1 Java Essentials

1.1 Primitives vs. Reference Types

A **primitive type** is a simple, indecomposable value. Primitive types include `int`, `double`, `char`, `float`, and `boolean`.

```
byte   short   int   long   float   double
```

(You can go up the list without **typecasting**.)

A **reference type** represents a memory address rather than the actual item stored at that address. All instances of **classes** are reference types. For example, `String` is a reference type.

Declaring a variable is the process of giving it a specific type. **Initializing** a variable is the process of giving it a specific value. If you do not initialize a variable, it will revert to its default value (e.g., for `int` types, this is `0` and for objects, this is `null`).

Typecasting (explicitly) changes the data type of a variable.

```java
double distance = 9.14;
int points = (int)distance; // points = 9 (truncation)
```

Note: For binary operations, i..e, `*` or `+`, the result is given the data type of the most "complex" operand (i.e., `double * int = double`).

The **increment** operator can be used in one of two ways.

```java
int i = 1;
// pre-increment example
System.out.println("--i is " + (--i)); // prints 2
// post-increment example
System.out.println("i-- is" + (i--)); // prints 2
```

1.2 Variable Scope

Variable **scope** applies to anywhere there are braces in a Java program. For example:

```java
for (int i = 0; i < 3; i++) {
    System.out.println(i); // this is fine
}
System.out.println(i) // this is NOT fine
```

However, if we declare `i` before the loop:

```java
int i;
for (i = 0; i < 3; i++) {
    System.out.println(i); // this is fine
}
System.out.println(i) // this is also fine
```

Note that the concept of scope also applies to **local variable** in method definitions: variables declared inside of methods are only available inside that method.

1. Variables declared within for-loops or while loops are only available within the loop.

2. Parameters of methods/constructors are only available within that method/constructor.
3. Variables declared within a class definition (as data members) are only available within the class (if declared private).

Note that if, within a method, there is a local variable with the same name as a data member of the class, the local variable takes precedence over the data member. (This is why we use `this.name` when referring to a *class* data member.)

1.3 Boolean Expressions, Conditionals

Example of a compound `if-else` statement in Java:

```
if ((something)&&(another)) {
    statement1;
    statement2; }
else if ((one)||(two)) {
    statement 3; }
else {
    statement 4; }
```

The `switch` statement in Java:

```
switch (stoplightColor) { // where stoplightColor is a string
    case "green":
        System.out.println("go");
        break;
    case "yellow":
        System.out.println("yield");
        break;
    case "red";
        System.out.println("stop");
    default:
        System.out.println("unknown");
        break;
}
```

1.4 Loops

There are four types of loops in java. The first type is the standard indefinite **while loop**.

```
while (condition) {
    do this; }
```

The second is a variant on this loop, called the **do-while loop**. The difference is that a do-while loop checks the continuation condition after running the contents of the loop body (which ensures that the loop will always run at least once).

```
do { some statements; }
while (condition);
```

The third is the **for loop**. Note that you can initialize multiple variables and have multiple update operations within the loop control statement (separated by commas).

```
for (int i = 0; i <= ubound ; i++) {
    some statements; }
```

The fourth is the **for each loop**. Note that this type of loop can only be used on objects which implement the `Iterable` interface (more on this later).

```
for (type obj : iterable) {
    some statements; }
```

Using a `break` statement stops the loop entirely while using a `continue` skips to the next iteration of the loop.

1.5 Strings and String Operations

Method Name	**Description**
`str.charAt(3)`	returns `char` object at index 3
`str.length()`	returns length of string
`str.substring(2,4)`	returns the equivalent of `str[2,4]`
`str.indexOf('x')`	returns index of first instance of x
`str.split('\s')`	splits the string on whitespace and returns an array of strings
`str.toCharArray()`	returns an array of characters

Figure 2. Useful String methods

Use `Integer.toString(num)` to convert an integer to a string. Use `Integer.valueOf("string")` to convert a string to an integer.

1.6 Wrapper Classes

A **wrapper** class is used to convert values of primitive types to objects of their corresponding class types. For example, the `int` primitive has the `Integer` wrapper class.

Two useful wrapper class constants are the `Integer.MAX_VALUE` and `Integer.MIN_VALUE` constants.

```
double num = 5.0;
Double objectD = new Double(num); // creates a Double object, "boxing"
// we are calling the Double constructor on a double primitive
double r = objectD.doubleValue(); // since we have an obj, we can call methods
```

1.7 Standard I/O

To get user input, use a `Scanner` object.

```
import java.util.Scanner;
Scanner input = new Scanner(System.in);
type varname = input.nextType();
```

Method Name	**Description**
`hasNext()`	returns true if more data is present
`nextInt()`	returns the next entered integer value
`nextDouble()`	returns the next entered double value
`next()`	returns the next string
`nextLine()`	returns the entire entered line as a string

Figure 1. Useful Scanner methods

1.8 Arrays and Array Operations

Arrays are objects. Note that arrays must be of homogeneous type and have fixed length.

```
double[] temps = new double[7]; // sets up an array with seven "slots"
temps[0] = 36.0;
int[] values = {3,7,11,15}; // declares and initializes list in one step
```

Remember that for two-dimensional arrays, the first index is the number of “rows” and the second index is the number of “columns”.

```
int[][] table = new int[10][6];
for (int row = 0; row < table.length; row++) { // to iterate over multi-dim array
    for (int column = 0; column < table[row].length; column++) {
    }
}
// note that table.length == 10!
```

Note that the `==` operator will test whether two arrays are references to the same object. To test whether two arrays contain the same values, you need to test each value individually.

2 Object-Oriented Design

Three principles of OOP: encapsulation, inheritance, and polymorphism.

2.1 Type: Data and Methods

Classes have **data members** and **methods**. Data members determine the characteristics of an instance of the class while methods determine the actions of that instance (i.e., what an object can “do”).

2.2 Class and Method Definitions

Classes have **accessor** methods (valued methods) and **mutator** methods (void methods). The essential parts of a method include

`access-modifier`	`use-modifier`	`return-type`	`method-name`
public, private, protected	abstract, final, static	void or type	

Every class should also have a **Constructor** method. Note that the constructor is a public method and has no return type. If you do not define a Constructor, Java will use the default Constructor.

Note that methods can call other methods within the class definition.

```
public class Name {
    private String first;
    private String last;

    ... // Constructor, other methods, etc

    public void setFirst(String first) {
        this.first = first;
    }

    public void setLast(String last) {
        this.last = last;
    }

    public void setName(String first, String last) {
        setFirst(first);
        setLast(last);
    }
}
```

A `static` data member or method is one that can be accessed without instantiating an object of the class (i.e., you can call it on the class itself). The `final` modifier indicates that the method or data member cannot be changed by any object of the class or of any derived class.

The `protected` modifier indicates that a method/data member can be accessed: (1) by its own class definition, (2) by any derived classes, and (3) any class in the same package.

Note: A nonstatic data member **cannot** be used within a `static` method. However, `static` data members can be used within nonstatic methods.

2.3 Instantiating/Using Objects

Objects are **instantiated** by calling their constructor. Note that a class can have multiple different constructors (with different argument numbers and types).

Remember that objects are **reference** types. This means that the equality operator `==` tests to see if two variables are references to the *same* object. The only case in which you can (and should) use the equality operator on objects is when you are checking for `null` types.

2.4 Composition: Generics and Adapters

Composition occurs when a class has a data member that is an instance of another class. For example, suppose we had a `Name` class which stored first and last names as `String` types.

- This is a *has a* relationship (a `Name` has a first name that is of type `String`).
- The class must go through the public interface of the class corresponding to the data member.

A **generic** type is a class with data fields that can be of any *reference* type (i.e., think of it like a "container"). Note that generics will *not* take primitives.

```
public class Point<T> { // T must be an object
    private T x;
    private T y;

    public Point(T inX, T inY) {
        x = inX;
        y = inY;
    }

    public static void main(String[] args) {
        Point<Integer> start = new Point<Integer>(5,7);
        Point<Double> end = new Point<Double>(0.5, 0.7);
        start.setPoint(0.5, 0.5); // this won't work! "start" is expecting integers
    }
}
```

An **adapter** class uses composition to define a new class that has the original class as a data member and redefines ("adapts") certain methods.

```
// derived from the Name class
public class NickName {
    private Name nick;

    public NickName() {
        nick = new Name();
    }

    public void setNickName(String name) {
        nick.setFirst(name); // uses methods of the Name class!
    }
}
```

2.5 Inheritance

Inheritance is a *is a* relationship.

- Derived classes can use or override all methods of the base class.
- Derived class *cannot* access private data members and methods of the base class.

Note that you should *always* call the superclass constructor using `super()` in the derived class method. Otherwise, Java will call it for you!

```
// suppose we have a Student class which inherits from a Person class
// also suppose that setName() is defined in the Person class
public void setStudent(Name studName, int gradYear, String degreeSought) {
    setName(studName); // must use public interface of super class
    year = gradYear;
    degree = degreeSought;
}
```

All classes are derived from the `Object` class. Generally, the `toString()` and `equals()` methods of this class need to be overridden in your derived class. In fact, if `equals()` is not re-defined, it will do the same thing as `==` (compare memory addresses).

```
// tests if two Name objects are equal
public boolean equals(Object other) {
    boolean result = false;
    if (other instanceof Name) {
```

```
    // "instanceof" tests if the other object is a Name
        Name otherName = (Name)other;
        // other is still an object; need to typecast to Name
        result = first.equals(otherName.first) && last.equals(otherName.last);
    }
    return result;
}
```

2.5.1 Overriding vs. Overloading

Overriding occurs when a derived class re-defines a method present in the super class with the exact same method signature (same name, parameters, etc).

- By definition, when an overridden method is called on an instance of the derived class, the method body used will be the overridden method.
- If the method is called on an instance of the super class, the original method body will be used.
- You can still access the original method from within the derived class by calling `super()`.
- Overriding is useful when you want a derived class to implement a different version of the method from what is implemented by the parent class.

```
// in Student class
public String toString() {
    return id + " " + fullName.toString();
}

// in CollegeStudent class
public String toString() {
    return super.toString() + ", " + degree + ", " + year;
}
```

Overloading occurs when a new method is defined (either in a derived class or in the original super class) with the same method name but different parameters.

- Both the overloaded method and the original method are available to the object; Java differentiates by looking at the parameters.
- Overloading is useful when you want a single method to do different things based upon the parameters passed (for example, multiple constructors that take different arguments).

```
// in Student class

public void SetStudent(Name studName, String studID) {
}

// in CollegeStudent class

public void setStudent(Name studName, String studID, int gradYear, String degreeSought) {
}
```

2.5.2 Abstract Classes and Interfaces

An **abstract class** is a base class for which you don't intend to have objects of that type (i.e., you *cannot* instantiate objects of an abstract class).

- An abstract class *can* define methods that it wants all derived classes to either share or override.
- An abstract class can also declare methods without a method body. Doing so forces all derived classes to implement that method. For example,

  ```
  public abstract void doSomething()
  ```

- Abstract methods cannot be final, static, or private.
- Abstract classes can have a mixture of abstract and "regular" methods but any class with even one abstract method must be declared abstract.

An **interface** is a program component that contains public constants, method signatures, and comments to describe them. Think of an interface like a "contract" between the creator of the interface and any class that implements the interface.

- A class that implements an interface must implement *every* method declared in the interface.
- By definition, an interface does not have a constructor or any static methods.
- Constants declared in an interface should be final, static, and public.
- An interface does *not* declare or initialize any data fields.

```
public interface Comparable<T> {
    public int compareTo(T other); // result is a signed integer
}

public class Circle implements Comparable<Circle> {
    ...
    public int compareTo(Circle other) {
        // method comparing radii of circles
    }
}
```

If you have a class that both inherits from a super class and implements an interface, use:

```
public class myClass extends anotherClass implements myInterface
```

Note that methods can take interfaces as parameters. This means that the method can take any instance of a class implementing that interface as an argument. Furthermore, any variable of an interface type can be assigned to any object of a class implementing that interface.

Example of an interface versus an example of an abstract class:

```
public interface Circular {
    public void setRadius(double radius);
    public double getRadius();
    // note: no data fields!
}

public abstract class CircularBase {
    private double radius;
```

```
    public void setRadius(double radius) {
        this.radius = radius;
    }
    public double getRadius() {
        return radius;
    }
    /* note: because radius is private and derived classes inherit from CircularBase
    just like normal inheritance, we need getter/setter methods */
}
```

2.6 Polymorphism

Polymorphism is the idea of that an object can have many types. For example, if we have an UGStudent class which is derived from a CollegeStudent class, which is in turn derived from a Student class, an instance of the UGStudent class is also a CollegeStudent *and* a Student.

Rule of thumb: Is <constructed object> a <declared variable>?

```
Student amy = new CollegeStudent(); // this is fine
CollegeStudent brad = new Student(); // this is NOT fine
```

A variable's **static type** is the one that appears in its declaration. It's **dynamic type** is the type of object that the variable references at the current point in execution (can change and may be different from static type.

```
UGStudent ug = new UGStudent(...);
Student s = ug;
s.displayAt(2); // calls the UGStudent method because s "remembers" that it is UGStudent
```

Java calls the method corresponding to the object type of the constructor that created s which was the constructor that created ug.

```
UGStudent ug = new UGStudent(...);
Student s = (Student)ug;
s.displayAt(2); // STILL calls the UGStudent method
```

Note that the *variable* type determines which method names can be accessed. For example, if we had an UGStudent object assigned to a Student variable, we can only call methods from the Student class. However, if a method from the Student class is overridden in the UGStudent class, Java will call the method in the UGStudent class. The tl;dr is that the **variable type determines the methods that can be accessed while the object type determines the method action that is performed**.

3 More Advanced Topics

3.1 Exceptions

Java has three types of exceptions.

- **Checked exceptions** are serious disruptions to program execution. Examples include FileNotFoundException, NoSuchMethodException, and ClassNotFoundException. Checked exceptions *must* be handled.
- **Runtime exceptions** are the result of logical errors occuring during execution. Examples include NoSuchElementException, NullPointerException, ArrayIndexOutOfBoundsException, and IllegalStateException. Runtime exceptions *do not* need to be handled.

- An **error** generally means something abnormal happened, for example, the system ran out of memory. (For example, a `OutOfMemoryError` or `StackOverflowError`.)

If a method throws a checked exception, you *must* either (1) handle the exception within the method, or (2) declare the exception in the method header (passing the handling onto the client).

```
public String readString(File inputFile) throws IOException
// do it this way if you DO NOT handle it within the method
```

If you want to handle the exception within the method, use a try-catch block.

```
File f = new File(inputFile);
Scanner input;
try {
    input = new Scanner(f);
}
catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
    System.exit(1);
}
```

To throw an exception, use:

```
throw new ExceptionName("message")
```

3.2 The Comparable Interface

Every class that implements the `Comparable` interface must have a `compareTo()` method. Calling `x.compareTo(y)`

- Returns a negative integer if `x < y`.
- Returns a positive integer if `x > y`.
- Returns zero if `x == y`.

For example, the built-in Java `String` class implements the `Comparable` interface. This allows us to use `compareTo()` to compare strings lexicographicaly.

Note: If `x` and `y` do not have the same types, `compareTo()` throws a `ClassCastException`.

3.3 The Iterator and Iterable Interfaces

An **iterator** is an object which (just as the name might suggest) is used to iterate over a class. Typically, the class in question is an **abstract data type** (ADT)—more on these later. The `Iterator` interface should be implemented by any class which defines an iterator.

```
public interface Interator<T> {
    boolean hasNext();
    T next(); // get next element
    void remove(); // optional
}
```

Using the `Iterable` interface is one way for a class to “give you” an iterator. The advantage of implementing `Iterable` is that it allows you to use the `for each` loop syntax.

```java
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

Note the distinction between `Iterator` and `Iterable`. The `Iterator` interface is implemented by the class in which you define your iterator (i.e., `LinkedListIterator`) while the `Iterable` interface is implemented by the client which uses that iterator (i.e., `LinkedList`).