CENG 443
Introduction to Object-Oriented
Programming Languages and Systems

01 Introduction to Java

History

- Java was started as a project called “Oak” by James Gosling in June 1991.
    - Sun Microsystems – a hardware company
- The goal was to develop software for consumer electronics that was portable so that it could be switched quickly to new hardware
- The language was first called Oak
    - After an oak tree outside Goslings window
    - However, there was already a language named Oak.
- The team went for a “Coffee break” and named the language Java
    - In recognition of the role that caffeine plays in software development

The Goal

- The goal was to implement a language much simpler than C/C++.
- Java was developed with the goal to implement
  “WORA: Write Once, Run Anywhere” programming model.
- However, the original project was cancelled.
- As the Internet emerged,
  the focus shifted towards a PL
  that could run on various platforms.

Philosophy

- The Java programming language was built on the following five philosophies.
    1. It will use the Object-oriented programming methodology
    2. The same program should be executable on multiple operating systems.
    3. Built-in support for using computer networks.
    4. Designed to execute code from the remote sources securely.
    5. It should be easy to use, take the good features of Object-oriented programming.

Java Versions

Version	Release Date
JDK 1.0	23rd January 1996
JDK 1.1	2nd February 1997
J2SE 1.2	4th December 1998
J2SE 1.3	8th May 2000
J2SE 1.4	13th February 2002
Java SE 5	29th September 2004
Java SE 6	11th December 2006
Java SE 7	28th July 2011
Java SE 8 (LTS)	18th March 2014
Java SE 9	21st September 2017
Java SE 10	20th March 2018
Java SE 11 (LTS)	25th September 2018
Java SE 12	19th March 2019
Java SE 13	17th September 2019
Java SE 14	17th March 2020
Java SE 15	16th September 2020
Java SE 16	16th March 2021
Java SE 17 (LTS)	14th September 2021
Java SE 18	22nd March 2022
Java SE 19	20th September 2022
Java SE 20	21st March 2023
Java SE 21 (LTS)	19th September 2023
Java SE 22	March 2024

What is Java?

- According to Sun Microsystems White Paper

Java is simple, object-oriented, network-savvy, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded, dynamic language

Java is Simple

- Java is partially modeled on C++ but greatly simplified and improved
- Pointers & multiple inheritance often make programming complicated
- Java replaces the multiple inheritance in C++ with a simple language construct
- Java eliminates pointers
- Java uses automatic memory allocation and garbage collection

Java is Object-Oriented

- It is an Object-Oriented programming language
- It has constructs to implement **encapsulation**, **polymorphism** and **inheritance**

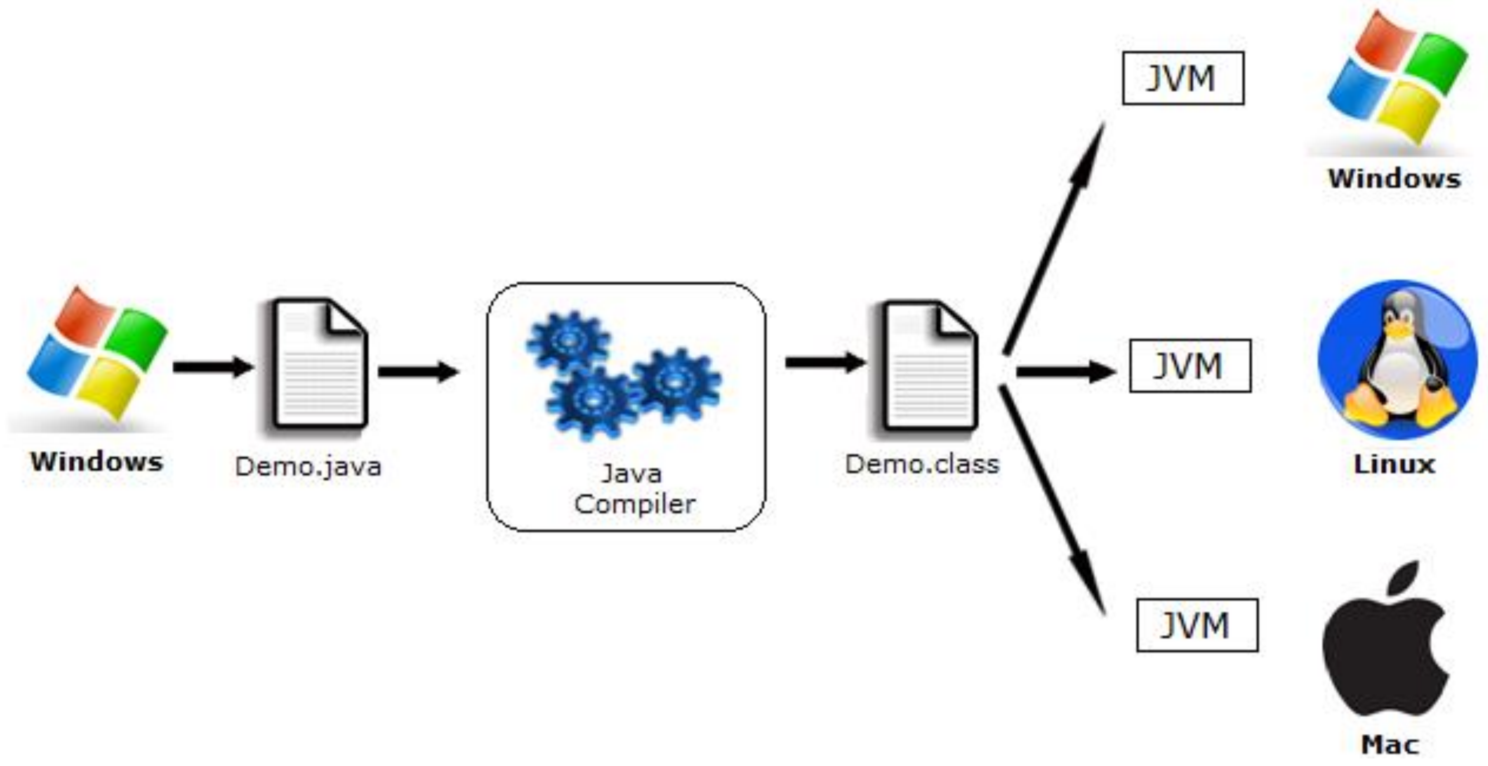
Java is Distributed

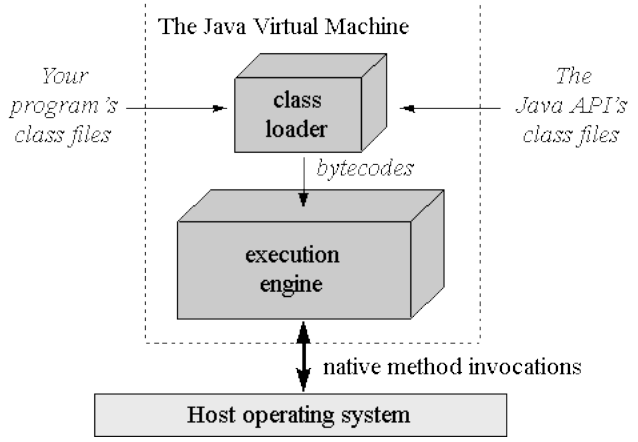
- Distributed computing involves several computers working together on a network.
- Networking capability is inherently integrated into Java so writing network programs is like sending and receiving data to and from a file.

Java is Compiled and Interpreted

- The Java platform has a compiler that translates Java source into a form called bytecodes
- Bytecode is an architecturally neutral representation of code written in the Java programming language.
- Bytecode is machine-independent and can run on any machine that has a Java interpreter.
- The bytecode, rather than Java source code, is interpreted when you run a Java program.

Java Execution Model





Java is Robust

- Robust means reliable.
- Java has eliminated certain error-prone programming constructs found in other languages.
- It doesn’t support pointer arithmetic, for example, thereby eliminating the possibility of overwriting memory and corrupting data.
- Java has a runtime exception-handling feature to provide programming support for robustness.
- The programmer must write the code to deal with exceptions.

Java is Secure

- Java security is based on the premise that *nothing should be trusted*.
- If you download a Java applet and run it on your computer, it will not damage your system because Java implements several security mechanisms.
- We don’t have pointers and we cannot access out of bound arrays (you get ArrayIndexOutOfBoundsException if you try to do so) in java. That’s why several security flaws like stack corruption or buffer overflow is impossible to exploit in Java.

Java is Architecture neutral and Portable

- All Java programs must be compiled into bytecodes before the JVM can run them.
- Java programs can be run on any platform without being recompiled, making them very portable.
- There are no platform-specific features in Java (size of an int).

Java is Multithreaded

- Multithreading is the capability for a program to perform several tasks simultaneously within a program
- Used in graphical user interfaces
  - listen to an audio recording while surfing a Web page
- Used in network programming
  - a server can serve multiple clients at the same time
- Classes are provided from the base language package to create and manage threads

Java is Dynamic

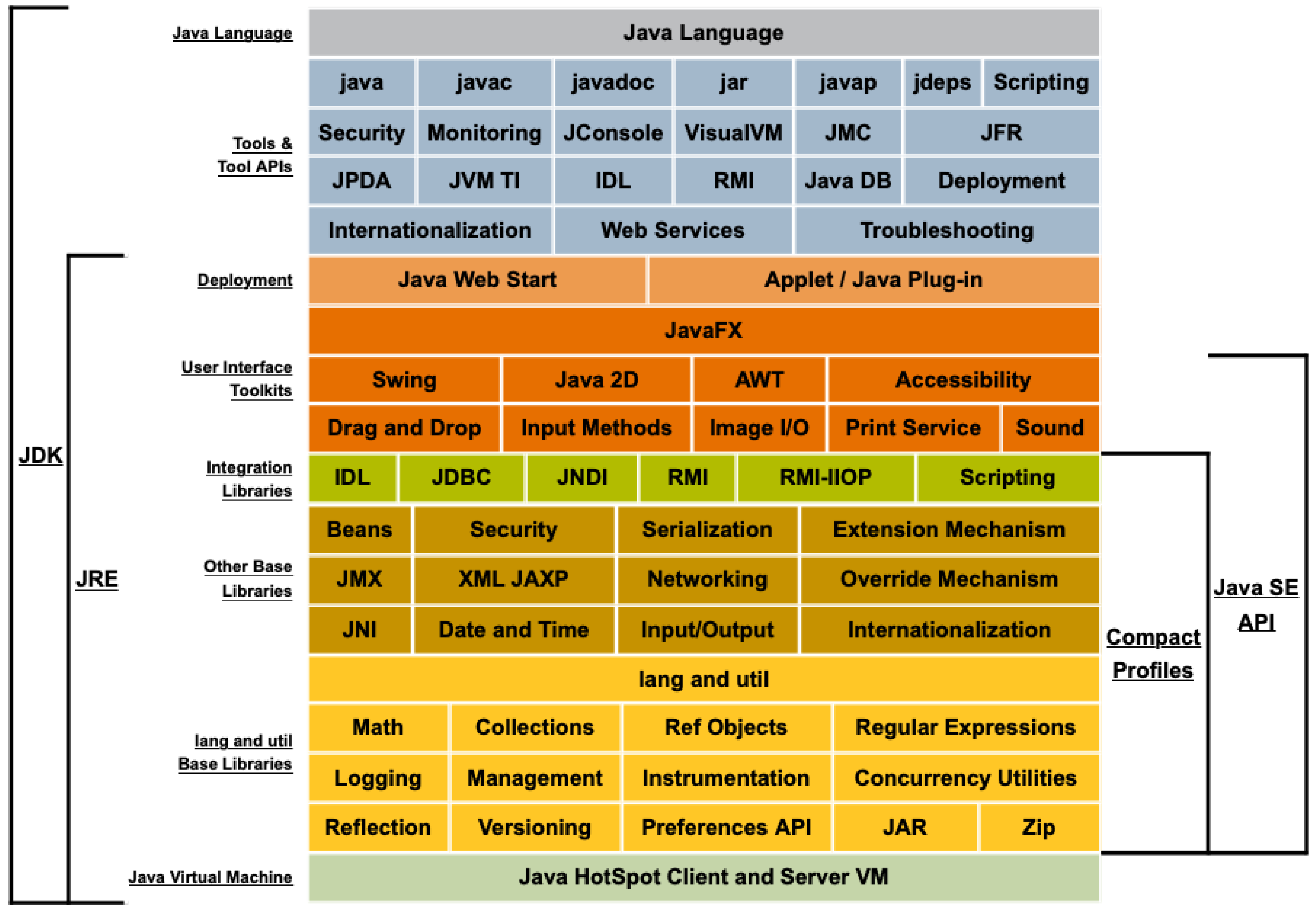
- Java was designed to adapt to an evolving environment.
- You can **freely** add new methods to a class without affecting its clients
  - Only if you follow the rules!
- Example: In the Circle class, you can add a new data property to indicate the color of the circle or a new method to obtain the circumference of the circle. The original client program that uses the circle class remains the same.
- Also, at runtime Java loads classes as they are needed.

Java’s Performance

- Java was historically considered slower than the fastest 3rd generation typed languages such as C and C++
- The main reason was using Java virtual machine (JVM) rather than directly using the computer's processor as native code, as do C and C++ programs
- Since the late 1990s, the execution speed of Java programs improved significantly via introduction of just-in-time compilation (JIT) (in 1997 for Java 1.1
- The addition of language features supporting better code analysis, and optimizations in the JVM (such as HotSpot becoming the default for Sun's JVM in 2000)
- HotSpots are parts of the bytecode executed frequently.
- Hardware execution of Java bytecode, such as that offered by ARM's Jazelle, was also explored to offer significant performance improvements.

What Java is TODAY

- It has become more than just another programming language
- Set of Technologies and tools with huge community
- An open standard



Hello World

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

- The simplest Java Program
- If you want to run a Java program you should have method called **main**.
- It is the identifier that the JVM looks for as the starting point of the java program. **It's not a keyword.**

Hello World

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

- you must use a class even if you aren’t doing OO programming

Hello World

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

- main must be public
- It is an *Access modifier*, which specifies from where and who can access the method. Making the *main()* method public makes it globally available. It is made public so that JVM can invoke it from outside the class as it is not present in the current class.

Hello World

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

- main must be static
- It is a *keyword* which is when associated with a method, makes it a class related method. The *main()* method is static so that JVM can invoke it without instantiating the class. This also saves the unnecessary wastage of memory which would have been used by the object declared only for calling the *main()* method by the JVM.

Hello World

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

- main must return void
- As soon as the *main()* method terminates, the java program terminates too. Hence, it doesn’t make any sense to return from *main()* method as JVM can’t do anything with the return value of it.

Hello World

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

- main must declare command line arguments even if it doesn’t use them

Hello World

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

- println is a method of the static attribute System.out

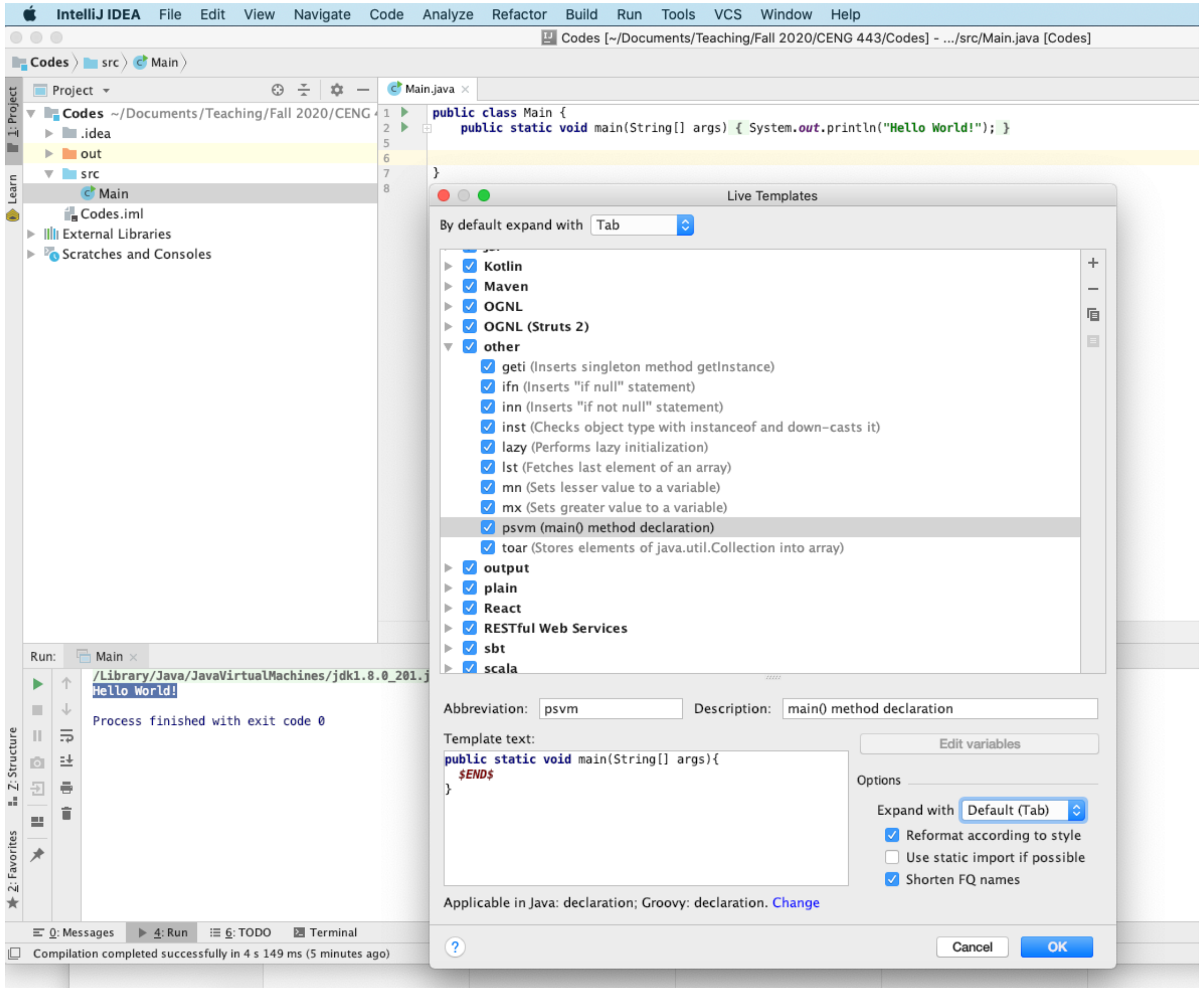
Execution is a bit complicated, too

- First you **compile** the source file
  - javac HelloWorld.java
  - Produces class file HelloWorld.class
- Then you **launch** the program
  - java HelloWorld
  - Java Virtual Machine (JVM) executes main method

On the bright side...

- Has many good points to balance shortcomings
- Some verbosity is not a bad thing
    - Can reduce errors and increase readability
- Modern IDEs eliminate much of the pain
    - Type **psvm** instead of public static void main
- Managed runtime (JVM) has many advantages
    - Safe, flexible, enables garbage collection
- It may not be best language for Hello World...
    - But Java is very good for large-scale programming!

Pick an IDE and use it to the fullest


IntelliJ IDEA File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
Codes [~/Documents/Teaching/Fall 2020/CENG 443/Codes] - .../src/Main.java [Codes]
Codes src Main
Project
Main.java
Codes ~/Documents/Teaching/Fall 2020/CENG
.idea
out
src
Main
Codes.iml
External Libraries
Scratches and Consoles
Run: Main
0: Messages
4: Run
6: TODO
Terminal
1: Project
Learn
Z: Structure
2: Favorites
Live Templates
Cancel
OK
Edit variables

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

}
```

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_201.j
Hello World!

Process finished with exit code 0
```

Compilation completed successfully in 4 s 149 ms (5 minutes ago)

By default expand with: Tab

- ✓ Kotlin
- ✓ Maven
- ✓ OGNL
- ✓ OGNL (Struts 2)
- ✓ other
  - ✓ geti (Inserts singleton method getInstance)
  - ✓ ifn (Inserts "if null" statement)
  - ✓ inn (Inserts "if not null" statement)
  - ✓ inst (Checks object type with instanceof and down-casts it)
  - ✓ lazy (Performs lazy initialization)
  - ✓ lst (Fetches last element of an array)
  - ✓ mn (Sets lesser value to a variable)
  - ✓ mx (Sets greater value to a variable)
  - ✓ psvm (main() method declaration)
  - ✓ toar (Stores elements of java.util.Collection into array)
- ✓ output
- ✓ plain
- ✓ React
- ✓ RESTful Web Services
- ✓ sbt
- ✓ scala

Abbreviation: psvm Description: main() method declaration

Template text:

```java
public static void main(String[] args){
  $END$
}
```

Options

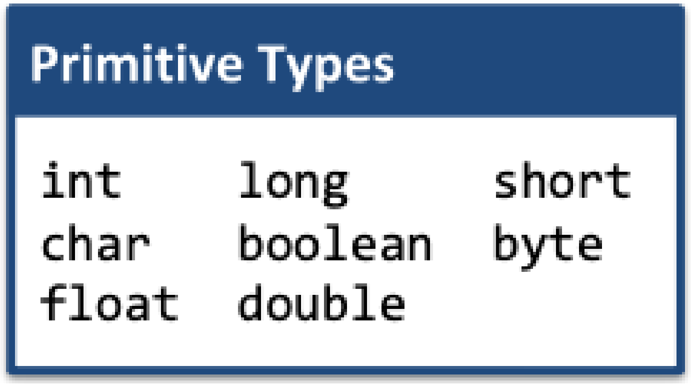
Expand with: Default (Tab)

- ✓ Reformat according to style
- ☐ Use static import if possible
- ✓ Shorten FQ names

Applicable in Java: declaration; Groovy: declaration. Change

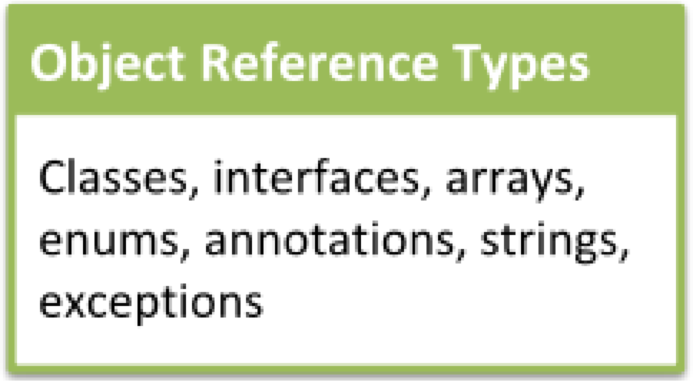
Java Type System

- Java has a *bipartite* (2-part) type system



Primitive Types

```
int     long     short
char    boolean  byte
float   double
```



Object Reference Types

Classes, interfaces, arrays, enums, annotations, strings, exceptions

- No identity except their value
- Immutable
- On stack, exist only when in use
- Can't achieve unity of expression
- Dirt cheap

- Have identity distinct from value
- Some mutable, some immutable
- On heap, garbage collected
- Unity of expression with generics
- More costly

Java Primitive Types

• int	32-bit signed integer
• long	64-bit signed integer
• byte	8-bit signed integer
• short	16-bit signed integer
• char	16-bit unsigned integer/character
• floa	32-bit IEEE 754 floating point number
• double	64-bit IEEE 754 floating point number
• boolean	Boolean value: true or false

“Practically Deficient” primitive types

- byte, short – typically use int instead!
  - byte is signed!!!
- float – typically use double instead!
  - Provides too little precision
  - Few compelling use cases, e.g., large arrays in resource-constrained environments

Objects

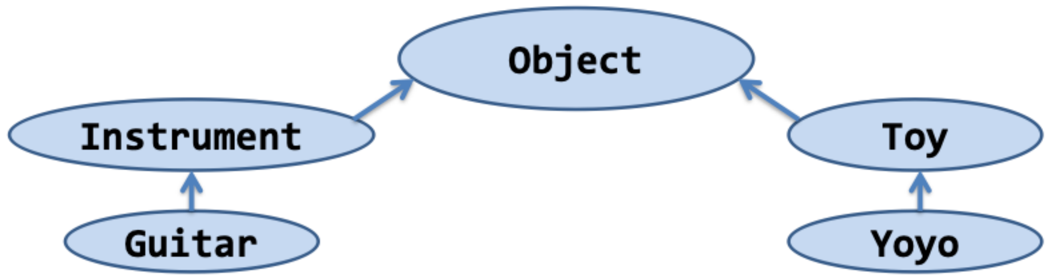
- All non-primitives are represented by objects.
- An **object** is a bundle of state and behavior
- State – the data contained in the object
  - In Java, these are called its instance **fields**
- Behavior – the actions supported by the object
  - In Java, these are called its **methods**
  - Method is just OO-speak for function
  - “Invoke a method” is OO-speak for “call a function”

Classes

- Every object has a class
  - A class defines methods and fields
  - Methods and fields collectively known as **members**
- Class defines both type and implementation
  - Type ≈ what object does (hence where it can be used)
  - Implementation ≈ how the object does things
- Loosely speaking, the methods of a class are its **Application Programming Interface (API)**
  - Defines how users interact with its instances

Java Class Hierarchy

- The root is Object (all non-primitives are objects)
- All classes except Object have one parent class
  - Specified with an **extends** clause
    class Guitar extends Instrument { ... }
  - If extends clause omitted, defaults to Object
- A class is an instance of all its superclasses

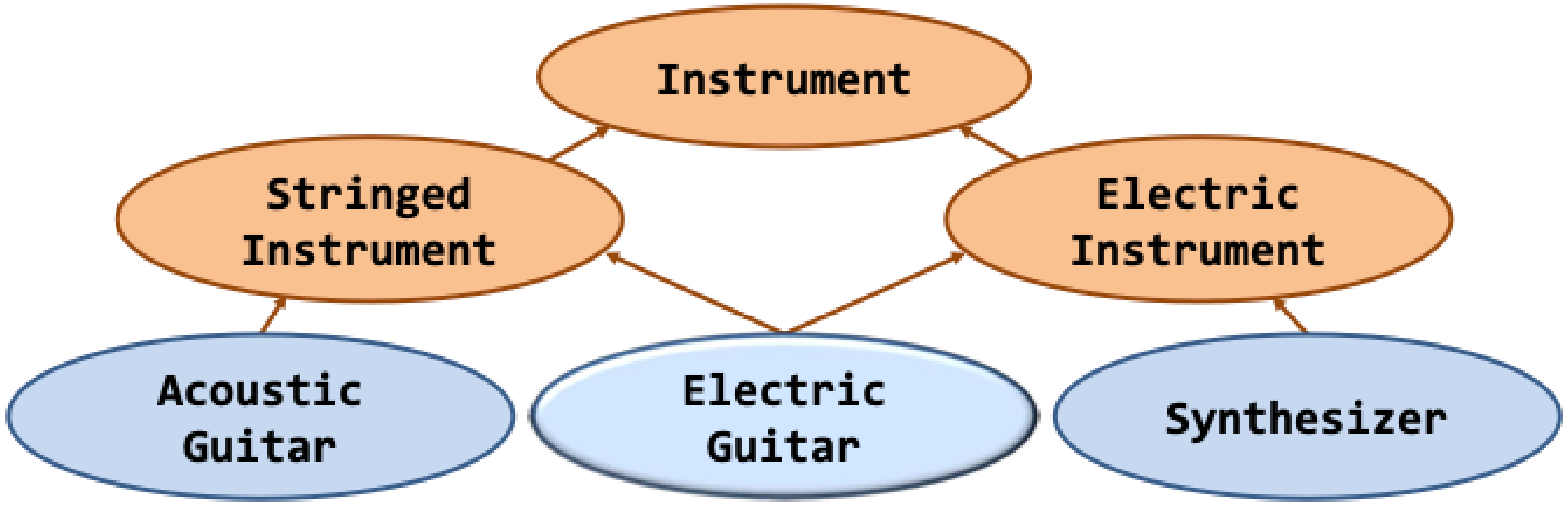


Implementation inheritance

- A class:
  - Inherits visible fields and methods from its superclasses
  - Can *override* methods to change their behavior
- Overriding method implementation must obey the contract(s) of its superclass(es)
  - Ensures subclass can be used anywhere superclass can
  - *Liskov Substitution Principle* (LSP)
  - We will talk more about this in a later class

Interface types

- Defines a type without an implementation
- Much more flexible than class types
  - An interface can extend one or more others
  - A class can implement multiple interfaces



Enum types

- Java has object-oriented enums
- In simple form, they look just like C enums

```
enum Planet { MERCURY, VENUS, EARTH, MARS, JUPITER, SATURN, URANUS, NEPTUNE }

Planet location = …;
if (location.equals(Planet.EARTH)) {
System.out.println("Honey, I'm home!"); }
```

- But they have **many** advantages!
  - Compile-time type safety
  - Multiple enum types can share value names
  - Can add or reorder without breaking existing uses
  - High-quality Object methods are provided
  - Screaming fast collections (EnumSet, EnumMap)
  - Can iterate over all constants of an enum

You can add data to enums

```
public enum Planet {
        MERCURY(3.302e+23, 2.439e6), VENUS (4.869e+24, 6.052e6),
        EARTH(5.975e+24, 6.378e6), MARS(6.419e+23, 3.393e6);

        private final double mass;    // In kg.
        private final double radius;  // In m.
        private static final double G = 6.67300e-11; // N m2/kg2

        Planet(double mass, double radius) {
                this.mass = mass;
                this.radius = radius;
        }
        public double mass() { return mass; }
        public double radius() { return radius; }
        public double surfaceGravity() { return G * mass / (radius * radius); }
}
```

You can add behavior too

```
public enum Planet {
        ... // As on previous slide

        public double surfaceWeight(double mass) {
                return mass *  surfaceGravity; // F = ma
        }
}
```

Example

```
public static void main(String[] args) {
        double earthWeight = Double.parseDouble(args[0]);
        double mass = earthWeight / EARTH.surfaceGravity();

        for (Planet p: Planet.values()) {
                System.out.printf("Your weight on %s is %f%n", p, p.surfaceWeight(mass));
        }
}
```

```
$ java WeightOnPlanet 180
Your weight on MERCURY is 68.023205
Your weight on VENUS is 162.909181
Your weight on EARTH is 180.000000
Your weight on MARS is 68.328719
```

You can even add value-specific behavior

```
public enum Operation {
       PLUS  ("+", (x, y) -> x + y),
       MINUS ("-", (x, y) -> x - y),
       TIMES ("*", (x, y) -> x * y),
       DIVIDE("/", (x, y) -> x / y);

       private final String symbol;
       private final DoubleBinaryOperator op;
       Operation(String symbol, DoubleBinaryOperator op) {
              this.symbol = symbol;
              this.op = op;
       }

       @Override
       public String toString() { return symbol; }

       public double apply(double x, double y) {
              return op.applyAsDouble(x, y);
       }
}
```

Lambda syntax…!
We'll learn the details later.

Example

```
public static void main(String[] args) {
       double x = Double.parseDouble(args[0]);
       double y = Double.parseDouble(args[1]);

       for (Operation op : Operation.values())
              System.out.printf("%f %s %f = %f%n", x, op, y, op.apply(x, y));
}
```

```
$ java TestOperation 4 2
4.000000 + 2.000000 = 6.000000
4.000000 - 2.000000 = 2.000000
4.000000 * 2.000000 = 8.000000
4.000000 / 2.000000 = 2.000000
```

Enums are your friend

- Use them whenever you have a type with a fixed number of values known at compile time

Enum (anti-) pattern

```
public static final int APPLE_FUJI         = 0;
public static final int APPLE_PIPPIN       = 1;
public static final int APPLE_GRANNY_SMITH = 2;

public static final int ORANGE_NAVEL  = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOOD  = 2;
```

- This technique, known as the *int enum pattern,* has many shortcomings.
- It provides nothing in the way of type safety and little in the way of expressive power.
- The compiler won't complain if you pass an apple to a method that expects an orange, compare apples to oranges with the == operator

```
public enum Apple  { FUJI, PIPPIN, GRANNY_SMITH }
public enum Orange { NAVEL, TEMPLE, BLOOD }
```

Boxed primitives

- Immutable containers for primitive types
- Boolean, Integer, Short, Long, Character, Float, Double
- Let you “use” primitives in contexts requiring objects
- Language does *autoboxing* and *auto-unboxing*
- **Don’t use boxed primitives unless you have to!**
- **Use Boxed primitives when**
    1. Using parameterized types (list Collection). Parameterized types do not permit primitives.
    2. Using value as a key or value in Collections, e.g. HashSet<Integer>
    3. Using reflective method invocation (another don't do). e.g. class.forName("java.lang.Integer");

```
public class Test {
    Integer i; // Initialized to null
    public static void main(String[] args) {
        if ( i == 45 )    // 1. auto-unbox i (convert Integer to int)
                          // 2. NullPointerException as i value is null.
            System.out.println( "i is 45." );
    }
}
```

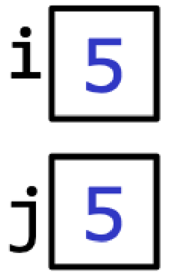
Comparing values

- x == y compares x and y “directly”:
- **primitive values:** returns true if x and y **have the same value**
- **objects refs:** returns true if x and y **refer to same object**

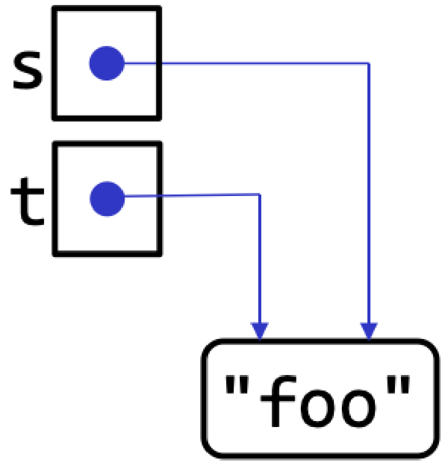
- x.equals(y) compares the ***values*** *of the objects referred to* by x and y

True or False?

```
int i = 5;
int j = 5;
System.out.println(i == j);
---------------------------
true
```

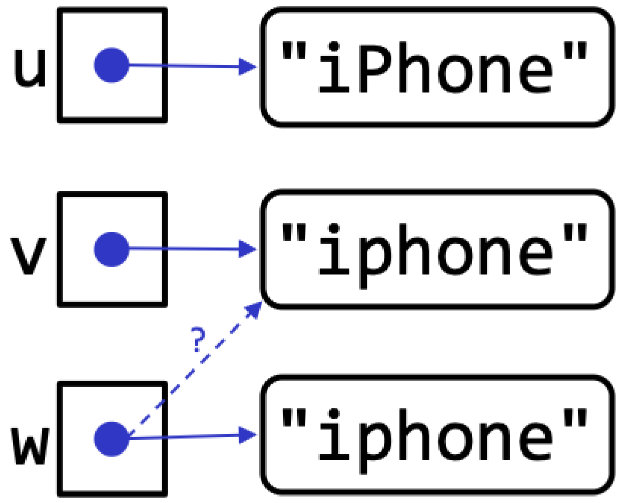


```
String s = "foo";
String t = s;
System.out.println(s == t);
---------------------------
true
```



```
String u = "iPhone";
String v = u.toLowerCase();
String w = "iphone";
System.out.println(v == w);
---------------------------
```

Undefined! (false in practice)



The moral of the example

- **Always use .equals to compare object refs!**
    - (Except for **enum**s, which are special)
- The == operator can fail silently and unpredictably when applied to object references
- Same goes for !=

Output

- Unformatted

```
System.out.println("Hello World");
System.out.println("Radius: " + r);
System.out.println(r * Math.cos(theta));
System.out.println();
System.out.print("*");
```

- Formatted – very similar to C

```
System.out.printf("%d * %d = %d%n", a, b, a * b); // Varargs
```

Command line input example

- *Echos all its command line arguments*

```
class Echo {
        public static void main(String[] args) {
                for (String arg : args) {
                        System.out.print(arg + " ");
                }
        }
}
```

```
$ java Echo Woke up this morning, had them weary blues
Woke up this morning, had them weary blues
```

Command line input with parsing

- *Prints the GCD of its two command line arguments*

```
class Gcd {
        public static void main(String[] args) {
                int i = Integer.parseInt(args[0]);
                int j = Integer.parseInt(args[1]);
                System.out.println(gcd(i, j));
        }

        static int gcd(int i, int j) {
                return i == 0 ? j : gcd(j % i, i);
        }
}
```

```
$ java Gcd 11322 35298
666
```

Scanner input

- *Counts the words on standard input*

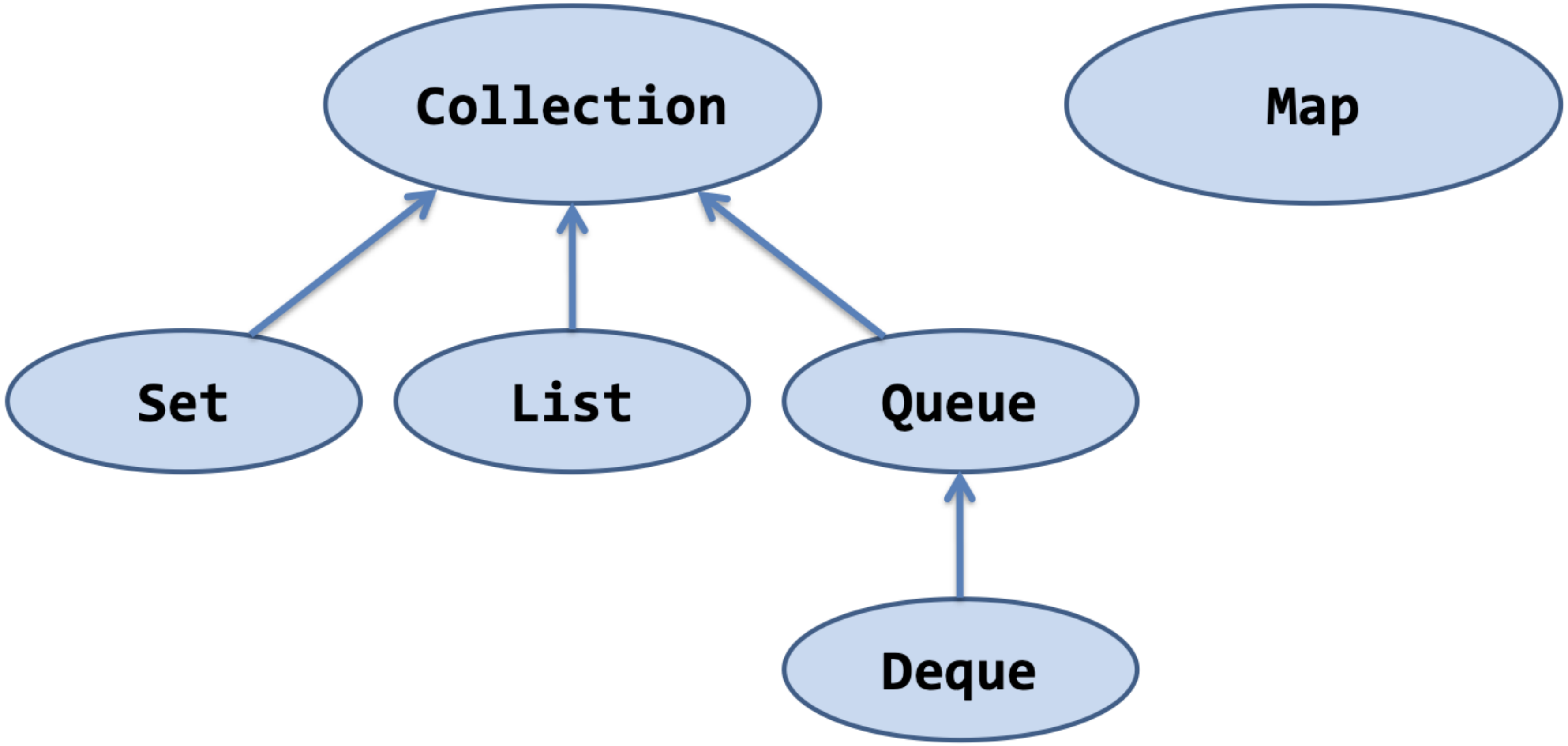
```
class Wc {
       public static void main(String[] args) {
              Scanner sc = new Scanner(System.in);
              long result = 0;

              while (sc.hasNext()) {
                     sc.next(); // Swallow token
                     result++;
              }
       System.out.println(result); }
}

$ java Wc < Wc.java
32
```

Java Collections

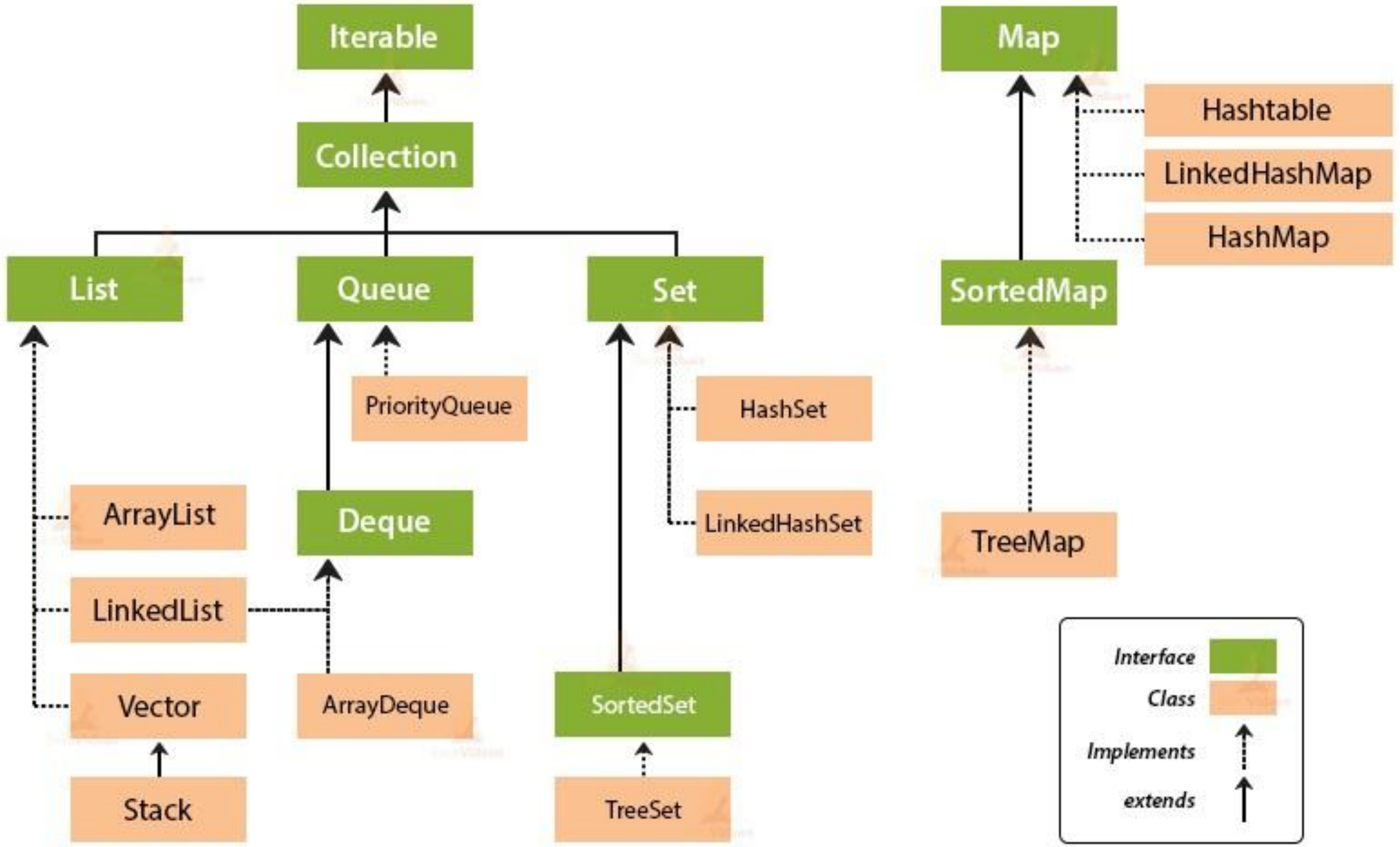
- A collection is an object that represents a group of objects
- Java Collections Framework
  - Interfaces for common abstract data structures
  - Classes that implement those data structures
  - Includes **algorithms** (e.g. searching, sorting)
    - Algorithms are *polymorphic*: can be used on many different implementations of collection interfaces.

Java Collections

- Primary collection interfaces



Collection Framework Hierarchy in Java



Collections usage example 1

- *Squeezes duplicate words out of command line*

```
public class Squeeze {
       public static void main(String[] args) {
              Set<String> s = new LinkedHashSet<>();
              for (String word : args)
                     s.add(word);
              System.out.println(s);
       }
}

$ java Squeeze I came I saw I conquered
[I, came, saw, conquered]
```

Collections usage example 2

- *Prints unique words in alphabetical order*

```
public class Lexicon {
	public static void main(String[] args) {
		Set<String> s = new TreeSet<>();
		for (String word : args)
			s.add(word);
		System.out.println(s);
	}
}
```

```
$ java Lexicon I came I saw I conquered
[I, came, conquered, saw]
```

Collections usage example 3

- *Prints the index of the first occurrence of each word*

```
class Index {
	public static void main(String[] args) {
		Map<String, Integer> index = new TreeMap<>();

		// Iterate backwards so first occurrence wins
		for (int i = args.length - 1; i >= 0; i--) {
			index.put(args[i], i);
		}
		System.out.println(index);
	}
}

$ java Index if it is to be it is up to me to do it
{be=4, do=11, if=0, is=2, it=1, me=9, to=3, up=7}
```

What about arrays?

- Arrays aren’t a part of the collections framework
- But there is an adapter: Arrays.asList
- Arrays and collections don’t mix well
- If you try to mix them and get compiler warnings, take them seriously
- Generally speaking, prefer collections to arrays
- But arrays of primitives (e.g., int[]) are preferable to lists of boxed primitives (e.g., List<Integer>)

Java Arrays

- Conceptually represented as an object
  - Provides .length, runtime bounds checking

```
String[] answers = new String[42];
if (answers.length == 42) {
    answers[42] = "no"; // ArrayIndexOutOfBoundsException
}
```

Methods common to all objects

- How do collections know how to test objects for equality?
- How do they know how to hash and print them?
- The relevant methods are all present on Object
  - **equals** - returns true if the two objects are “equal”
  - **hashCode** - returns an int that must be equal for equal objects, and is likely to differ on unequal objects
  - **toString** - returns a printable string representation

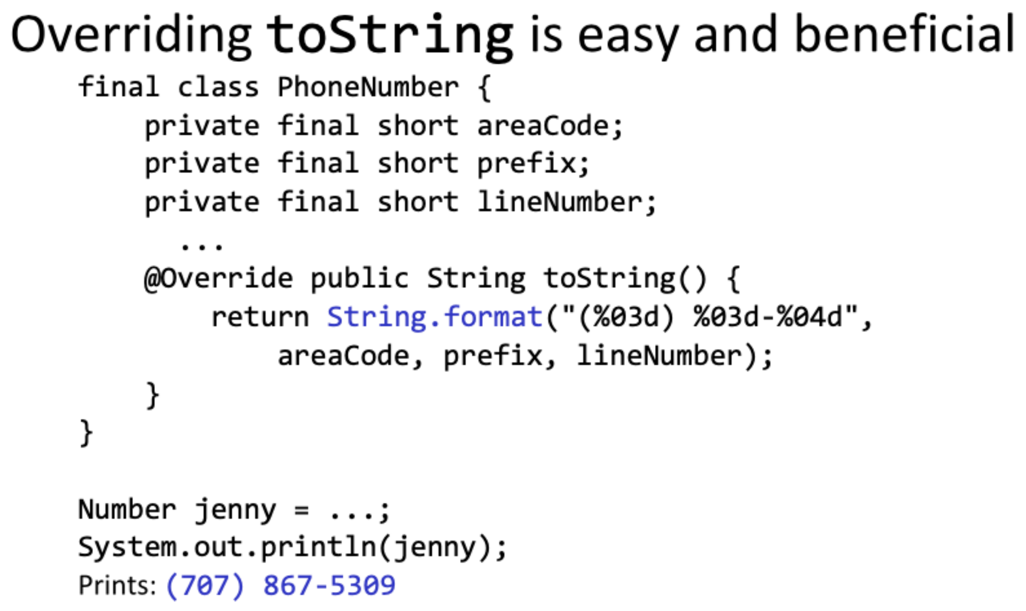
Object Implementations

- Provide identity semantics
  - equals(Object o) – returns true if o refers to this object
  - hashCode() – returns a near-random int that never changes over the object lifetime
  - toString() – returns a nasty looking string consisting of the type and hash code
    - java.lang.Object@659e0bfd

Overriding Object implementations

- No need to override equals and hashCode if you want identity semantics
  - When in doubt do not override
  - Identity semantics are often what you want
  - It is easy to get it wrong

- Nearly always override toString
  - println invokes it automatically
  - Why settle for ugly?



Overriding **`toString`** is easy and beneficial

```java
final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;
      ...
    @Override public String toString() {
        return String.format("(%03d) %03d-%04d",
            areaCode, prefix, lineNumber);
    }
}

Number jenny = ...;
System.out.println(jenny);
```

Prints: `(707) 867-5309`

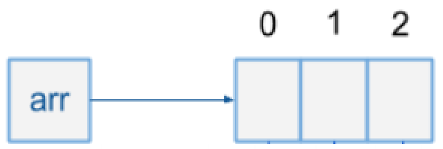
Java Annotations

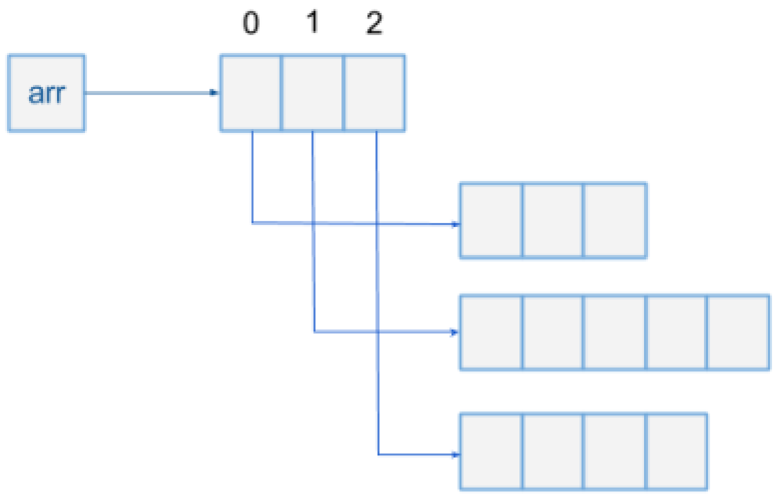
- Annotations mark code without any immediate functional effect
- @Override, @Deprecated, @SuppressWarnings

```
class Bicycle {
    ...
    @Override
    public String toString() {
        return ...;
    }
}
```

Misc.

```
"Hello, " + "World" → "Hello, World"
"Number " + 5 → "Number 5"


int[] arr = { 10, 100, 1_000 };


// irregular
int[][] arr = { { 1,2,3 },
                { 1,2,3,4,5 },
                { 1,2,3,4 } };


int[][] triangle = new int[10][];
for(int i=0; i<triangle.length; i++)
    triangle[i] = new int[i+1];
```





Summary

- Java is well suited to large programs; small ones may seem a bit verbose
- Bipartite type system – primitives & object refs
- Single implementation inheritance
- Multiple interface inheritance
- Easiest output – println, printf
- Easiest input – Command line args, Scanner
- Collections framework is powerful & easy to use