# Java

* **What is Java?**
* Programming Language: Syntax, Data Types, Control flow, Object Oriented
* This language equips us to work in all different places
* Runtime Environment: Configuration, Security, Threading, Input/output
* In general, RE is referred as Java SE (Java EE, Java ME, Java FX)
* Android-not tied to Java runtimes from Oracle
* To run and create Java applications,

**JRE vs. JDK**

* JRE: Required to run Java apps, End-users normally require only the JRE
* JDK: Provides tools required to create Java apps

Xyz.java -> Java Development Kit Tools -> Java App (platform independent)

Java Runtime Environment (provide what we need to execute on host environment)

Host Environment (Windows/ Mac/ Linux/ Android, Browser)

* How does JRE know where to start?
  + JRE loads the classes then executes special “main” method to get started
* **Summary**
  + Java Is a language and a runtime environment
    - Specific environment features may vary
    - Language remains pretty consistent
    - End-users require the JRE
    - Developers require the JDK
    - Many Integrated Development Environments(IED) are available for programming codes

# Variables, Data Types, and Math Operators

* **Variables**

: Named stored data

* **Data Types**
* Primitive Data Types

: Built into the language, Foundation of all other types

* Four categories

1. Integer Types: byte, short, int, long(L)
2. Floating Point Types: Stores values containing a fractional portion. Supports positive, negative, and zero values: float-32bits (ex. 0.0f), double-64bits (ex. 0.0 or 0.0d)
3. Character and Boolean Types
   * The char type stores a single Unicode Character
     + Literal values placed between single quotes
     + For Unicode code points, use \u followed by 4-digit hex value

* Primitive types are stored *By-value*

: When the value is initiated and assigned, the value is created or stored inside that memory. (each primitive type value has its own separate copy!!)

* **Math Operators**

: Add(+), Subtract(-), Multiply(\*), Divide(/: with integer, the floating point will be dropped), Modulus(%)

* Prefix / Postfix Operators
  + ++ increments value by 1
  + -- decrements value by 1
  + As prefix applies operation *before* returning value,

int myVal = 5;

System.out.println(++myVal); // 6

System.out.println(myVal); // 6

* + As postfix applies operation *after* returning value,

int myVal = 5;

System.out.println(++myVal); // 5

System.out.println(myVal); // 6

* Compound Assignment Operators

: +=, -=, \*=, /=, %=

* Operator Precedence

: Postfix > Prefix > Multiplicative > Additive

: Left-to-right

: Parenthesis

* **Type Conversion**
* Implicit Type Conversion = Widening

: performed automatically by the complier

: Safely converted

* Rules:

Mixed integer sizes = Uses largest integer in equation

Mixed floating-point sizes = Uses double

Mixted integer and floating point = Use largest floating point in equation

(ex)

int iVal = 50; // assign a 32 bit integer

long lVal = iVal; // the complier has convert a 32 bit integer into a 64 bit integer as its stored int lVal

* Explicit Type Conversion = Widening and Narrowing

: performed explicitly in code with cast operator

: Floating point to integer drops fraction

: Integer to floating point can lose precision

: Always use explicit casting when narrowing down

(ex) Narrowing (use caution with narrowing conversion!)

long lVal = 50;

int iVal = (int) lVal; // 62 bit into 32 bit integer (cast down)

# Conditional Logic, Looping, Arrays

* **Relational Operators**

Operator: Greater than (>), Greater than or equal to (>=), Less than (<), less than or equal to (<=), equal (==), not equal (!=)

* **Conditional assignment**
* Assign a value to a variable based on the result of a condition

*Result = condition? true-value : false-value;*

* If-else statement/ If-else if-else statement
* **Block Statements**: groups statements into a compound statement

{

statement-1;

statement-2;

…

}

* Block Statements and Variables
* **Logical Operators**

: And(&), Or(|), Exclusive or(^), Negation(!)

* **Conditional Logical Operators**

: Conditional And(&&), Conditional Or(||)

: Resolve following conceptually similar rules as non-conditional operator

: Only executed the right-side if needed to determine the result

* && only executes right-side if left-side is true
* || only executes right-side if left-side is false
* Looping
  + While loop
  + do-While loop
  + for loop
* Arrays

: Provides an ordered collection of elements

* Each element accessed via an index
* Indexes range starting from 0 to number-of-elements minus 1

(ex)

float[] theVals = new float[3];

theVals[0] = 10.0f;

theVals[1] = 20.0f;

theVals[2] = 15.0f;

// float[] theVals = { 10.0f, 20.0f, 15.0f };

float sum = 0.0f;

for(int I = 0; I < theVals.length; i++)

sum += theVals[i];

* For-each loop

: Executes a statement once for each member in an array

* Handles getting collection length
* Handles accessing each value
* Syntax:

for(loop-variable-declaration : *array*)

statement;

(ex) for(float currentVal : theVals)

sum += currentVal;

# Classes

* **Classes in Java**
* Java is an object-oriented language
* Objects encapsulates data, operations, and usage semantics
  + Allows storage and manipulation details to be hidden
  + Separates “what” is to be done from “how” it is done
* Classes provide a structure for describing and creating objects
* Classes
  + Template for creating an object
    - Declared with the class keyword followed by the class name
    - Java source file name normally has same name as the class
    - Body of the class is contained within { }
* A class is made up of both state and executable code
  + Fields: store object state
  + Methods: executable code that manipulates state and perform operations
  + Constructors

(ex)

// in Flight.java

class Flight{

int passengers;

int seats;

Flight() {

passengers = 0;

seats = 150;

}

void addPassengers() {

passengers += 1;

}

}

* **Using Classes**
* Use the new keyword to create a class instance (a.k.a an object)
  + Allocate the memory described by the class
  + Returns a reference to the allocated memory

(ex)

Flight nycToSf = new Flight(); // the variable nycToSf is a reference to the object

* Classes are Reference Types
  + As reference types, assignments cause us to point to the same object instances as opposed to the complete copies.
* **Encapsulation and Access Modifier**
* The internal representation of an object is generally hidden
* This is known as *encapsulation*.
* Java uses *access modifiers* to achieve encapsulation.

|  |  |  |  |
| --- | --- | --- | --- |
| Modifier | Visibility | Usable on Classes | Usable on Members |
| No access modifier | Only within its own package | Y | Y |
| public | Everywhere | Y | Y |
| private | Only within its own class | \*N  As private applies to top-level classes; private is available to nested-classes | Y |

* **Method Basics**
* Executable code that manipulates state and performs operations
  + Name

: Same rules and conventions as variables

: Should be a verb or action

* Return Type  
  : Use void when no value returned
* Typed parameter list

: Can be empty

* Body contained with brackets
* Special References: this and null
  + *this* is an implicit reference to the current object
    - Useful for reduce ambiguity
    - Allows an object to pass itself as a parameter

(ex) public class Flight {

private int passengers;

private int seats;

// constructor and other methods

public Boolean hasRoom(Flight f2){

int total = *this.*passengers + f2.passengers;

return total <= seats;

}

}

* *null* is a reference literal
  + Represents an uncreated object
  + Can be assigned to any reference variable
* Field Encapsulation, Accessor and Mutator
* Field Encapsulation

: Helps to hide implantation details

: Use methods to control access to the fields

* Use the accessor/mutator pattern to control field access
  + Accessor retrieves field value = getter
  + Method name: get*FieldName*

(ex) public int getSeats() { return seats; }

* Mutator modifies filed value = setter
* Method name: set*FieldName*

(ex) public void setSeats(int seats) { this.seats = seats; }

(ex) Flight slcToNyc = new Flight();

slcToNyc.setSetas(50);

System.out.println(slcToNyc.getSeats()); // 50

# Class Initializers and Constructors

(Summary)

* Objects should be created in some useful state
* Field initializers provide an initial value as part of the declaration

(ex) int, double, long, byte sets to 0, boolean to false

* Every class has at least one constructor
* If no explicit constructor, Java provides one with no arguments
* You can provide multiple constructors with different argument lists
* One constructor can call another
* Call must be first line

(ex)

* Initialization blocks(set of brackets) share code across constructors
* Order:

Field initialization -> Initialization block -> Constructors (override)

# Parameters

* **Parameter Immutability**
  + Parameters are passed by making a copy of the value

(Known as ‘passing by value’)

* Primitive Types; Classes(references)
* Changes made to passed value are not visible outside of method
* Changes made to members of passed class instances ae visible outside of method

(ex)

Flight val1 = new Flight(10);

Flight val2 = new Flight(20);

swapNumbers(val1, val2);

Void swapNumbers(Flight I, Flight j) {

int k = i.getFlightNumber();

i.setFlightNumber(j.getFlightNumber());

j.setFlightNumber(k);

}

* **Overloading**

= A class may have multiple versions of its constructor or methods

* Each constructor and method must have a unique signature
* Signature is made up of 3 parts
  + 1. Number of parameters
  + 2. Type of each parameter
  + 3. Name
* **Variable Number of Parameters**
  + A method can be declared to accept a varying number of parameter values
    - Place an ellipse(…) after parameter type
    - Can only be the last parameter
    - Method receives values as an array

# Class Inheritance

* **Inheritance & Member hiding and Overriding**
  + A class can be declared to inherit( a.k.a derive) from another class
    - Use the “extends” keyword
  + Derived class has characteristics of base class
    - Can add specialization
    - Can be assigned to base class typed reference
* Fields in derive class hide base class fields with same name
* Methods override base class methods with same signature
  + Derived class can override base class methods

(Optionally use @Override annotation)

* **Object Class**
  + The object class is the root of the Java class hierarchy
    - Every class has the characteristics of the Object class
* All classes derive from Object class either directly or indirectly
  + Useful for declaring variables, fields and parameters that can reference any class or array instance
  + Defines a number of methods that are inherited by all objects
  + **Inheriting from Object class**
* Object Class Methods
  + - clone: create a new object instance that duplicates the current instance
    - hashCode: get a hash code for the current instance
    - getClass: return type information for the current instance
    - finalize: handle special resource cleanup scenarios
    - toString: returning string of characters representing the current instance
    - equals: compare another object to the current instance for equality

(by default, object references are only equal when referencing the same instance)

* **Using super keyword**
  + Accesses current object as if instance of base class
* **Using Final and Abstract (Control over inheritance) keyworkd**
* Class that is marked as abstract cannot be initiated with new keyword
* **final**
  + Class marked as final = not allowed to be inherited
  + Method is marked final, it can’t be overridden.
* **abstract**
* abstract method is itself abstract, meaning that that class must be overridden.
  + It doesn’t actually have a body(in the base class)
* Any class that contains a method marked abstract
  + must be marked as abstract
  + must be inherited from other classes
* Constructors are not inherited

# More about Data Types

* **String class**

- The String class stores a sequence of Unicode characters(stored using UTF-16 encoding)

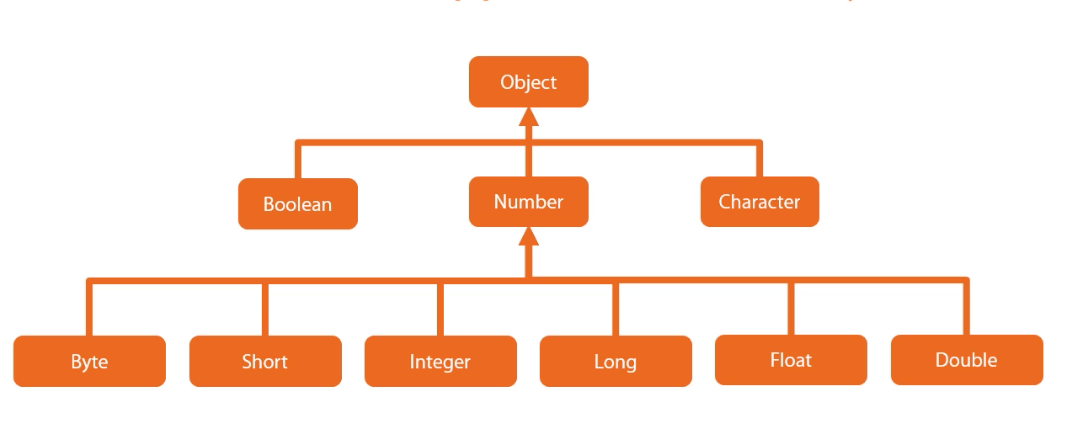
* + - Literals are enclosed in double quotes(“ ”)
    - Values can be concatenated using + and +=
    - String objects are immutable
* Select String Class Methods
  + - length : get length of String operated on Length
    - valueOf : operated on String for non-String

(ex)

int iVal = 100;

String sVal = String.valueOf(iVal); // “100”

* + - concat, replace, toLowerCase, toUpperCase, trim, split
    - fomat
    - charAt, substring
    - contains, endsWith, startsWith, indexOf, lastIndxOf
    - compareTo, compareToIgnoreCase, isEmpty, equals, equalsIgnoreCase
* **Equality**
* ==: compares values based on whether they both point to the exact same instance
* equals method: compare character by character of each string in the String objects
* intern method: returns back a canonicalized value = when you all the intern method on a string value, it will always return back the same exact string object for a given string value
* **Converting Non-string types to String types**
* String.valueOf provides overrides to handle most types
* Conversions often happen implicitly
* Class conversions controlled by the class toString() method
* **StringBuilder**
* StringBuilder provides mutable string buffer
* **Primitive Wrapper Class**
* can be treated as object
* fields and methods that are particular to that time
* Instances are immutable



* **Wrapper class & Primitive Conversation**
* Common conversions handled automatically
* Wrapper classes provide methods for explicit conversations
  + - Primitive to wrapper(boxing)

: valueOf

* Wrapper to primitive(unboxing)

: xxxValue

* String to primitive

: parseXxx

* String to wrapper

: valueOf

(ex)

//automatically handled

Integer d = 100;

int b = a;

Integer c = b;

// Primitive to wrapper and wrapper to primitive

Integer d = Integer.valueOf(100);

int e = d.intValue();

// String to primitive type and String to wrapper

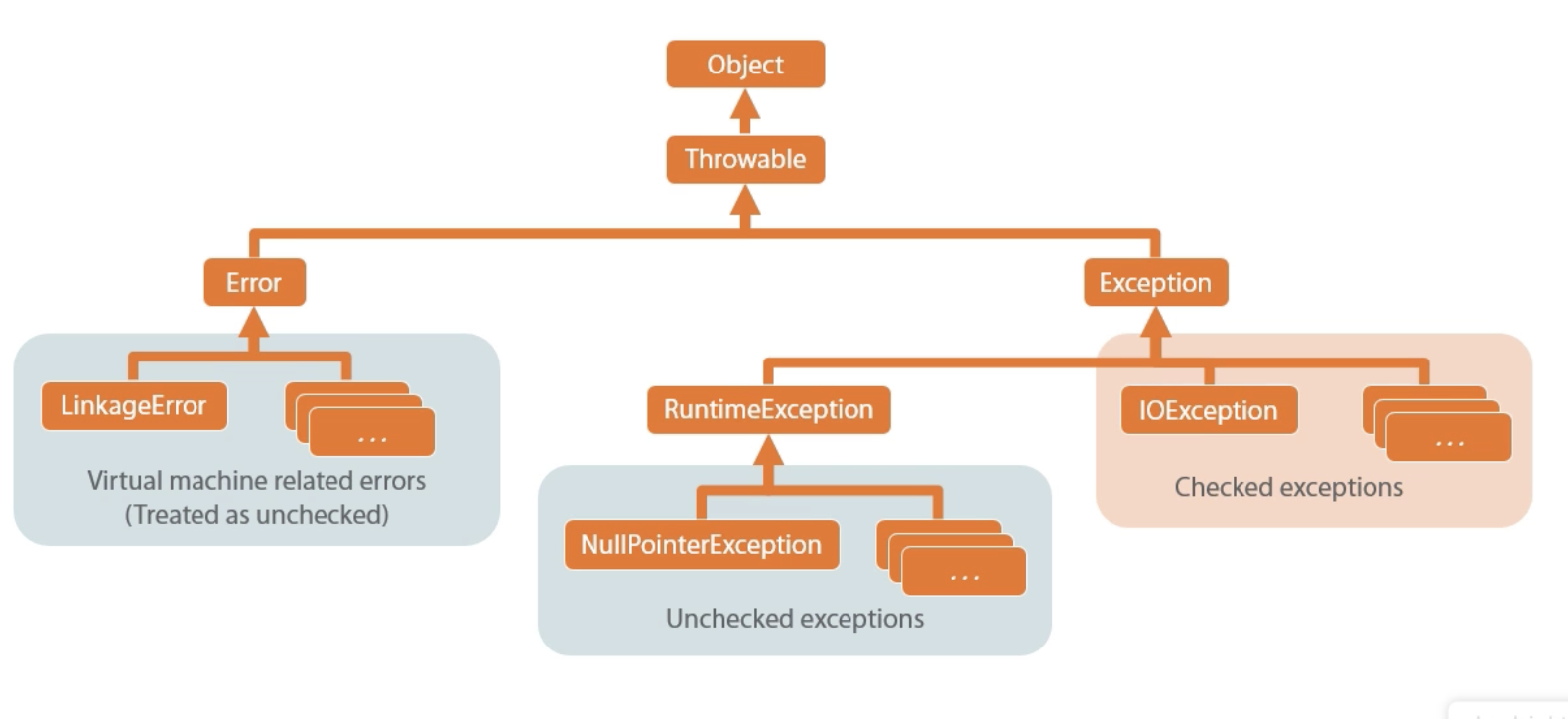
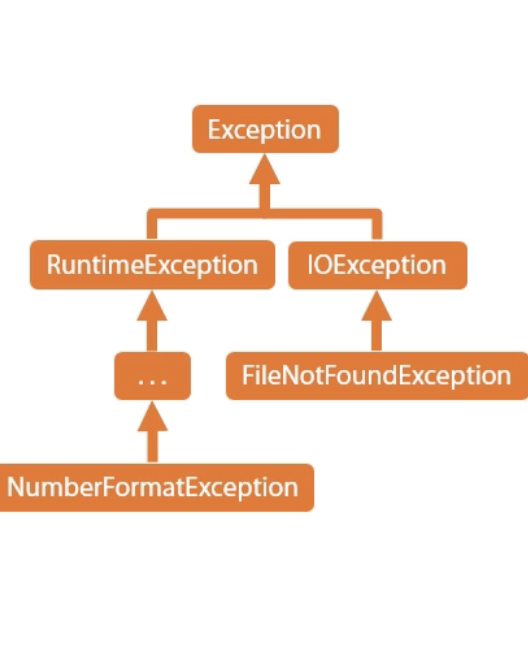
String s = “100.24”;

double s1 = Double.pareseDouble(s);

Double = Double.valueOf(s);

* **Wrapper class equality**
  + Use equals method would be safe way
  + (object1).equals(object2)

# Exception handling



* Checked exception should be handled using try-catch phrase

# Creating an Abstract Relationship with Interfaces

* **What is an Interface?**

: An Interface defines a contract

* Provides no implementation

: Classes implement interfaces

* Expresses that the class conforms to the contract

: Interfaces don’t limit other aspects of the class’s implementation

* Implementing interface

(ex) java.lang.Comparable ( comparable interface)

* Used for determining relative order(sort)
* One method: compareTo
* Receives item to compareTo
* Return indicates current instance relative sequence
  + - * + Negative value: before
        + Positive value: after
        + Zero value: equal

(ex) public class Passenger implements Comparable{

….

public int compareTo(Object o){

Passenger p = (Passenger) o;

If(memberLevel > p.memberLevel)

return -1;

// continue

}

}

: Some interfaces require additional type information

* Implementing a Generic Interface

public interface Comparable<T> {

int compareTo(T o);

}

(ex)public class Passenger implements Comparable<Passenger> {

public int compareTo(Passneger p){

If(memberLevel > p.memberLevel)

return -1;

// continue

}

}

: Classes are free to implementing multiple interfaces = Implementing Multiple Interfaces

(ex) java.lang.iterable

* Methods: iterator, hasNext, next

public interface Iterable<T>{

Iterator<T> iterator();

}

public interface Iterable<T>{

boolean hasNext();

T next();

}

// Iterator over Person class to interact with their information

(ex)

public class Flight implements Comparable<Flight>, Iterable<Person>{

….

public Iterator<Person> iterator(){

return FlightIterator(crew, roaster);

}

}

public class FlightIterator implement Iterator<Person> {

private CrewMember[] crew;

privae Passenger[] roaster;

private int index;

public FlightIterator(CrewMember[] crew, Passenger[] roaster) {

this.crew = crew;

this.roaster = roaster;

}

public hasNext() {

return index < (crew.length + roaster.length);

}

public Person next() {

// first, work through the crew array and then passenger array

Person p = (inext < crew.length) ? crew[index] : roaster[index – crew.length];

index++;

return p;

}

}

🡪 for each statement is useful