleObject {

    /**
     * A simple object class, to illustrate the basic anatomy of a Java Class
     */
    2 usages
    private int sumTo;

    /**
     * Class constructor that specifies an integer value for the sumTo attribute.
     */
    2 usages  Shuhao Dong
    public SimpleObject(int sumTo) {
        this.sumTo = sumTo;
    }

    /**
     * Public method that returns the value this object will sumTo.
     *
     * @return the value this object will sumTo.
     */
    1 usage  Shuhao Dong *
    public int getSumTo() {
        return this.sumTo;
    }

}
```

A given class may have other public methods, beyond the constructors, too!

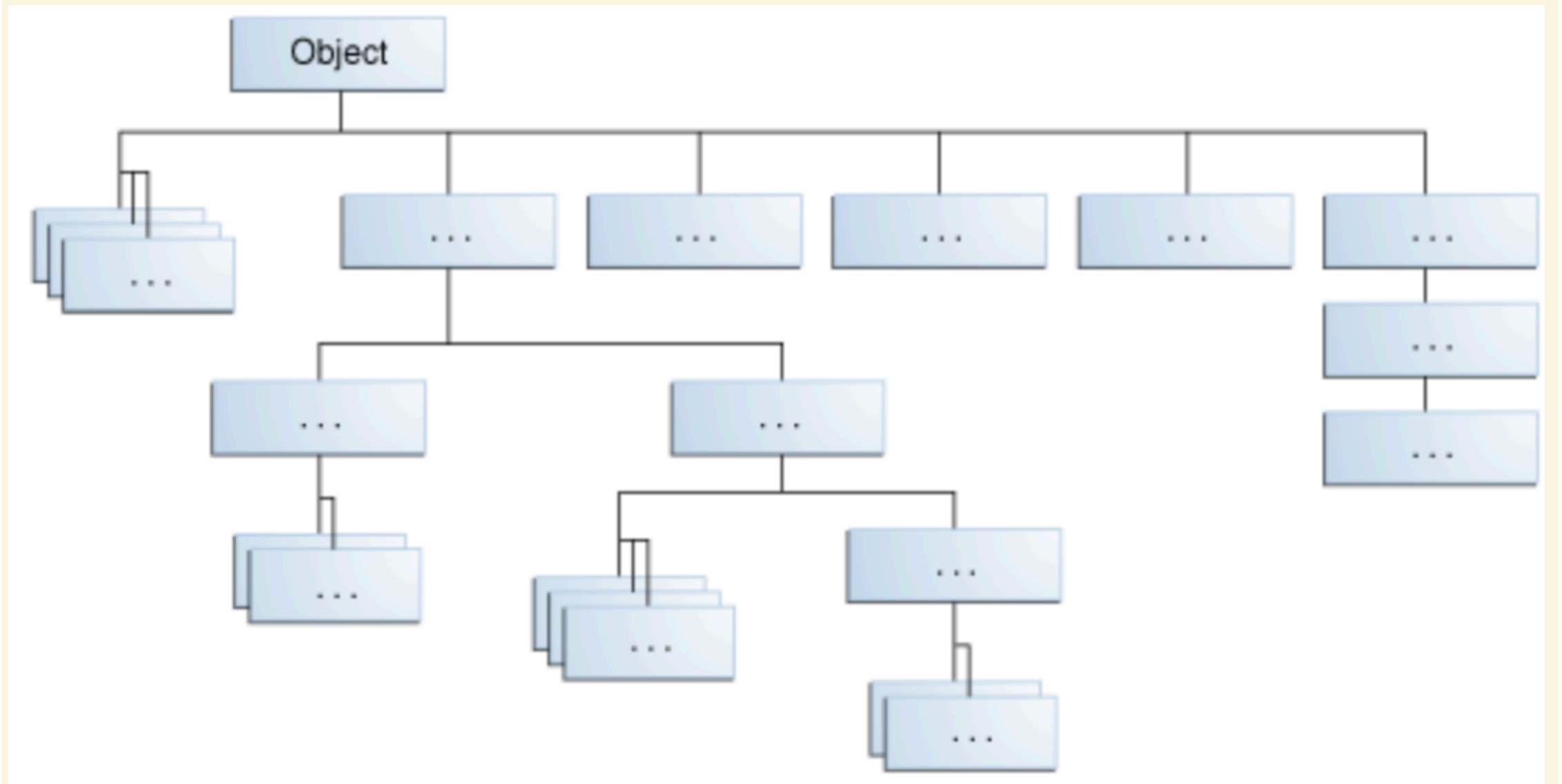
The method getSumTo is a **public non-static* class method.** Methods may have **parameters** and/or **local variables.**

Non-static methods will be called via a given instance of the class SimpleObject.

** We will talk more about the static keyword next class!*

```java
SimpleObject s = new SimpleObject( sumTo: 10);
System.out.println("This object will sum to " + s.getSumTo());
```

All Objects Inherit from Java's Object Class



All Objects Inherit these methods from the Object Class

toString

equals

hashCode

More on this shortly….

Inheritance

A class can **inherit** from another class. In Java, inheritance is indicated by use of the **extends** keyword.

Abstract classes **must** be extended.

- A sub-class of an abstract class can still be abstract.

We say a Child Class ***IS-A*** (subtype of) the Parent Class.

- E.g., Dog is a subtype of (or **IS-A**) Animal.

Abstract Classes

Not all classes are the same. Some are **abstract**!
An **abstract** class:

- is **designed** to be inherited. It **CANNOT** be instantiated, because it is not ***concrete***.
- may contain some implemented class **methods**
- may contain some **attributes** (aka class variables)
- may contain methods that are **NOT** implemented but must be implemented by classes that inherit from it (sub-classes)

Abstract Class Example

```
public abstract class Animal {
    1 usage
    private String name;

    2 usages
    public Animal(String name) {
        this.name = name;
    }

    1 implementation
    public abstract void makeSound();

    public void eat(String food) {
        System.out.println("Yum! I am eating " + food + " and I like it.");
    }

}
```

Example: Extending a Class

```java
public class Dog extends Animal {

    public Dog(){
        super( name: "Dog");
    }

    public Dog(String name){
        super(name);
    }
    @Override
    public void makeSound() {
        System.out.println("Woof, woof.");
    }

    public void fetchStick() {
        System.out.println("Look at me, I am fetching a stick.");
    }
}
```

Java Packages

A **package** is a namespace, or module, which contains several classes.

Packages can be used to organize classes and interfaces.

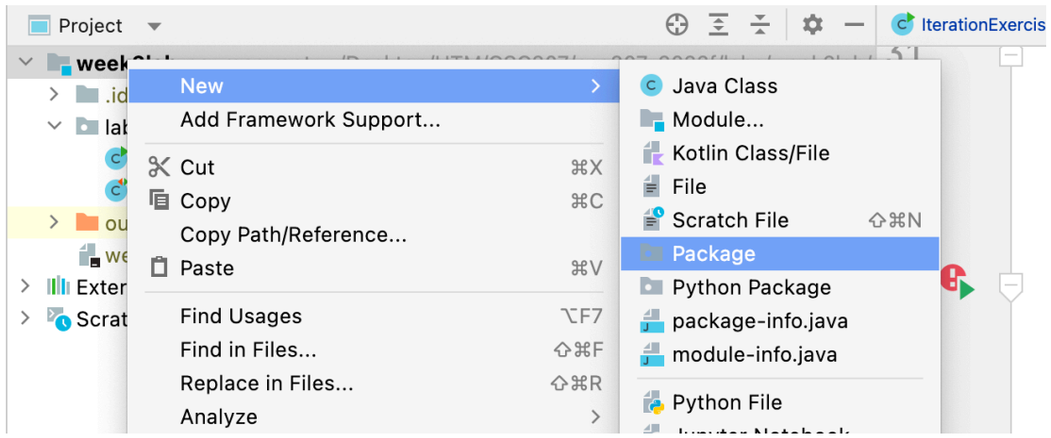
We have two types of packages in Java:

- **built-in packages (e.g. package collections in java.util.* or java.lang.*)**

- **user defined packages (e.g. lab02)**

You will be working within a *package* during your next lab.

You can create a package in IntelliJ as follows:



Packages, Libraries and Frameworks

Libraries are collections of packages with classes that we may want to use again and again.

A ***Framework*** contains groupings of useful packages, too. But a Framework **also** defines flow of control in your program.

java.lang.Math and **java.util.Random** are *Libraries*
JUnit5 is an example of a *Framework*
JavaFX is another example of a *Framework*

Access Modifiers

The **protected** keyword restricts access to an object's methods and attributes to classes that exist **within the same package**, and to all classes that are **inherited** from the object class.

Access Modifiers

Modifier	Class	Package	Subclass	Global
Public	✓	✓	✓	✓
Protected	✓	✓	✓	✗
Default	✓	✓	✗	✗
Private	✓	✗	✗	✗

The **default**, or **lack of an access modifier**, restricts access to within the **package** of the object's class.

On the keyword “static”

When related to a class **attribute**, the **static** keyword means a single copy of the attribute is shared across *all instances of that class.* A **static attribute** may also be called a **class variable**.

```
class Counter {
    2 usages
    static private int count = 0;//this will be shared across all Counters
    1 usage
    private int id;

    Counter() {
        this.id = count; //assign this object a unique ID
        count++;//increment the shared counter
    }
}
```

When related to a class **method**, it means the method belongs to a **class**, not an **object**. We can call these methods **without creating an object instance.**

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Passing arguments to methods

```
Balloon b1 = new Balloon();
int someAir = 5;
// someAir is passed by value
b1.inflate(someAir);

Person p1 = new Person();

String name = new String("Ilir");
// name is an object, hence passed by reference
p1.setName(name);
```

Arguments supplied method parameters may be **references to objects,** or **values**. More on this at https://www.javadude.com/articles/passbyvalue.htm

Primitive types are passed by value. Objects are passed by reference.

Primitive types, Wrapper classes and Strings are **IMMUTABLE**.
Many other kinds of objects (like Hashtables) are **MUTABLE**.

Can someone remind us what this means?