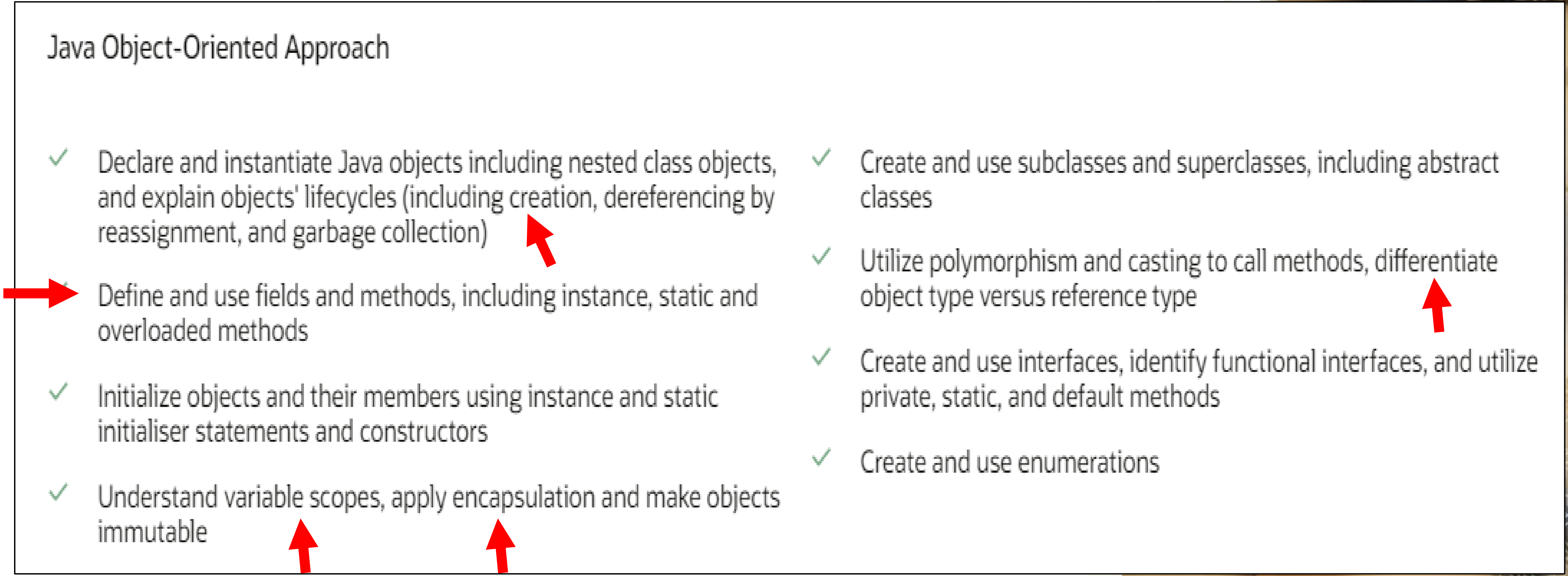
Java Object Oriented Approach

scope, classes, encapsulation, overloading

Java Object-Oriented Approach

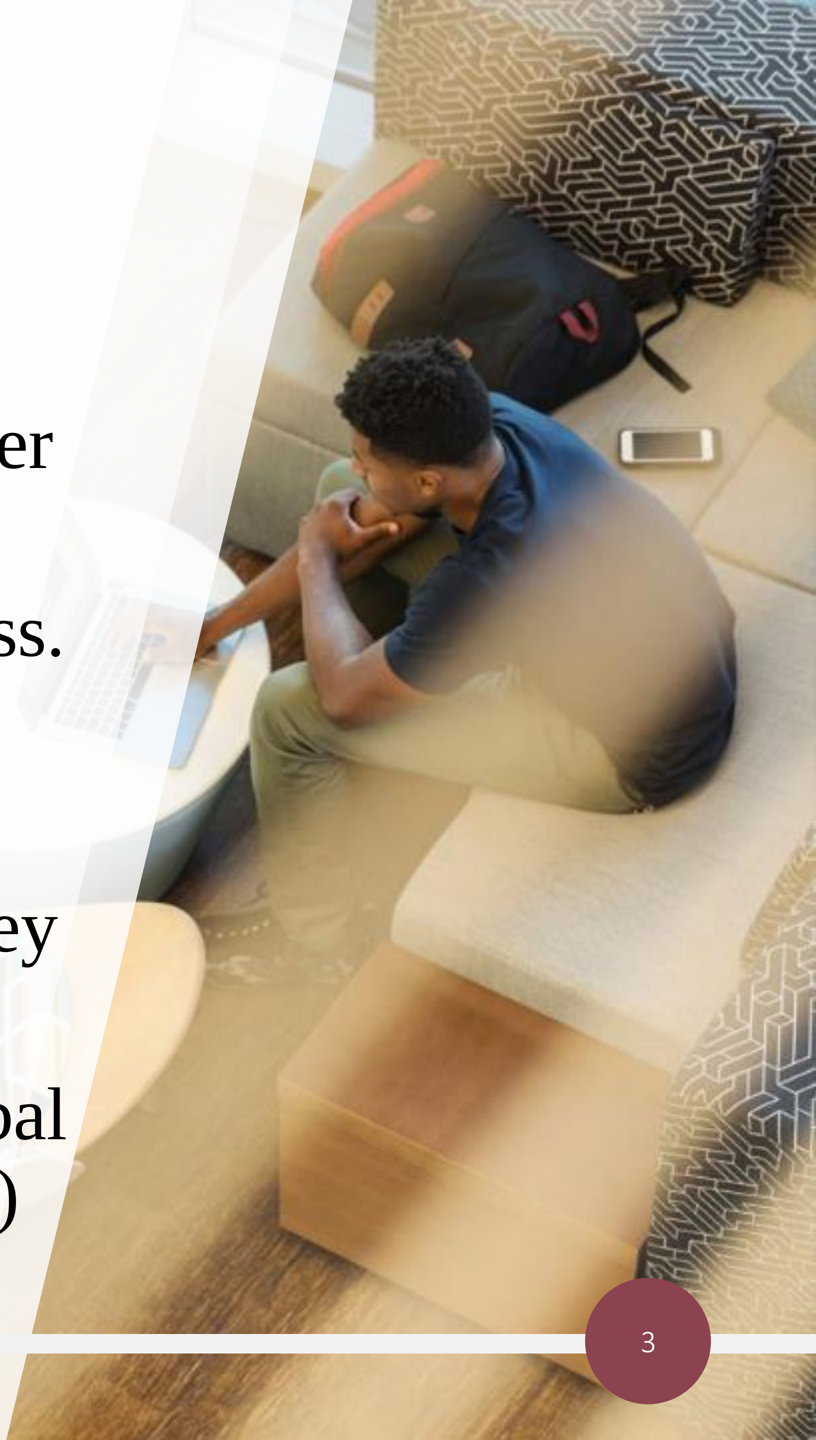


Java Object-Oriented Approach

- ✓ 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

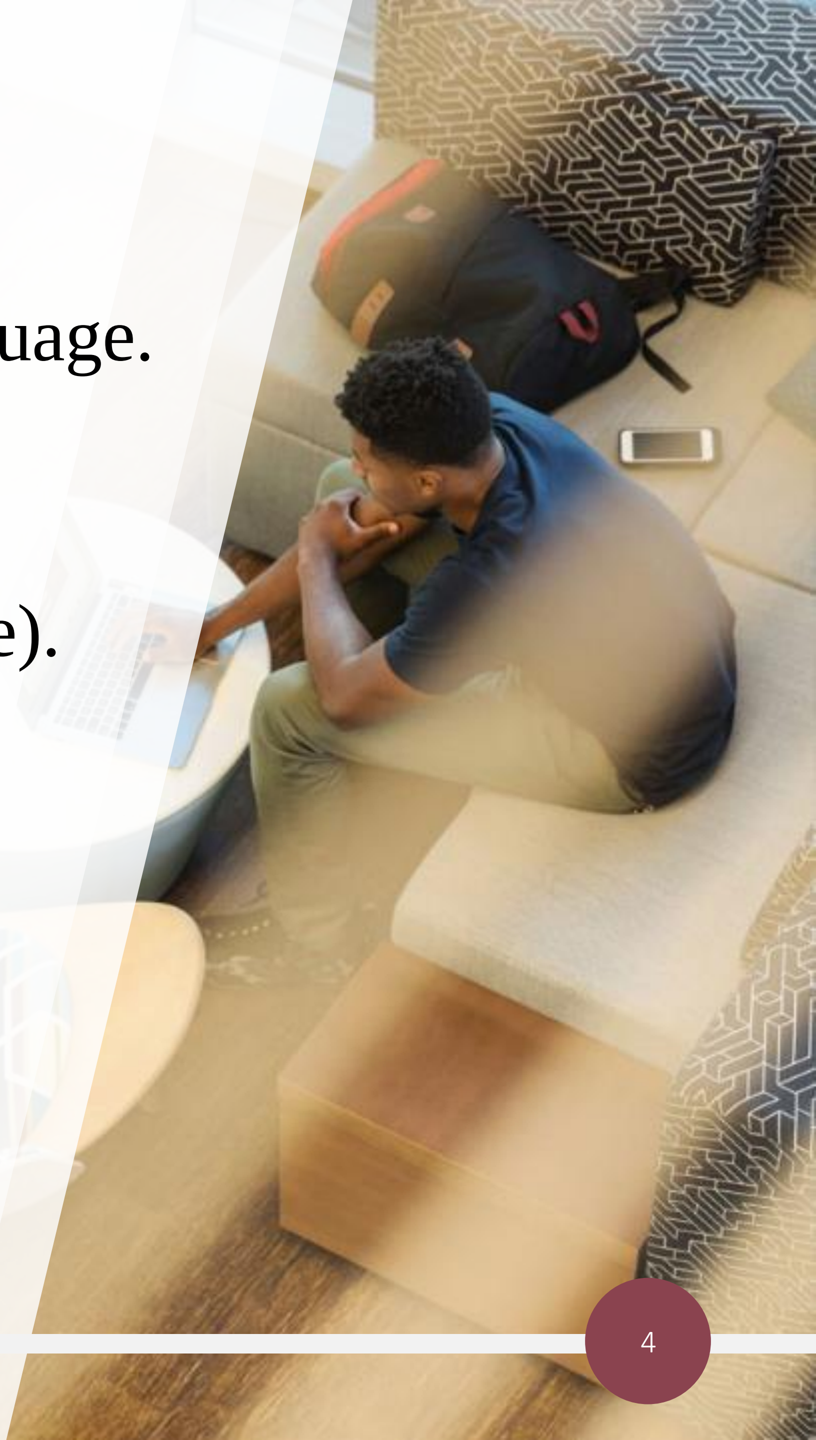
Scope

- Java uses block scope. A block is defined with { }.
- Blocks can be nested e.g. a variable defined in an outer block is visible (in scope) in any inner block.
- Instance and class variables have the scope of the class.
- Method parameters have the scope of the method.
- Local variables have the scope of where they are declared to the end of the enclosing block (in which they are declared).
- Top-level types (classes and interfaces) can have global (*public*) scope and package-private scope (no keyword)



Classes

- Java is an OOP (Object-Oriented Programming) language.
- Classes define a template (plan of a house).
- Objects are the in-memory representation of the class i.e. firstly, you need to create an object (build the house).
- Classes contain both instance and class variables.
  - ➢ instance variables are non-static
  - ➢ class variables are *static*.
- Creating an object:
  - ➢ Person p = new Person(); // reference vs object

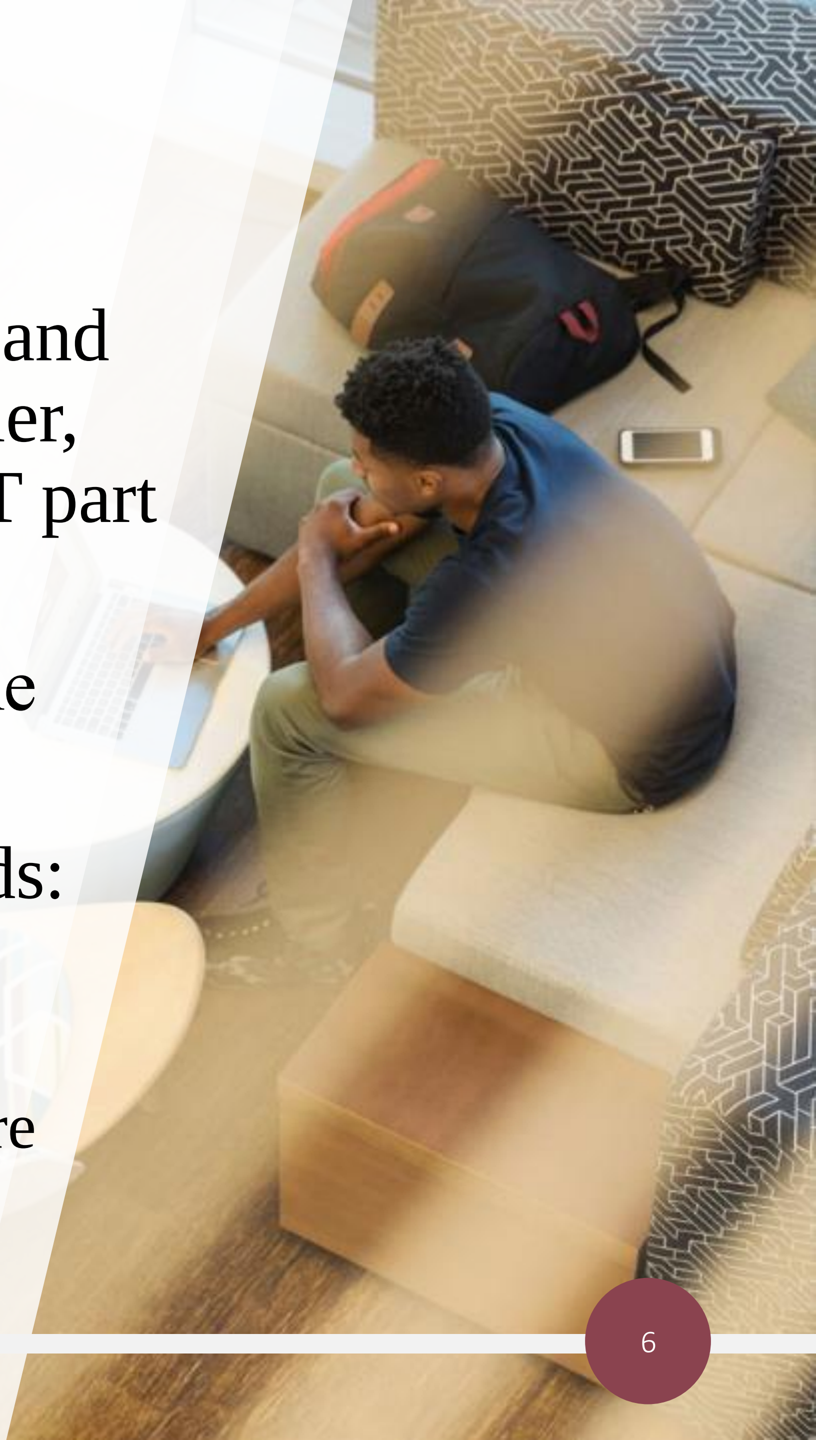


Encapsulation

- This is a core concept in OOP; the principle of "data-hiding".
- Access modifiers enable encapsulation.
- Typically, you keep your data *private* and provide access to the data via *public* methods.
- Access modifiers:
  - ➢ *private* - available to the class only
  - ➢ package-private – no-keyword, available within the same package only
  - ➢ *protected* – available within the same package and also to subclasses in any package (in a specific way)
  - ➢ *public* – available everywhere

Method Overloading

- The signature of a method is the name of the method and it's parameters types and their order. The access modifier, parameter names and crucially, the return type are NOT part of the method signature.
- To "overload" a method means you are using the same name but the parameter types and/or order is different.
- For example, the following are all overloaded methods:
  - ➢public void **m1()**{}
  - ➢public int **m1(int** x){} // return type not part of signature
  - ➢void **m1(double** y){} // access modifier not part of signature
  - ➢public void **m1(String** s, **int** x){} // order important
  - ➢public void **m1(int** x, **String** s){}



this reference

- Refers to the object instance which invoked this instance method.
- Available to instance methods only (compiler error to use *this* in a *static* method).
- The compiler secretly passes the *this* reference in as the first argument to any instance method.
- Useful if parameter names “hide” instance variable names i.e. if a method parameter uses the same identifier as an instance variable, you can use *this* to distinguish the instance member from the parameter.