ap comp-sci:a test cram

for 2023+ test takers 💛
source code license

! this resource assumes you already know somethings about Java
it also does not cover things you don't need to know (take with a grain salt)

helpful resources

- Review Sheets
- 2022 FRQ Scoring Guidelines
- Reference Sheet

introductory

- **printing a string:**

```
System.out.println("With a new line");
System.out.print("Without new line");
```

  - `ln` signifies that a new line will be printed after the supplied content is printed
  - `System.out.print()` `System.out.println()` both can be called without parameters, but `println` is usually called when we just want a new line
  - `System.out.print('\n')` is the same as `System.out.println()`
- **comments** - you can add notes that won't affect your code
  - *regular comments*

```
// this is a one line comment

/*
 * this is a multi line comment
 */
```

  - ⛔ *javadoc style comments* - you will see these mostly in question formatting! don't worry about this!

```
/**
 * my description here!
 * @param my parameter for my method
 * @return what this method returns
 */
```

data types

- **boxed typed**

`java.lang.Integer`, `java.lang.Character`, `java.lang.Boolean`, `java.lang.Double`, `java.lang.Float`, `java.lang.Short`, `java.lang.Long`

  - refers to an `Object` representation of a regular primitive type (int, double, float, etc.).
  - they are primarily used because of Java's nature with generics only accepting `Object` type
  - you can see this in action in a class like `java.util.ArrayList` or identified when the angle brackets are used (<, >)
- **type casting** refers to changing one variable's data type to another
  - we can exchange certain types of data between each other ONLY
  - **explicit cast** when you, the programmer, explicity state the cast

```
Type myVar = (AnotherType) value;
```

  - **implicit cast** when the compiler handles casting automatically by following this ordering:

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

which type is more to the right (more wide) gets the pick by the compiler to be casted to

    - for example, when ever a `double` and `long` are in the same expression, since `double` is more "wide" or higher

up in the ordering, the compiler chooses to cast the original `long` value to a `double`

- **integer** `int`, `java.lang.Integer`
  - represents a whole number or a regular integer
  - is not a floating point number (not a decimal)
  - default initializes to 0 → `int myInt;`
  - to get the max value of the integer type, use `Integer.MAX_VALUE` → `2147483647`
  - to get the minimum value of the integer type, use `Integer.MIN_VALUE` → `-2147483648`
  - if the aforementioned limits are exceeded, an **overflow/underflow** may happen where if the value is too big → becomes super small and vice versa

```
int primitiveInt = 69;
Integer boxedVersion = new Integer(69);
```

- **double** `double`, `java.lang.Double`
  - represents a floating point (aka a decimal)
  - default initialization to `0.0` → `double myUnInitializedDouble;`
  - similar to the `Integer`, you can use `Double.MAX_VALUE` and `Double.MIN_VALUE` to get the appropriate representable range and can suffer from overflow and underflow
  - range from `[1.7976931348623157*10^308, 4.9406564584124654*10^-324]`

```
double primitiveInt = 0.392;
Double boxedVersion = new Double(31.3231);
```

- **character** `char`, `java.lang.Character`
  - represents a keyboard character or some kind of ASCII character (⛔ UTF-8 and other encoding types are not tested are on)
  - since it is a more "visual" way to define an ASCII character, it can be casted to
  - it is an unsigned (meaning without `+ -` signs → only positive or 0) integer in the range of `[0, 65535]`
  - you can think of it as a letter in a word
  - to represent this value, you use the `'` single quote on both side of the character

```
char primitiveCharacter ='c';
Character boxedVersion = new Character('1');
```

- **boolean** `boolean`, `java.lang.Boolean`
  - represents a `true` or `false` value
  - this type is SPECIAL in that it cannot be casted to any other primitive types
  - initialized default to `false`

```
boolean primitiveBoolean = true;
Boolean boxedVersion = new Boolean(false);
```

- **string** `String`, `java.lang.String`
  - they can be thought of as a list of regular `char` values
  - compared to `char`, `String` use `"` (double quotes) to encase its value(s)
  - `String`s are **IMMUTABLE**

```
String str1 = "A";
String str2 = "B";
str1 = str1 + "C";
```

here `str2` would not be the same as `str1` because Java creates a new object everytime the original object is modified, meaning Strings are immutable, where the modified object is discarded

  - you might have seen `\n` which is used to represent the new line control character, you can use the backslash `\` to print special characters or to override ones that are quoted
  - **literal vs object** when a string is initialized with double quotes, Java checks the pool of existing strings to see if it can find duplicates and optimized. on the other hand, with the `new` keyword, we create an object and the Java

runtime does not perform such check and cause unnecessary memory usage (where Objects are located on the heap)
    - the String type is very knitted with its Object counterpart, where you are easily able to use methods on the value like so: `"Hi There!".toLowerCase()`
        1. `toLowerCase()` - makes all possible "A" through "Z" be their lowercase counterparts
        2. `toUpperCase()` - makes all possible "a" through "z" be their uppercase counterparts
        3. `length()` - how many characters make up this String
        4. `substring(int from, int to)` - returns a part of a String (aka returns a String) starting from index `from` and ending with `to - 1`
        5. `substring(int from)` - similar to calling `substring(from, length())`
        6. `indexOf(String str)` - returns the index of the first occurence of the parameter, returns -1 if not found
        7. `equals(String str)` - returns a boolean on whether the two strings are equal based on content **DO NOT USE `==` to check if two are the same based on content and not memory location**

```
String literalString = "hi there";
String objectString = new String(":)");
```

- arrays
    - arrays are like lists
    - they are declared following the type declaration with `[]` like: `Type[] myArray = new Type[size]` where `size` is how many elements the array can hold. the `Type` attribute can be both Objects, primitives, or other arrays of the same type (2D arrays! d ND arrays!) → 2D array: `Type[][] my2DArray = new Type[row_size][column_size]`
    - the size of an array cannot be changed and can only be changed by overwriting it **IMMUTABLE**
    - initializing with pre-existing elements: `Type[] myArray = new Type[]{ element_0, element_1, ...}` notice how the size parameter is not needed. For a 2D array we can follow the following format (for an N-Dimension Array, just append more bracket pairs): `Type[][] myArray = new Type[][]{ {element_0_0, element_0_1, ...}, { element_1_0, element_1_1, ...}, ...}`
    - to access an element we use `myArray[index]`. note that indexing starts at 0, meaning if you want to access element 2, it would be `myArray[1]` (i.e. you do "canonical_index-1") and goes up to `myArray.length-1`. For an n-dimension array, like a 2D array, you have to additional bracket pairs depending on the dimension. For example, a 2D array uses `myArray[row_index][column_index]`
    - if accessing goes out of bounds (i.e. you provide an invalid index), an `ArrayIndexOutOfBoundsException` will be thrown
    - to avoid the aforementioned potential for going out of bound we can utilize the length of the array to help us keep ourself in check with `myArray.length`. *note that is it not a method and is a field builtin into the type itself*
    - when you access an element, you can treat it as both a regular data value or a variable itself, here's an example of swapping two values in an `int[]` array:

```
int tempVariable = myArr[0];
myArr[0] = myArr[myArr.length - 1];
myArr[myArr.length - 1] = tempVariable;
```

classes n objects

- objects `java.lang.Object`
    - objects are instances of classes that pertain certain attributes (think of it as the final product of a

blueprint)

- like arrays and boxed types (which are objects), we use the `new` keyword to create a new object like so: `MyObj variable = new MyObj()`
- due to OOP and Java design, all objects created by the user are a child of `java.lang.Object`
- default initialization of all objects is `null`. if the user tries using the object in any way besides comparison of the raw object, a `NullPointerException` will be thrown. to prevent this, you can achieve a check:

```
if(myObj == null) {
  // handle
} else {
  System.out.println("wow it isn't null!");
}
```

- **local v global** - scope
  - global → declared at the top of the class usually right after the class declaration's first bracket pair:

```
public class MyClass {
  int myGlobalVariable;
}
```

  - local → only the most inner scope (aka bracket pairing):

```
public class MyClass {
 ...

 public void myMethod() {
    int localVariable = 69;
 }
}
```

- **visibility modifiers** - how data is visible to the class `public`, `private`
  - **Data Encapsulation** → "A pillar of OOP that protects data from being accessed or modified by any part of a program, except with explicit calls to the accessor methods and mutator methods"
  - Private → non-accessible and only within the class itself; not exposed
  - Public → Accessible; exposed (a variable without a visibility modifier is automatically marked as public)
- **static vs nonstatic** - `static`
  - **static** → the variable or method belongs to the class and not dependent on the object instance, marked by the `static` keyword
  - **nonstatic** → belongs to the object instance and cannot be accessed with just `MyClass.method()` or `MyClass.field`
- **classes** - blueprints for creating objects
  - they follow this template:

```
public class ClassNameHere {
 private int fieldsArePrivate;

 public ClassNameHere(int val) { // constructor
   this.fieldsArePrivate = val;
 }

 public void setField(int x) { this.fieldsArePrivate = x; }

 public int getField() { return this.fieldsArePrivate; }
}
```

  - getters → return instance fields, in the form of `getFieldName`, should take no arguments and return the respective type
  - setters → should overwrite an instance field's value, in the form of `setFieldName`, should take as many arguments as necessary and return `void`
  - fields/instance variables should always have the modifier `private` (and should also not `final` unless stated)

class mechanics

- **child v parent**
  - child inherit from parent, aka child extends parent literally using the `extends` keyword: `public class ChildClass extends ParentClass { ... }`
  - to go from child to parent, we use the `super` keyword and behaves similarly to the `this` keyword

    - if the parent class specifices a specific constructor (that is one that takes parameters), the child class must call super in a method way: `super(...)` similar to calling another constructor in the same class using `this(...)`
- **inheritance**
    - when the child class extends on the parent, everything from the parent is automatically inherited (like injected) into the child class that is `public`
        1. we can call parent methods from child
        2. the child class can override methods by simply just restating the parent methods and redefining their functionality. when we initialize said class (where class ,`B` extends class `A`: `A e = new B()` every overriden method in B will be called instead of those in A (unless they are not overriden). However, for this edge case, we can access explicit child methods by casting upwards: `((Child)variable).method()`
    - **parent cannot go to child** only the other way around (*"the parent is blind to how many kids it has"*): Similarly this also means the **last** cannot be correct:

```
Parent e = new Parent();
Parent e1 = new Child();
Child e2 = new Child();
Child e3 = new Parent();
```

good luck! :3
~exoad~