Java Object Oriented Approach

creating an immutable type, call-by-value

Java Object-Oriented Approach

Java Object-Oriented Approach

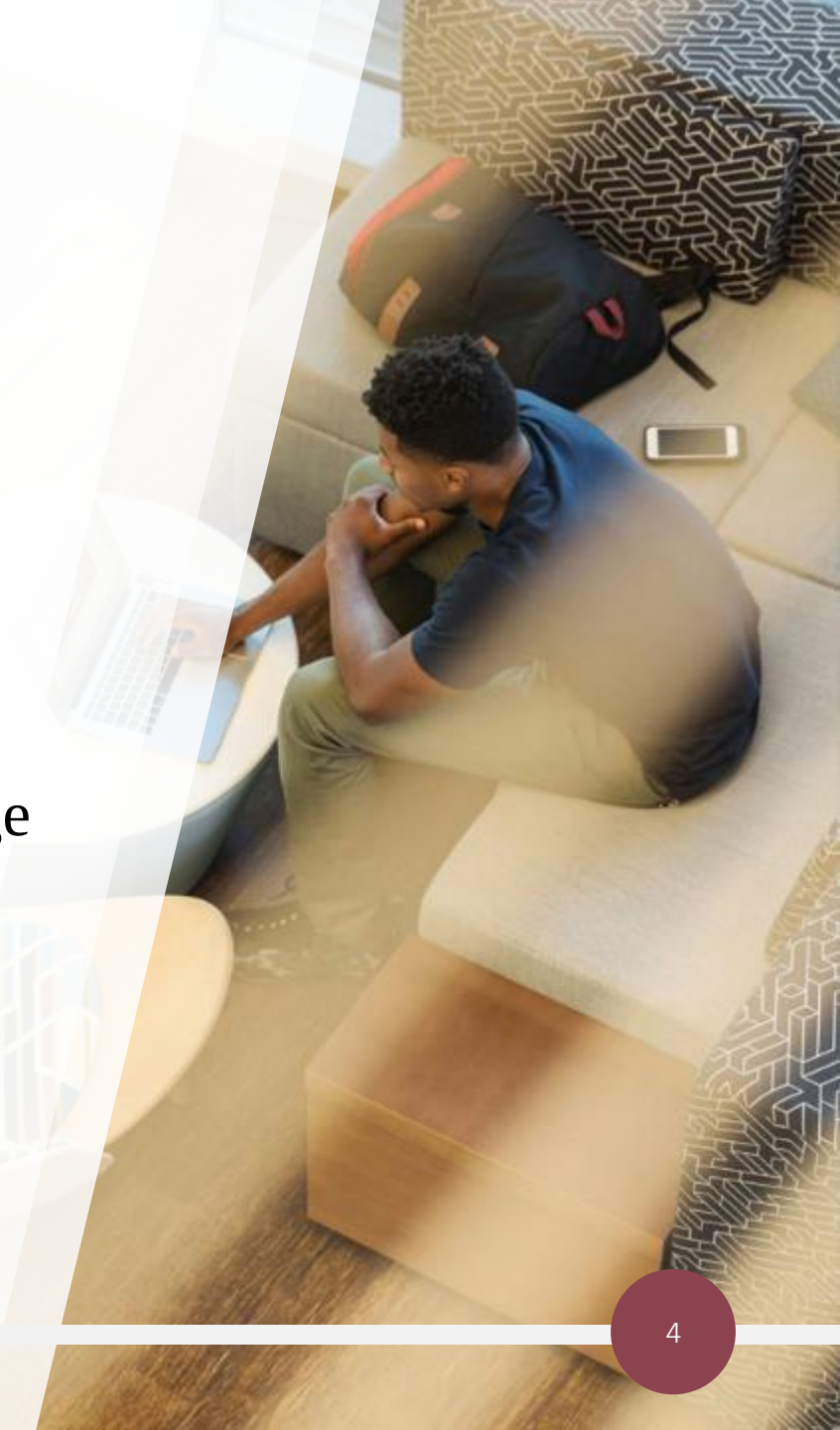
- Declare and instantiate Java objects including nested class objects, and explain objects' lifecycles (including creation, dereferencing by reassignment, and garbage collection)
- Define and use fields and methods, including instance, static and overloaded methods
- Initialize objects and their members using instance and static initialiser statements and constructors
- Understand variable scopes, apply encapsulation and make objects immutable
- Create and use subclasses and superclasses, including abstract classes
- Utilize polymorphism and casting to call methods, differentiate object type versus reference type
- Create and use interfaces, identify functional interfaces, and utilize private, static, and default methods
- Create and use enumerations

Immutability

- Certain API classes are immutable by default e.g. *String*, *Integer*, *Double* etc.. and some are mutable by default e.g. *StringBuilder*.
- While it may look as though the *String* object has changed i.e. the output reflects the changes you requested, in reality a **new** *String* object was created (and the old *String* object can be garbage collected, provided no reference refers to it).

Immutability

- Immutability means something cannot change.
  - primitives – simply the primitive itself
  - objects – both the reference and the object can be made immutable; however, we need to be able to make our own user-defined type immutable
- The keyword "*final*" is important here.
  - if applied to a primitive, the primitive value cannot change
  - if applied to a reference, the reference cannot change
    - however, if the object referred to is mutable e.g. *StringBuilder* then we can change that object.



Immutability Checklist

- Making your own type immutable:
  1. Do not provide any “setter” methods.
  2. Make all the fields *private* and *final*.
  3. Prevent subclassing (prevents overriding):
     a) make the class *final*
     b) make the constructor private and provide a *public static* factory method e.g. “createNewInstance”
  4. Instance fields:
     a) immutable types e.g. *String*, ok
     b) mutable types e.g. StringBuilder, do NOT share references i.e. use “defensive copying” and “advanced encapsulation”

Call-By-Value

- Java supports “call-by-value” when a method is invoked. Simply put, “a copy is made” of something.
- What has been copied?
  - If it is a primitive, then the called method cannot change the primitive value in the caller method (as it is only a copy).
  - If it is a reference, then the called method can change the object that the caller method is using (as the 2 references refer to the same object).