A (Re-)Introduction to Java for C++/C Programmers

Why Java?

- Developed by Sun Microsystems (now Oracle) beginning in 1995.
- Conceived as a better, simpler version of C++.
- Closely imitated by Microsoft's C#.

Notable Characteristics

- Supports object-oriented programming.
- Strong static type safety.
- Garbage collection; strong memory security at runtime.
- Highly portable, via bytecode intermediate representation.
LEARNING A LANGUAGE: RESOURCES

Sun documentation URL (for version 1.11):
https://docs.oracle.com/en/java/javase/11/

Textbooks and Tutorials

(Older version for Java 5/6 is *Thinking in Java, 4th ed.* at https://archive.org/details/TIJ4CcR1)
(book, for Java 5)
- Many, many other books...
- Java tutorials, on web at Oracle site.

Language and Library Specifications and Manuals

- *Java Language Specification*, (on web at Oracle) (Very hard reading!)
- *Java Platform API Specification* (version-specific; on web at Oracle)
- *Java JDK Tools and Utilities* (version-specific; on web at Oracle)
Language has had several revisions, usually identified by JDK version numbers.

- Currently at version 13 (approximately; depends on platform)
- Course will assume version 11.
- Last major language changes occurred at version 8.
- Confusingly, Sun/Oracle used to refer to version $n$ as version $1.n$. 
A small collection of types are completely “built-in.”

- **Integral types** (sizes same on all platforms):

  byte (8 bits, signed)
  short (16 bits, signed)
  int (32 bits, signed)
  long (64 bits, signed)
  char (16 bits, unsigned – uses Unicode representations)

  Integer arithmetic is always performed in 32 bits, unless a long operand is involved, in which case it is done in 64 bits. Values of smaller sizes are automatically promoted to larger ones where needed, but conversions the other way require explicit casts, e.g.,

  ```java
  short c;
  short d = (short) (c+1);
  ```

- **Floating-point types** are float (32 bits) and double (64 bits) in IEEE format.
Booleans are not integers! They form a distinct type boolean with two literal values true and false.

Boolean operators are the same as in C/C++, except that you cannot do arithmetic on boolean values. Booleans are used to govern if, for, do, and while statements as usual.
Declaring variables of primitive types is roughly as in C/C++, but:

- Compiler must be able to convince itself that all variables have been assigned a value before they are used, e.g.:

```java
int i;
int j = 0;
if (j == 0) // always true, but
  i = 1; // compiler doesn’t know that!
i = i + 1; // compile-time error!
```

Simple approach: always initialize variables in declarations!

- There are no `const` declarations, but variables can be declared `final`, which means that they can only be assigned to once, e.g.:

```java
final int j = 0; // j is constant
final int i;
i = 100; // i is constant from now on
i = i + 1; // compile-time error!
```
Expressions and Statements

These are mostly the same as in C and C++.

One difference: Java has no `goto` statement. (But you never use that anyway, right?)

Instead, it has a labeled `break` statement, which jumps to the end of labeled enclosing control structure (`for`, `while`, `do`, or `switch`). Unlabeled `break` jumps to the end of the innermost enclosing control structure, as usual. For example...
The code

```java
int i = 0;
outer:
    while (true) {
        System.out.print (i);
        switch (i) {
            case 0:
                i++;   // falls through
                case 1:
                    i += 2;
                    break;  // break out of switch
                case 3:
                    break outer;  // break out of while
        }
    }
System.out.print (99);
```

prints 0 3 99.
Every value in Java that does not belong to a primitive type is an object. Each object is an instance of some class, which is much like a C++ class. Each class definition can contain fields and methods (i.e., functions). Constructors are a special kind of method used to create new instances; they typically initialize the values of the fields.

Each instance object contains its own copy of the field contents (ordinarily – more below!)

As in C++, methods (including the constructor method) can refer to the fields of the object for which they were invoked. Unless specifically restricted, fields can also be read or written from outside the class definition. (There are several possible kinds of restrictions; the details are similar but not identical to C++.)

An example using objects....
/ A class to represent points in the plane
class Point {
   // Fields contain the point’s coordinates
   int x;
   int y;

   // Constructor for creating points
   Point (int xInit, int yInit) {
      x = xInit; y = yInit;
   }

   // Method on points
   void translateY (int deltaX) {
      x += deltaX;
   }

   ...  
Point p = new Point(3,4);  // create new point object in p
p.translateX(7);  // use method to change fields inside object
int x = p.x + p.y  // extract current values of fields (x = 14)
**Objects Live at Abstract Locations in the Heap**

Objects are always* heap-allocated, i.e., `new` acts much like in C++.

Objects are never explicitly deallocated. Instead, the Java runtime system automatically deallocates them when they are no longer pointed to from anywhere in the running program.

This feature is called garbage collection. It has the huge advantage that the programmer doesn’t have to worry about deallocation, and can’t introduce dangling pointers (pointers that still point to deallocated objects) or space leaks (objects that are still allocated but no longer pointed to).

Also, object addresses are abstract; you can’t do pointer arithmetic on them. (This is crucial for maintaining memory safety.)

(* Well, almost always. Clever compilers may be able to avoid the cost of heap allocation in certain special cases. Since addresses are abstract, you can’t really tell.)
Variables of object type (like \( p \) above) **always** contain **references** (or pointers) to objects, rather than objects themselves. That is, the Java declaration

\[
\text{Point } p;
\]

is like the C++ declaration

\[
\text{Point } *p;
\]

Similarly, the Java notation

\[
p.x
\]

corresponds to the C++ notations

\[
(*p).x
\]
\[
p->x
\]

Unlike in C++, there is simply no way to declare storage for the object itself (e.g., on the stack, or inside another object). This is because fixed-size storage doesn’t work well for object-oriented programming...
class Link {
    Point p;
    Link next;
    Link (Point pInit, Link nextInit) {
        p = pInit; next = nextInit;
    }
}
Point p1 = new Point(0,1);
Point p2 = new Point(2,3);
Link x = new Link(p1, new Link (p2, null));

Since both fields in Link are really references (pointers), the representation of x in the heap looks like...
Example of References (continued)

Note that the special value `null`, representing an “empty” or “missing” object, is a legal value for all class types.
COPYING IS SHALLOW

It is crucial to remember that assigning an object variable just assigns a pointer.

    Point p1 = new Point(0,1);
    Point p2 = p1; // p1, p2 point to same object
    p1.x = 2; // now p2.x == p1.x == 2

To copy the contents of an object, we must copy the individual fields one at a time. In Java this is called cloning.
Both fields and methods of a class can be declared **static**.

```java
class Stuff {
    static int counter = 0;
    static int sqr(int x) { return x * x; }
}

Stuff.counter++;
int z = Stuff.sqr(33);
```

A **static field** has only one copy, no matter how many objects of the class are created. In effect, it is like a global variable.

A **static method** has no associated object; it operates only on its arguments (and possibly static fields from its own class or other classes).

Both static fields and methods are named using the dot notation, just as in ordinary field and method references. But for static members, the class name is what appears before the dot, rather than an expression that identifies an object. In fact, the semantics of static and non-static members are very different, so the similarity in notation for accessing them can lead to confusion.
Many classes contain both static and non-static members, but some classes contain only static members. Such classes are never used to create instances (there would be no point, since the instances would contain no data); they just serve to organize the name space of top-level global variables and functions.

Example: the `System` class defined in the Java library contains useful top-level things, like

```java
static PrintStream out; // standard output
```

which we can use to print things to the terminal, using the (non-static) methods defined on `PrintStream`, e.g.

```java
System.out.println("hello world");
```
Unlike in C++, all functions in a Java program are class methods (possibly static).

All arguments are passed **by value** just as in C. However, remember that object argument **values** are in fact **references**. For example:

```java
class Foo {
    static void foo(Point p) {
        p.x = 0;
    }
}
...
Point p = new Point(10,20);
Foo.foo(p);
// p.x now = 0
```

If a method has a non-**void** return type (primitive or class), the compiler must be able to convince itself that **all** possible paths through the function lead to a **return** statement with a value of appropriate type.
A single class can define multiple methods with the same name, provided that their arguments are of different types. Such methods are said to be overloaded. The choice of which method will be called is made at compile time, based on the types of the actual arguments provided at the call site.

```java
class Foo {
    static int foo(int i) { return 0; }
    static int foo(double d) { return 1; }
}
...
Foo.foo(3) + Foo.foo(3.14)
// evaluates to 0 + 1 = 1
```

Methods cannot be overloaded based on return type.
Strings are (almost ordinary) objects of library class `String`. They are immutable, i.e., their contents can never change. We can use class member functions to access characters inside string.

There is another library class `StringBuilder` for handling mutable sequences of characters.

Strings are not arrays of characters.

Special language-level support for strings:

- Literal string constructors: "abc" creates a new `String` object.

- Applying the + operator to a string acts like string concatenation, e.g.,

  ```java
  String c = "abc" + "def"
  ```

  makes `c` a new 6-character string object.

Since + with a string operand is overloaded on all the primitive types, we can write things like: "2+2=" + (2+2) // produces string "2+2=4"
Arrays are (slightly special) objects!

Each array contains elements of some primitive type or class, e.g., int[], char[], String[], int[][].

For compatibility with C/C++, can declare array variables in two equivalent ways:

```java
int[] a; int a[];
```

As with other objects, an array variable is just a reference to an array; to create the actual array (with contents) we must use `new`

```java
int[] a = new int[10];
```

or an explicit initializer

```java
int[] a = {1,2,3,4,5,6,7,8,9,10}
```
• The length of an array is fixed forever when the array is created; it can be retrieved using the built-in final instance variable `length`.

• All loads and stores on the array are checked against the array bounds; out-of-bounds index causes an exception to be raised.

• As in C/C++, multi-dimensional arrays are just one-dimensional arrays containing arrays as elements.
static double[] vAdd (double[] v1, double[] v2) {
  double[] r = new double[v1.length];
  for (int i = 0; i < v1.length; i++)
    r[i] = v1[i] + v2[i];
  return r;
}

public static void main(String[] argv) {
  double a[] = {1.1,2.2,3.3};
  double b[] = vAdd (a,new double[] {4.4,5.5,6.6});
  ...
}
The **package** is Java’s (almost) top-level code structuring mechanism.

- A package is just a namespace containing the definitions of one or more classes.
- Packages can include Sun’s own library, other vendors’ libraries, and your own local code.
- Important library packages include `java.lang`, `java.util`, and `java.io`.

Since Java 9, packages can live inside even higher-level collections called **modules**.

- Most standard library packages live in module `java.base`. 
To refer to elements of a package, you can use **fully qualified names**, e.g.,

```java
java.util.LinkedList myList =
    new java.util.LinkedList();
```

Better: import the package name (or specific class names) that you need:

```java
import java.util.*;  // at top of file
...
LinkedList myList = new LinkedList();
```

Package `java.lang` is always implicitly imported, so you never need to qualify its class names.

By default, classes **you** define go into a default anonymous package, which is fine for now.
Standard Sun JDK set-up for building Java applications:

- Source file (.java extension) contains one or more class definitions.
- Use compiler executable (javac) to compile the source file into byte-code files (.class extension). (You get one .class file for each defined class. Name depends on class name, not on .java file name.)
• Byte-codes are an intermediate format that must be executed by a **Java virtual machine (JVM)** executable (**java**). You must specify the name of the class (**without** .class extension!) containing desired **main** method (more below).

• JVM may interpret byte-codes directly, or may internally compile them to machine code and then execute that code. Much more about this later...

• Both **javac** and **java** access library packages (in standard location you don’t have to specify) to get executable code and also static typing information (equivalent of C .h file info). Library may be in .class or .jar (Java archive) files.

• Both **javac** and **java** read and write in your current directory.

• You can direct them to look for input files in other directories by setting your **classpath**, either using the shell variable **CLASSPATH** or via a flag. For now, you should make sure the **CLASSPATH** variables is **NOT** set; some other packages on PSU CS may set it incorrectly.
Every application must define some class with a method having this signature:

    public static void main(String[] argv)

(where the argument name must be present, but is arbitrary).

When the application starts up, `main` is invoked with the argument set to an array containing the command line parameters.
Suppose file `myapp.java` contains the following (complete) program

```java
class MyApp {
    public static void main(String [] argv) {
        for (int i = 0; i < argv.length; i++)
            System.out.println(argv[i]);
    }
}
```

This can be compiled to bytecode as follows:

```
% javac myapp.java
%
```

This produces a file `MyApp.class`, which can be executed thus:

```
% java MyApp p d q
p
d
q
%
```